Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics

David Ellsworth*
AMTI / NASA Ames Research Center
Ling-Jen Chiang†
AMTI / NASA Ames Research Center
Han-Wei Shen‡
Department of Computer and Information Science, The Ohio State University

Abstract

This paper describes a new hardware volume rendering algorithm for time-varying data. The algorithm uses the Time-Space Partitioning (TSP) tree data structure to identify regions within the data that have spatial or temporal coherence. By using this coherence, the rendering algorithm can improve performance when the volume data are larger than the texture memory capacity by decreasing the amount of textures required. This coherence can also allow improved speed by appropriately rendering flat-shaded polygons instead of textured polygons, and by not rendering transparent regions. To reduce the polygonization overhead caused by the use of the hierarchical data structure, we use a fast incremental polygon slicing algorithm. The paper also introduces new color-based error metrics, which more accurately identify coherent regions compared to the earlier scalar-based metrics. By showing experimental results from runs using different data sets and error metrics, we demonstrate that the new methods give substantial improvements in volume rendering performance.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation - Display Algorithms
Additional Keywords: scalar field visualization, volume visualization, volume rendering, time-varying fields, graphics hardware.

1 Introduction

Time-varying data sets are common, and are often difficult to visualize using volume rendering because of their size. Volume rendering can be accelerated by using 3D texture mapping on standard graphics hardware. The volume rendering algorithm for these accelerators loads the volume data into texture memory, and textures a series of polygons as part of the volume rendering process. Most 3D texturing hardware uses dedicated memory to hold the texture data. While many accelerators can render using textures that are larger than the dedicated memory, the rendering is at reduced performance because the texture data must be moved from main memory to the accelerator memory. This limitation particularly affects time-varying volumes since they tend to be large.

Better use of the dedicated volume memory would increase the amount of volume data that can be rendered at full speed. Many volumes have portions that do not vary, or are coherent, in certain regions. Time-varying volumes often have regions that also do not vary within a series of time steps. These spatial and temporal regions of coherence can be exploited by using a data structure introduced by Shen *et al.* [1], the Time-Space Partitioning (TSP) tree. By using this data structure along with a new rendering algorithm, we will show that regions that have spatial coherence can instead be rendered using untextured polygons, and the associated texture memory freed. The data structure will also detect regions that are entirely transparent, which can be skipped during rendering. Regions with temporal coherence can be shared between two or more time steps, thus also saving texture memory. In addition, the reduction in memory means that smaller amounts of textures need to be created, speeding up the texture creation process.

The decision to use untextured polygons or to share regions of volume memory is made by computing error metrics for a hierarchy of regions, or subvolumes, that indicate the amount of spatial and temporal coherence. At runtime, the user specifies spatial and temporal error tolerances. Regions with error tolerances greater than the error metrics are rendered using flat-shaded polygons or voxels from a previous time step. Specifying zero error tolerances result in renderings using data equal to the actual data, but will still result in a smaller memory requirement in many cases.

The error metrics described in the earlier TSP paper [1] were based on the scalar values of the voxels. Since the scalars are mapped into colors using a transfer function, the amount of coherence in the scalar values can be unrelated to the amount of coherence in the colors. This paper introduces color-based error metrics that improve the selection of texture volumes to be loaded into texture memory. Two color-based error metrics are described. One uses the same statistics as the earlier paper but based on the voxel's color values. The second metric uses metrics that are approximations to the first metric. The first metric is quite slow, taking a few to many minutes to compute, but is included to show that the second metric performs similarly even though it is an approximation. The second metric takes a fraction of a second to compute, which allows interactive modification of the transfer function.

The remainder of the paper is structured as follows. Section 2 reviews related work, Section 3 provides a review of the TSP tree

*NASA Ames Research Center, Mail Stop T27A-2, Moffett Field, CA 94035 (ellswort@nas.nasa.gov)
†NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035 (lchiang@nas.nasa.gov)
‡Department of Computer and Information Science, The Ohio State University, 2015 Neil Ave., 395 Dreese Lab., Columbus, OH 43210 (hwshen@cis.ohio-state.edu)

Volume Visualization 2000 Salt Lake City, Utah USA

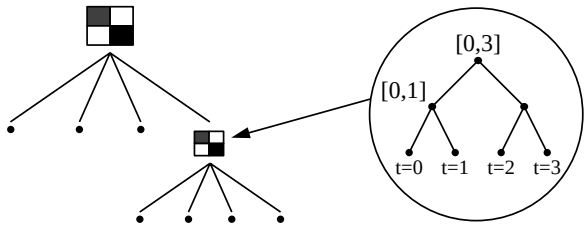

Figure 1: The TSP tree's skeleton is an octree, and each of the TSP tree nodes is a binary time tree. In the example here, the time-varying field has four time steps.

data structure and algorithm, and Section 4 describes how TSP trees can be used for hardware volume rendering. Section 5 discusses the error metrics considered, and the last sections cover the experiments performed, the results, and the conclusions.

2 Related Work

Several earlier efforts have used data coherence to accelerate volume rendering. In general, two types of coherence can be usually observed in a time-varying volume data set. One is called *spatial coherence*, which refers to the fact that voxels in adjacent regions tend to have values that are very close to each other. The other is *temporal coherence*, which refers to the fact that voxels tend not to change drastically from one time step to the next. In the past, researchers have proposed the use of hierarchical data structures to exploit the coherence for speeding up the rendering of steady-state volumes [1, 2, 3, 4]. Laur and Hanrahan proposed a Hierarchical Splatting algorithm [3], where a pyramid data structure is used to store the voxels' mean value and the standard deviation in different subvolumes. Given a user-supplied error tolerance, an octree is fit to the pyramid, and the traversal of the octree allows different regions in the volume to be drawn in different resolutions. Shen *et al.* proposed a hierarchical data structure called the *Time-Space Partitioning (TSP)* tree that decouples the characterizations of temporal and spatial coherence, and allows efficient rendering of time-varying volume data. This work is discussed in more detail in the next section.

Another approach that can significantly speed up volume rendering is to use 3D texture hardware [5, 6, 7, 8]. In essence, 3D texture hardware can be used to perform volume rendering by first generating a sequence of slicing planes perpendicular to the viewing direction, and then use 3D texture hardware to map colors and opacities based on the underlying data attributes to these planes. The sequence of parallel planes are then composited together in a back-to-front visibility order to generate the final image. Yagel *et al.* [9] proposed an algorithm that can efficiently generate the slicing planes based on incremental slicing. LaMar *et al.* [10] proposed using texture hierarchies and spherical shells to render very large data sets. Both of the methods allow fast volume rendering of large data sets. However, the rendering of time-varying data, which requires fast texture animation, was not discussed.

3 Time-Space Partitioning Trees

This paper uses the Time-Space Partitioning (TSP) tree data structure and algorithm, first proposed by Shen *et al.* [1], for capturing both the temporal and spatial coherence in time-varying data. The skeleton of a TSP tree is a standard complete octree, which recursively subdivides the volume spatially until the size of the subvolume is less than a predefined threshold. At each node of the octree skeleton, there is a binary time tree, which recursively bisects the

```
void octree_traverse()
{
  TimeSpan span = time_tree_root.timetree_traverse();
  if (span == Failed)
    add_to_list(subvolume(current_octree_node), span);
  else if (is_leaf(current_octree_node))
    add_to_list(subvolume(current_octree_node),
      curr_time_step)
  else for (each octree_child under current_octree_node)
    octree_child.octree_traverse()
}

TimeSpan timetree_traverse()
{
  if (time_tree_node.temporal_error <= temporal_tol) {
    if (is_leaf(current_octree_node))
      return current_timetree_node.time_span;
    else if (time_tree_node.spatial_error <= spatial_tol)
      return current_timetree_node.time_span;
    else if (is_leaf(current_timetree_node))
      return Failed;
    else
      return child_for_curr_timestep().timetree_traverse();
   }
   else if (is_leaf(current_timetree_node))
     return Failed;
   else
     return child_for_curr_timestep().timetree_traverse();
}
```

Figure 2: TSP tree traversal algorithm.

time-varying data set's time span. Figure 1 depicts a two dimensional version of TSP tree and one of its tree nodes in the form of a binary time tree.

The nodes of the binary time tree store both the temporal and spatial error metrics. Our TSP tree implementation also includes statistics about the corresponding subvolume and time span: the mean, minimum, and maximum scalar values as well as the standard deviation. The earlier work's spatial error metric was the coefficient of variation, which is a normalized standard deviation of the voxels. To quantify a subvolume's temporal error in a given time span, the mean of the individual voxels' coefficients of variation over time was used. This temporal error measurement is more effective in capturing those subvolumes that do not change dramatically even with the presence of high spatial variation [1].

For a time-varying volume data set, the TSP tree can be constructed once and then updated whenever the data used by the error metric calculations changes. Once the tree has been constructed or updated, it can be used repeatedly. The TSP tree traversal algorithm traverses both the TSP tree's octree skeleton and the binary time tree associated with each encountered octree node, as shown in Figure 2. When rendering a frame, the TSP tree is traversed to identify a set of subvolumes that cover the entire volume. Each subvolume is chosen to cover the largest spatial and temporal extent and also satisfy the user-supplied spatial and temporal error tolerances. The tolerance for the spatial error provides a stopping criterion for the octree and time tree traversal. The traversal first descends the octree skeleton, checking if the error tolerance allows the subvolume corresponding to the current node to be added to the subvolume list. The octree traversal is different from the time tree traversal: the octree traversal descends until the entire volume is covered, while the time tree traversal only follows the time spans enclosing the current time step.

Shen *et al.* use the TSP tree data structure to accelerate a software ray casting algorithm for time-varying data [1]. Their rendering algorithm first collects the subvolumes that meet the specified spatial or error tolerance. Then, it renders each of these subvolumes independently into a partial image. The final image is constructed by compositing the colors and opacities of the partial images. In

this implementation, partial images for those subvolumes that have high temporal coherence are cached. The time span for each subimage is also saved. When the user chooses to render the volume at a different time step, the tree traversal process is performed again. During traversals when the viewing parameters remain the same, if a subvolume that has high temporal coherence is encountered and the subimage cached previously is re-usable, then the cached image is directly used, and the re-rendering of the subvolume is entirely skipped. From the experimental studies in [1], the utilization of previously cached images due to high temporal coherence can significantly reduce both the rendering time and I/O overhead.

4 Using TSP Trees for Hardware Volume Rendering

The rendering algorithm described above for a ray-casting implementation must be modified when it is used for hardware volume rendering using 3D texture mapping. The TSP tree construction algorithm does not need to be changed, nor does the initial traversal algorithm need to be changed. Like before, the TSP tree is traversed at rendering time to gather a list of subvolumes to be rendered. These subvolumes may have a larger spatial extent than the octree leaf nodes if the spatial error tolerance caused the traversal algorithm to terminate early. Or, the subvolumes may represent several time steps if the temporal error tolerance caused early termination.

Once the list of subvolumes has been collected, the subvolumes can be rendered. Subvolumes that meet the spatial error tolerance are rendered using flat-shaded polygons. This is an advantage because many graphics systems render flat-shaded polygons faster than 3D textured polygons, and because the associated texture memory is saved. The time to load the volume data into texture memory is also saved. Other subvolumes are rendered using 3D textured polygons. However, if a subvolume meets the temporal error tolerance and represents a time span, then that subvolume uses the texture defined for the first time step in the span. This is the mechanism that causes textures to be shared between time steps.

When using TSP trees, the standard 3D-texture-based volume rendering algorithm must be changed in two ways. The first change is that the volume data are rendered a subvolume at a time, with each subvolume using its own set of slicing polygons to allow both the flat-shaded versus textured polygon decision and the texture sharing decision to be made on a per-subvolume basis. This can be done by using the standard hardware volume rendering algorithm but slicing a single subvolume instead the entire data set. The second major change is that an order must be defined among the subvolumes so that the subvolumes are rendered in back to front order. The order can be determined during the octree traversal by selecting the correct traversal order for the children of the node. Fang *et al.* [11] have devised a solution for parallel projections that examines the signs of components of the view direction vector and from them chooses from eight fixed orderings.

4.1 Algorithm Overhead

While using TSP trees with hardware volume rendering has several important performance benefits, it also has some additional costs. One cost is the overhead of traversing the octree data structure, but this cost is minimal since there are typically only tens or hundreds of octree nodes. A second cost is the increased number of duplicated voxels at the subvolume boundaries. Most hardware volume rendering algorithms only divide the volume data into subvolumes if the data are larger than texture memory. These algorithms only create large subvolumes, which have only a small fraction of duplicated voxels at the boundaries. The TSP-tree algorithm uses small subvolumes, which increases the fraction of duplicated voxels. Subvolumes with $16 \times 16 \times 16$ voxels duplicate 18% of the voxels, and $32 \times 32 \times 32$ subvolumes duplicate 9% of the voxels.

4.2 Slicing Polygons

The third, and largest, additional cost of to the TSP algorithm is the increased number of slicing polygons. There are more slicing polygons compared to the number used with the single-volume hardware rendering algorithms since each of the single-volume polygons is broken up along the subvolume boundaries in our algorithm. The additional polygons require more time to calculate, which must be done whenever the view direction changes. Also, the additional polygons may also take additional rendering time since the polygons must be sent to the graphics system and transformed. However, the additional polygons do not change the number of pixels that must be rendered because the per-subvolume slicing polygons could be merged to form the slicing polygons for an unsubdivided volume.

The cost of generating the additional polygons can be reduced using several methods. One approach would be to save the polygons between frames, and to only generate them when the view direction changes. This will improve the performance during time animations that do not also have viewpoint changes. A second approach would be to reuse the calculated polygons for several subvolumes. This approach would generate a set of slicing polygons for each of the different subvolume sizes, and then reuse this set of polygons for each subvolume of the same size by translating the polygons to the actual location of the subvolume. The approach requires the fewest calculations, but introduces artifacts because the single-subvolume polygons do not match up at the subvolume boundaries. The artifacts can be significant for some viewpoints.

We use a third approach which uses a fast, incremental slicing algorithm to reduce the cost of calculating the slicing polygons. Our algorithm is quite similar to the incremental tetrahedra slicing algorithm developed by Yagel *et al.* [9], and has several similarities to the standard scan-line polygon rasterization algorithm. Figure 3 shows the high-level pseudocode for the algorithm.

The slicing algorithm computes the slicing polygons for one subvolume at a time. This allows most of the data structures to be allocated statically or on the stack. It outputs slicing polygons that have between three and six vertices, which reduces the number of polygon vertices that must be sent to the graphics system compared to generating triangles. The key part of the algorithm is that it maintains an active edge list, which is a list of the edges that intersect the current slice. Each edge in the list stores the intersection between the edge and the current slice; that intersection is a vertex in the current slicing polygon. The active edge list is ordered so that a traversal of the list's edges encounters the edges' stored intersection vertices in the same order as they occur in the current slicing polygon. This avoids the need to compute the convex hull for the three- to six-vertex polygons.

The algorithm also maintains an array of subvolume vertex lists. The array has one more than the number of slices. The first element holds the list of subvolume vertices before the first slice, and the last element holds the vertices after the last slice. The middle elements holds the lists of vertices that are between the element's corresponding pair of slices. The subvolume vertices in each slice's list are sorted by decreasing distance from the view plane (decreasing z). If this sorting was omitted, the algorithm could try to process a vertex which has edges that have no connection to the existing active edge list. Vertices with equal z values must be ordered as well. The algorithm sorts the subvolume vertices using first one of the original subvolume axes that is the most perpendicular to the viewing direction, and then by the second most perpendicular subvolume axis.

```
initslicer()       // called once per frame
{
  Calculate slope of each edge
}
calcslices()       // called once per subvolume
{
  Compute z value for each vertex, and zmin and zmax
  Calculate number of slices using zmin and zmax
  Allocate storage for vertex buckets and slice output
  Clear buckets
  Clear active edge list
  Add each vertex to the vertex buckets, sorting
    vertices in each bucket by z

  for each slice {
    for each vertex in bucket {
      Calculate slice intersection for unprocessed edges

      // update active edge list
      switch (number of unprocessed edges)
      case 3:  // first vertex => edge list is empty
        Add all three edges to active edge list
        break
      case 2:
        Replace processed edge with 2 unprocessed edges
        break
      case 1:
        Replace 2 processed edges with 1 unprocessed edge
        break
      case 0:  // processing last vertex
        Delete active edge list
      }
    }

    Copy slice to output memory using active edge list
    Increment slice intersection for each active edge
  }
}
```

Figure 3: High-level polygon slicing pseudocode.

The algorithm starts by computing the slopes for each edge once per frame. Then, for each subvolume, it computes the z value for the eight subvolume vertices, calculates the number of slices, and allocates the slice bucket headers and the storage for the output polygons. Enough polygon storage is allocated to allow six vertices for each slicing polygon. Memory is reused between subvolumes if the previously-allocated memory is sufficient.

The algorithm then loops for each slice. In each iteration, the algorithm processes the subvolume vertices that are between the current slice and the previous slice. Each vertex's edges are initialized if the edge is not in the current active edge list, and then the active edge list is updated. Different cases handle different numbers of edges that have not been previously placed in the active edge list. The cases handling one or two new edges replace the old edges in the active edge list with the new edge or edges. This avoids having to traverse the edge list to find the location to insert the edge. This edge processing uses a static list of each subvolume vertex's edges. These edges are listed so they are in counterclockwise order when looking at the vertex from outside the subvolume. This allows the edges to be inserted into the active edge list in the correct order.

After the vertices are processed, the active edge list holds the vertices of the current slicing polygon. The slicing polygon's vertices are copied into the output polygon list. Then, each edge's intersection point is advanced to the next slice by adding the edge slope to the current points, and the algorithm starts the processing for the next slice.

Our current polygon slicing implementation is fast enough so that polygons can be recomputed for each frame without making the CPU time the bottleneck. The polygon slicing could become the bottleneck if the subvolumes were smaller, or if the algorithm was run on a system with a CPU that is slower relative to the graphics system.

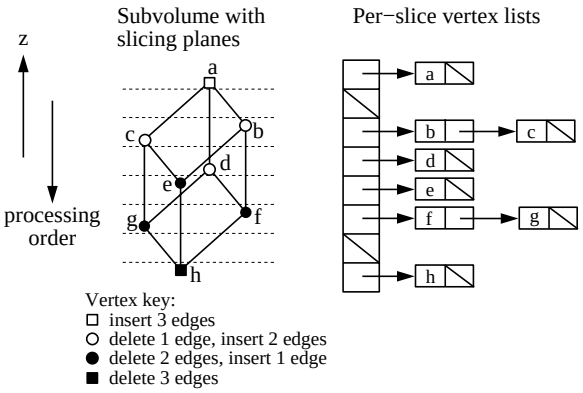

Figure 4: Slicing algorithm data structure.

4.3 Texture Caching

Our algorithm has another optimization that reduces the rendering time. We use the OpenGL function glBindTextureExt that allows the textures to be loaded into texture memory and retained when a time step is first displayed. By not deleting a time step's textures when the algorithm displays later time steps, the time required to display the time step is reduced when it is shown the second and subsequent times. However, this texture caching requires additional logic when the transfer function or error tolerances are changed. When these values are changed, some previously-created textures may no longer be needed, and should be deleted. The texture generation code must maintain data structures so that this texture management can be done correctly.

4.4 Choosing the Subvolume Size

TSP tree algorithms have one important parameter: the minimum subvolume size. Smaller subvolumes allow smaller regions of coherence to be exploited without reducing image quality. If the entire subvolume has high spatial or temporal coherence, the image quality will not be significantly reducing if it, respectively, is replaced with flat-shaded polygons or uses another time step's volume data. Larger subvolumes are less likely have spatial or temporal coherence throughout the subvolume, which means that the TSP tree optimizations cannot be done without reducing image quality.

However, smaller subvolumes have associated costs. One cost is a further increase in the number of slicing polygons, as described above. A second cost is increased overhead in managing textures. We did not expect this to be a large cost, but our OpenGL implementation has shown greatly reduced performance when using more than 4096 textures. Sloan *et al.* [12] have seen similar limitations. Because of this limit on the total number of textures, we were unable to get good results when using $16 \times 16 \times 16$ subvolumes. Instead, we present results using $32 \times 32 \times 32$ subvolumes.

5 Error Metrics

The TSP tree algorithm uses two types of error metrics. Spatial error metrics indicate the amount of coherence within a subvolume, and determine which subvolumes should be rendered without textures. Temporal error metrics indicate the amount of coherence within a series of subvolumes, and determine which subvolumes should share textures between time steps. This section describes

three methods for calculating both types of error metrics. The first method for calculating error metrics bases the metrics on the voxels' scalar values, and are called *scalar-based* error metrics. The second two methods are *color-based* error metrics, and are based on the actual color of the voxels. Although the description of the error metrics in this section assumes that they are used for hardware volume rendering, the error metrics could also be used with the ray-casting implementation by Shen *et al.* [1].

5.1 Scalar-Based Error Metrics

The scalar-based error metrics are the same ones used in the first TSP tree paper [1]. The metrics calculate the coefficient of variation (COV), which is the standard deviation divided by the mean. The COV can be thought of as a normalized version of the standard deviation. The scalar-based spatial error metric e_{ss} is calculated by computing the coefficient of variation of scalars for the voxels in the subvolume and time steps in question, and is given by the following formulas:

$$\bar{v} = \frac{1}{N}\sum_{i,t} v_{i,t} \tag{1}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i,t}(v_{i,t} - \bar{v})^2} \tag{2}$$

$$e_{ss} = \frac{\sigma}{\bar{v}} \tag{3}$$

where $v_{i,t}$ is the value of voxel i at time step t, N is the total number of voxels in the subvolume across the time steps, $\bar{v}$ is the voxel's mean, and σ is the voxel's standard deviation.

The temporal error metric is calculated by first computing a COV for each voxel position within the subvolume for all the voxels in the desired time span $[t_1, t_2]$. The scalar temporal error metric e_{st} is the average of the per-voxel COV's, as shown below:

$$\bar{v}_i = \frac{\sum_{t=t_1}^{t=t_2} v_{i,t}}{t_2 - t_1 + 1} \tag{4}$$

$$\sigma_i = \sqrt{\frac{\sum_{t=t_1}^{t=t_2}(v_{i,t} - \bar{v}_i)^2}{t_2 - t_1 + 1}} \tag{5}$$

$$e_{st} = \frac{1}{n}\sum_{i} \frac{\sigma_i}{\bar{v}_i} \tag{6}$$

where n is the number of voxels in the subvolume, $\bar{v}_i$ is the per-voxel mean, and σ_i is the per-voxel standard deviation. If this error metric seems similar to the previous spatial error metric, note that the spatial error metric calculates a single COV for all the voxels in the subvolume across the time series, while the temporal error metric calculates COV for each voxel in the subvolume, and then averages the COV's across the time series.

If implemented naïvely, these error metrics would require two passes over the voxels, one to compute the mean and a second to compute the standard deviation. However, the formulas can be rearranged so that they only require the sum of all the voxel's values and the square of all the values, as shown in many statistics texts. Because the scalar-based metrics only depend on the data, they can be precomputed and saved in the octree file.

5.2 Reference Color-Based Error Metrics

The color-based error metrics are more accurate because they are based on the color of the voxel, which is more closely related to the image than the scalar value. Their disadvantage is that they must be recomputed when the transfer function is changed, which is often done interactively.

The first color-based error metrics compute statistics based on each voxel's color, much like the scalar-based metrics compute statistics based on each voxel's scalar value. Because the equations are quite similar, we call these error metrics the reference error metrics. Since the standard deviation is not defined for vector quantities such as colors, we instead compute the squared distance in RGBα space between each voxel's color and the mean color of the set of voxels in question. We compute distances in RGB space for higher performance; in the future, we may try computing distances in a more a perceptually uniform color space such as L*u*v* [15]. The squared distance in RGB space is weighted by the opacity of the voxel because low-opacity voxels have a smaller contribution to the final image than high-opacity voxels. The squared distance function d takes two color vectors $\mathbf{c}_1 = (r_1, g_1, b_1, \alpha_1)$ and $\mathbf{c}_2 = (r_2, g_2, b_2, \alpha_2)$, and is:

$$d(\mathbf{c}_1, \mathbf{c}_2) = \alpha_1[(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2] + (\alpha_1 - \alpha_2)^2 \tag{7}$$

where r, g, b, and α have been normalized to the range [0,1]. We weight only by α_1 because $\mathbf{c}_2$ is usually an average color or zero. The mean color values $\bar{r}$, $\bar{g}$, $\bar{b}$, and $\bar{\alpha}$ are computed with equations similar to equation 1. We combine them by computing the square root of $d(\bar{\mathbf{c}}, \bar{0})$, the alpha-weighted distance between the average color $\bar{\mathbf{c}} = (\bar{r}, \bar{g}, \bar{b}, \text{and } \bar{\alpha})$ and the origin. The color-based mean analogue $\hat{\mu}$ is:

$$\hat{\mu} = \sqrt{d(\bar{\mathbf{c}}, \bar{0})} = \sqrt{\bar{\alpha}(\bar{r}^2 + \bar{g}^2 + \bar{b}^2) + \bar{\alpha}^2} \tag{8}$$

We can create the color-based standard deviation analogue $\hat{\sigma}$ by replacing the squared difference between $v_{i,t}$ and $\bar{v}$ in equation 2 by the distance equation 7. The reference color-based spatial error e_{rcs} is the mean replacement divided by the standard deviation analogue. The formulas are:

$$\hat{\sigma} = \sqrt{\frac{1}{N}\sum_{i,t} d(\mathbf{c}_{i,t}, \bar{\mathbf{c}})} \tag{9}$$

$$e_{ncs} = \frac{\hat{\sigma}}{\hat{\mu}} \tag{10}$$

The reference color temporal error equation also computes a per-voxel COV like the scalar-based spatial temporal error equation (6). The analogue to the per-voxel mean value is, like before, $d(\bar{\mathbf{c}}_i, \bar{0})$, where $\bar{\mathbf{c}}_i = (\bar{r}_i, \bar{g}_i, \bar{b}_i, \bar{\alpha}_i)$). Each of the mean values are computed using equations similar to equation 4. The equations for the standard deviation analog and the reference color-based temporal error metric are modified versions of equations 5 and 6 with the scalar value difference replaced by the distance equation. The equations are:

$$\hat{\mu}_i = \sqrt{\bar{\alpha}_i(\bar{r}_i^2 + \bar{g}_i^2 + \bar{b}_i^2) + \bar{\alpha}_i^2} \tag{11}$$

$$\hat{\sigma}_i = \sqrt{\frac{\sum_{t=t_1}^{t=t_2} d(\mathbf{c}_{i,t}, \bar{\mathbf{c}}_i)}{t_2 - t_1 + 1}} \tag{12}$$

$$e_{rct} = \frac{1}{n}\sum_{i} \frac{\hat{\sigma}_i}{\hat{\mu}_i} \tag{13}$$

The reference color-based error metrics have one large drawback: they are very slow. They are slow because every voxel must have its color computed, and then have equations 8 to 13 evaluated using the voxel color. The time to compute the error metric for the three data sets ranged from 4 to 25 minutes; see Table 2 for more details. The computation can be accelerated by optimizing the calculation so it can be made in one pass. Another optimization is to

not recalculate the metrics for subvolumes that do not use any part of the lookup table that was changed in the editing operation. This second optimization can be done efficiently by precomputing the minimum and maximum scalar values for each subvolume. However, these optimizations cannot speed up the calculations so they will run at interactive rates for typical large data sets.

5.3 Approximate Color-Based Error Metrics

This section describes approximations to the reference color-based error metrics that can be computed quickly. As will be shown in the results section, they can be computed in a fraction of a second for reasonably large data sets, and give similar results.

The approximation uses the fact that, if the frequency of occurrence f_k for every unique value x_k is known, the generic standard deviation equation $\sigma = \sqrt{1/n \sum_i (x_i - \bar{x})^2}$ can be rewritten as $\sigma = \sqrt{1/n \sum_k f_k (x_k - \bar{x})^2}$. Our approximation does not actually count the frequency of appearances of the colors in the population, but instead assumes that the counts are normally distributed. We precompute and store in the TSP tree the parameters for each subvolume's distribution, which are the mean and standard deviation of the scalar values in the subvolume. The population distribution uses a distribution equation that only gives a population estimate for every transfer function entry j because the later equations iterate over the entries. The population equation is:

$$p(j) = \exp(-\frac{(x(j) - \bar{v})^2}{2\sigma^2}) \tag{14}$$

where the mean $\bar{v}$ and standard deviation σ are from equations 1 and 2 in the scalar-based error metric section, and $x(j)$ is the scalar value corresponding to the center of transfer function entry j. Next, we define the total estimated population of a subvolume p_{tot} as:

$$p_{tot} = \sum_{j=j_{min}}^{j_{max}} p(j) \tag{15}$$

where j_{min} and j_{max} are the transfer function entries corresponding to the minimum and maximum scalar values of the subvolume. That is, we only iterate here, and in the following equations, over the transfer functions used by the subvolume.

The error metric computes the difference between each transfer function entry and the estimated mean color values. The mean values are computed by multiplying each transfer function entry by the fraction of the population that is expected to use each transfer function entry, $p(j)/p_{tot}$, and summing the products. The estimated mean red value $\bar{r}_{est}$ is :

$$\bar{r}_{est} = \sum_{j=j_{min}}^{j_{max}} \frac{p(j)}{p_{tot}} r(j) \tag{16}$$

where $r(j)$ is the red value of color entry j. The equations for $\bar{g}_{est}$, $\bar{b}_{est}$, and $\bar{\alpha}_{est}$ are similar.

The final steps in computing the approximate spatial color-based error metric are to compute the approximate mean and deviation values. The mean $\tilde{\mu}$ is a combination of the average color values as shown in equation 8. The deviation value $\tilde{\sigma}$ uses the distance function defined earlier to compute the distance between each transfer function entry $\mathbf{c}(j)$ and the average subvolume color $\bar{\mathbf{c}}_{est} = (\bar{r}_{est}, \bar{g}_{est}, \bar{b}_{est}, \bar{\alpha}_{est})$. The distance for each transfer function entry is weighted by the population estimate $p(j)$, and summed as before. The error metric e_{acs} is the deviation divided by the mean.

$$\tilde{\mu} = \sqrt{d(\bar{\mathbf{c}}_{est}, \bar{0})} = \sqrt{\bar{\alpha}_{est}(\bar{r}^2_{est} + \bar{g}^2_{est} + \bar{b}^2_{est}) + \bar{\alpha}^2_{est}} \tag{17}$$

$$\tilde{\sigma} = \sqrt{\sum_{j=j_{min}}^{j_{max}} \frac{p(j)}{p_{tot}} d(\mathbf{c}(j), \bar{\mathbf{c}}_{est})} \tag{18}$$

$$e_{acs} = \frac{\tilde{\sigma}}{\tilde{\mu}} \tag{19}$$

This error metric can be computed quickly since it at most iterates over the number of transfer function entries, which is 256 in our implementation. Typical computation times take a fraction of a second, as shown in Table 2.

We cannot use the population estimate approach for computing an approximate temporal error because it would require storing (or recomputing) a mean and standard deviation for every voxel for every node in the time tree. This would consume more storage than the original data, and would also be slow to compute. Instead, we use a more ad-hoc approach that computes a scaling factor to turn the scalar-based temporal error metric e_{st} into a color-based metric.

This scaling factor is a measure of the amount of variation in the transfer function. The idea is that a large or small amount of variation will magnify or minimize the amount of deviation computed by the scalar-based error metric, The variation measure calculates the distance between the colors in successive transfer function entries; the total scale factor is the square root of the sum of the distances between the entries. The distance measure is the same one used in earlier equations. No normalization is necessary since the color components have been normalized to the range $[0, 1]$. The equation for the approximate color-based temporal error metric e_{act} is:

$$e_{act} = e_{st} \sqrt{\sum_j d(\mathbf{c}(j), \mathbf{c}(j+1)} \tag{20}$$

where the summation is over the transfer function entries excluding the last entry. This error metric can be very quickly computed because the variation measure is only computed once after the color table changes. The individual subvolume error metrics can then be computed by multiplying the precomputed e_{st} for each subvolume by the variation measure.

6 Implementation and Results

We have implemented the TSP tree algorithm using the three sets of error metrics. We also have a non-TSP-tree reference implementation based on the SGI Volumizer subroutine library [13]. Our TSP tree implementation does not use the Volumizer library. We have run experiments to compare the performance with and without TSP trees, and also to show the improved performance of the color-based error metrics. Other experiments show how the error tolerances give the user a tradeoff between image quality and performance.

6.1 Experimental Design

We ran the implementations on three regular, structured grid data sets. Two data sets, Delta and F18, are CFD computations that were originally computed on a curvilinear grid. Because we had access to only one native regular data set, we resampled these data sets' density values onto a regular grid for our experiments. The Delta data set shows a delta wing aircraft flying at a high angle of attack, and was performed on a single curvilinear grid. We resampled the data set at two different resolutions so we could explore the effect of data set size. The data set's main feature is the vortex flow over the wings. The F18 data set was computed using multiple overlapping curvilinear grids. The density values from the data show a vortex structure over the leading-edge extension that breaks up as it

Data Set	Num. Time Steps	Dimensions	Texture Size (MB)	Num. Sub-Volumes (leaf/all)
Small Delta	30	$111 \times 126 \times 51$	20.4	40/49
Large Delta	12	$222 \times 253 \times 103$	66.2	288/337
F18	12	$402 \times 135 \times 103$	64.0	260/313
Shock	25	$512 \times 64 \times 64$	50.0	153/200

Table 1: Experimental data sets.

passes over the wing. The third data set, Shock, was computed on a regular grid, and is a simulation of the unsteady interaction of a planar shock wave with a randomly-perturbed contact discontinuity [14]. The data sets are shown in Figures 10 through 13 as well as in animations of the data sets on the conference CD-ROM. Table 1 gives some statistics about the data. The texture sizes in this table are for one byte per voxel; the original data files use 4-byte floating point values. The sizes in the table do not include the effect of duplicated pixels at subvolume boundaries.

The Delta and F18 data sets were run with a *sparse* transfer function that makes many of the voxels transparent, and reveals the main features of the data. In addition, we ran experiments with the Delta and Shock data set with a *filled* transfer function that makes most of the voxels have a positive opacity. The function shows the slow variations in the data and thus makes it less likely that the spatial error metric will classify a subvolume as coherent. As shown below, the filled transfer function makes it more difficult for our algorithm to improve performance. Fortunately, most volume visualizations use fairly sparse transfer functions.

The experiments were run using OpenGL on a SGI Onyx2 workstation using one of four 195 MHz MIPS R10000 processors, 1GB of main memory, and InfiniteReality2 graphics with 64MB of texture memory and one RM7 Raster Manager. All of the runs used a minimum subvolume size of $32 \times 32 \times 32$ voxels (except for boundary subvolumes), and were rendered at a resolution of 640×480. The slicing plane spacing was half the voxel size (the Nyquist frequency). The experimental runs used the same viewpoint for all the frames, as shown in Figures 10 through 13. However, we noticed informally that the frame times do not vary appreciably when the data sets are rotated.

We measured runs with three error tolerances: a zero error tolerance, a *slight* error tolerance that showed barely noticeable artifacts, and a *moderate* error tolerance that showed small but noticeable artifacts. Figure 14 shows the images with slight and moderate error tolerances for the Delta data set with the filled transfer function. The effects of temporal error can be seen in animations on the conference CD-ROM.

When using non-zero error tolerances, we used error tolerances for the three types of error metrics that gave the same image quality. This was complicated by the fact that using the same error tolerance with different error metrics produces images with different amounts of error, which meant that we had to make multiple runs to search for the correct error tolerances. We measured image quality by computing the average distance between images with zero error tolerance and the ones with some error allowed. The distance was defined as the distance in L*u*v* color space, a perceptually uniform color space [15, 16]. The RGB to L*u*v* conversions assumed a D_{65} white point.

6.2 Results

Overall, the TSP tree algorithm using the color-based error metrics had higher rendering performance than the non-TSP-tree SGI Volumizer implementation, and also higher performance than the TSP

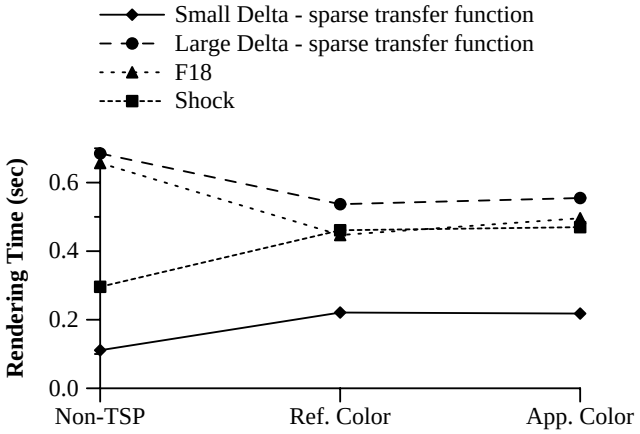

Figure 5: Average rendering times for three error metrics with an error tolerance of zero, and without texture caching.

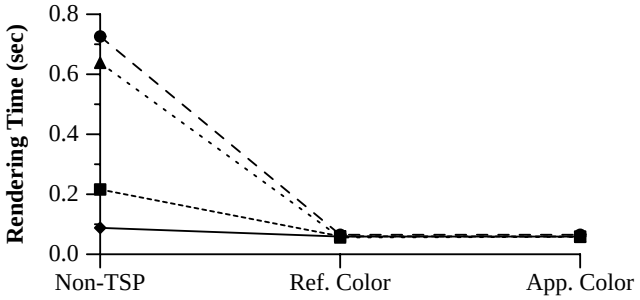

Figure 6: Average rendering times for three error metrics with an error tolerance of zero and texture caching.

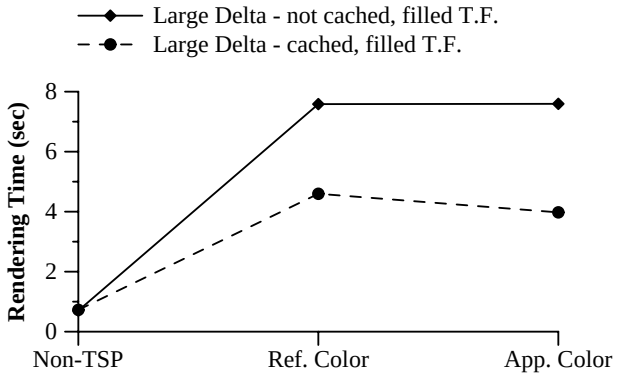

Figure 7: Average rendering times for the large Delta data set using a zero error tolerance and the filled transfer function.

tree algorithm using scalar based error metrics. Tables 3 and 4 show the average time required to render all of each data set's time steps for a number of different configurations. Some of this data is shown graphically in Figures 5 to 8.

Table 5 and Figure 9 contain statistics about the average amount of coherence exploited by the algorithm. The table's first two columns for each error metric show the fraction of subvolumes that pass the spatial error tolerance test so that they can either be not rendered, if the mean opacity was zero, or rendered as polygons. The values are given as percentages of all the voxels to avoid possible distortions caused by over weighting the small subvolumes at the edges of the volume. The third column for each error metric

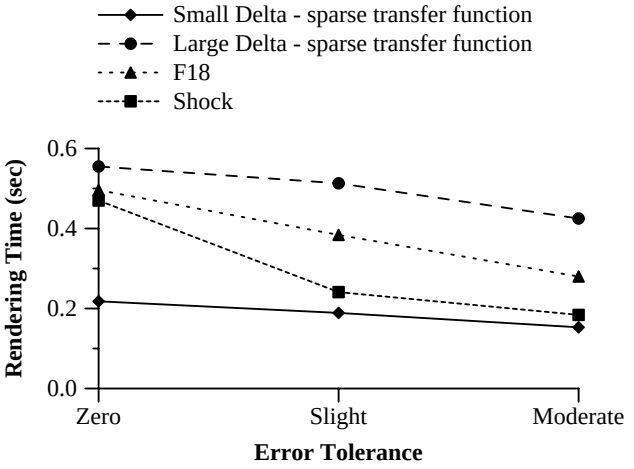

Figure 8: Average rendering times for three error tolerances without using texture caching.

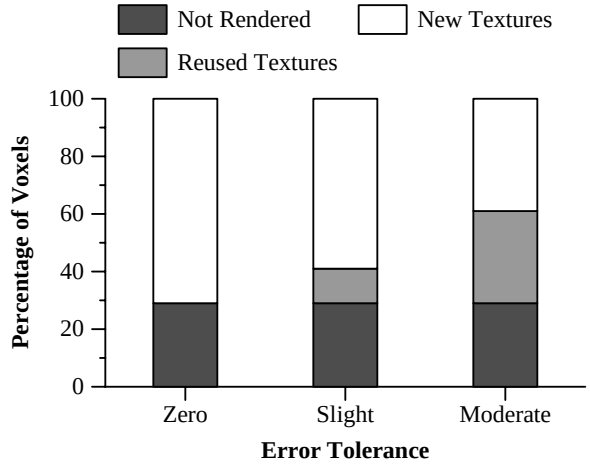

Figure 9: Voxel statistics for the small Delta and the sparse transfer function for three error tolerances.

gives a temporal coherence statistic, the percentage of voxels that were in subvolumes that were reused from an earlier time step. The last column gives the balance of the voxels, the ones in subvolumes that correspond to the current time step.

Non-TSP-tree vs. TSP tree with color error metrics. The new color-based TSP tree algorithm can be compared to the previous non-TSP-tree algorithm by specifying a zero error tolerance. The results are shown in Table 3, and Figures 5 and 7. When textures are being created (non-cached values), the results are mixed. The non-TSP-tree algorithm has higher performance with the smaller data sets, small Delta and Shock. Because the data sets' voxels fit into texture memory, the TSP tree's overhead of more numerous textures and duplicated voxels at subvolume boundaries outweighs the small savings from subvolumes that do not need to be created. The non-cached performance of the TSP tree algorithm is higher with the large data sets, large Delta and F18, because the subvolume size is smaller relative to the data set size. This causes a larger fraction of the subvolumes to be completely filled with transparent voxels, which do not need to be loaded into texture memory. Finally, the zero-error TSP tree performance with the large Delta and the filled transfer function is quite low. The TSP tree only finds a small amount of texture memory savings in this case (see Table 5), so the TSP tree algorithm's overhead increases the time. The performance increases when a moderate amount of error is allowed: the time decreases from 7594 ms to 602 ms for the TSP tree algorithm, which is less than the non-TSP-tree time of 726 ms.

Data Set	Transfer Function	Scalar	Ref. Color	Appr. Color
Small Delta	sparse	21	248	0.21
	filled	21	312	0.21
Large Delta	sparse	170	916	0.33
	filled	175	1117	0.33
F18	sparse	161	887	0.23
Shock	filled	160	1481	0.24

Table 2: Build time in seconds. The scalar metrics are built once, while the color metrics must be built after each transfer function change.

The performance is excellent with the TSP tree algorithm and color-based error metrics once the textures have been created and cached. As shown in Table 3, and Figures 6 and 7, the TSP tree algorithm is nearly always the same speed or faster than the non-TSP-tree algorithm. The best performance with a sparse transfer function is seen with the large data sets, the large Delta and the F18. For example, the rendering time of the F18 decreases from 638 ms to 58 ms when the TSP tree algorithm is used. The exception is with the large Delta and the filled transfer function, where there are no texture savings with a zero error tolerance. The large Delta has better performance when moderate error is allowed, since the TSP tree algorithm can reduce the texture memory requirement so that the frame time is 141 ms instead of the non-TSP-tree's 726 ms.

Scalar-based vs. color-based error metrics. Runs using the color-based error metrics equal or surpass the performance of ones that use the earlier scalar-based error metrics introduced by Shen *et al.* [1]. In some cases, the rendering runs using scalar error metrics took so much time that the times could not be included in Figures 5 and 6 because they would distort the graphs. The relatively low performance happens because the scalar error metric is not based on the actual colors, and thus in order to have the same image quality a larger fraction of the subvolumes must be rendered using textures. For example, Table 5 shows that the scalar error metric required all of the subvolumes to be textured when a zero error tolerance is specified for a data set that is being rendered using a sparse transfer function. The color error metrics located the transparent subvolumes, which meant that textures did not need to be created for them. In addition, the transparent subvolumes were skipped during rendering. When a filled transfer function is used, the runs using scalar error metrics were more comparable to the color error metric runs, but the scalar metric runs were not faster.

Effect of increasing error tolerance. By increasing the error tolerance, the user can trade image quality for speed. Figure 8 shows how the non-cached performance increases as the error tolerance increases. The performance increases because the total size of the textures to be loaded decreases, as shown in Figure 9. In this figure, the New Textures box is the fraction of the overall number of voxels that must be loaded.

The cached performance is different than the non-cached performance. When more error is allowed, the cached performance does not increase in most cases because increasing the amount of error does not decrease the number of subvolumes that is not rendered. When the texture memory is sufficient, not rendering polygons (either textured or flat shaded) is the only way to increase the performance on the InfiniteReality2 graphics subsystem because the subsystem renders flat shaded and tri-linearly interpolated polygons at the same rate [17]. However, the cached performance does increase with the large Delta and the filled transfer function. Here, the average rendering time with the approximate color error metric de-

Model	Sparse Transfer Function								Filled Transfer Function							
	Non-TSP		Scalar		Ref. Color		App. Color		Non-TSP		Scalar		Ref. Color		App. Color	
	NC	C	NC	C	NC	C	NC	C	NC	C	NC	C	NC	C	NC	C
Small Delta	111	88	379	88	221	59	218	59	110	87	378	88	380	88	378	88
Large Delta	685	726	9060	4580	537	65	555	65	702	726	9080	4596	7585	3974	7594	3976
F18	657	638	8278	3734	447	56	496	58	—	—	—	—	—	—	—	—
Shock	—	—	—	—	—	—	—	—	296	216	1049	1307	461	60	470	59

Table 3: Rendering times, averaged over all the time steps, with a zero error tolerance specified. The columns give times, in milliseconds, when using the scalar, reference color, and approximate color error metrics for when textures are cached (C) and not cached (NC).

Model	Error Tolerance	Sparse Transfer Function						Filled Transfer Function					
		Scalar		Ref. Color		App. Color		Scalar		Ref. Color		App. Color	
		NC	C	NC	C	NC	C	NC	C	NC	C	NC	C
Small Delta	slight	214	88	185	59	189	59	216	88	214	88	218	87
	moderate	181	88	156	59	153	59	180	87	180	88	180	88
Large Delta	slight	2508	1985	517	63	513	64	2630	2079	1105	1502	885	1481
	moderate	591	142	495	64	425	64	570	153	525	156	602	141
F18	slight	2819	2056	367	57	384	57	—	—	—	—	—	—
	moderate	666	156	273	57	280	57	—	—	—	—	—	—
Shock	slight	—	—	—	—	—	—	458	59	265	60	241	51
	moderate	—	—	—	—	—	—	356	59	198	60	184	51

Table 4: Average rendering times when some error is allowed. The columns give times, in milliseconds, when using the scalar, reference color, and approximate color error metrics for when textures are cached (C) and not cached (NC).

Model	Transfer Function	Error Tolerance	Scalar				Reference Color				Approximate Color			
			% NR	% UP	% RT	% NT	% NR	% UP	% RT	% NT	% NR	% UP	% RT	% NT
Small Delta	sparse	zero	0	0	0	100	29	0	0	71	29	0	0	71
		slight	0	0	40	60	29	0	13	58	29	0	12	59
		moderate	0	0	60	40	29	0	32	39	29	0	32	39
	filled	zero	0	0	0	100	0	0	0	100	0	0	0	100
		slight	0	0	40	60	0	0	42	58	0	0	40	60
		moderate	0	0	60	40	0	0	63	37	0	0	60	40
Large Delta	sparse	zero	0	0	0	100	66	0	0	34	66	0	0	34
		slight	0	0	42	58	65	0	2	33	65	0	2	32
		moderate	12	0	56	32	66	0	7	27	64	0	8	26
	filled	zero	0	0	0	100	0	9	0	91	0	9	0	91
		slight	0	22	18	59	0	16	42	42	0	42	16	42
		moderate	0	22	47	31	0	15	59	26	0	42	25	33
F18	sparse	zero	0	0	0	100	61	0	4	33	62	0	0	38
		slight	0	0	40	60	61	0	11	26	62	0	10	28
		moderate	3	0	60	37	60	0	20	18	62	0	20	18
Shock	filled	zero	0	31	0	69	0	44	1	55	0	43	0	57
		slight	0	31	17	51	0	36	37	27	0	63	11	27
		moderate	0	30	30	40	0	36	46	18	0	61	21	18

Table 5: Statistics on how subvolumes were rendered, given as a percentage of voxels in each case. Key: % NR = not rendered, subvolume was transparent; % UP = rendered as untextured polygons; % RT = rendered using reused textures, from an earlier timestep; % NT = rendered using new textures, textures loaded specifically for the current time step.

creases from 3976 ms to 141 ms when moderate error is allowed. This happens because the increased texture coherence used reduces the texture memory requirements to be less than the texture memory capacity.

Texture caching. Retaining textures increases the performance in most cases. Texture caching increases the performance dramatically with nearly all of the TSP tree runs. For example, the small Delta rendering time with the approximate color error metric decreases from 218 ms to 59 ms after the textures are cached. The time decreases by a factor of 8 for the large data sets, the large Delta and F18. More cases can be seen by comparing the color error metric values between Figures 5 and 6. However, in a few cases, the cached time is higher than the non-cached time. We believe that this occurs when the total size of the textures is only slightly larger than the texture memory. When a texture is to be loaded into a graphics subsystem that has no free texture memory, the textures resident in memory may need to be moved so that there is enough contiguous memory available for the new texture. This can make the initial texture loading faster because loading textures into free texture memory is faster. In these cases, better performance would be achieved by disabling texture caching. For the larger number of cases where texture caching is effective, the use of texture caching dramatically speeds up the animation of time-varying data sets.

Reference vs. approximate color-based error metrics. Fi-

nally, the rendering times and the subvolume statistics are quite similar for comparable runs using the reference and approximate color based error metrics. This shows that the approximations used in the approximate metrics are valid for the data sets tested. The main differences are seen in the runs using a filled transfer function and a nonzero error tolerance. In these runs, the reference error metric causes textures to be reused more than the approximate color metric (see Table 5), which instead uses untextured polygons. Even though the two error metrics do use nearly the same fraction of the overall textures, the texture usage difference causes some differences in performance. The texture usage differences happen because we did not have an effective way to measure perceived spatial and temporal image quality independently. Instead, our method of finding error tolerances by only comparing the overall image error appears to have found error metrics that cause the two error metrics to perform differently.

The advantage of the approximate error metrics is that they can be recomputed in at most 0.33 seconds. This time is much less than the 4 to 25 minutes required for the reference color-based error metrics. The recomputation times are shown in Table 2. The fast recomputation is important because it allows the transfer function to be changed interactively, which is a common operation. Because the timings in this table give the time to compute the metrics for all the time steps, the approximate error metrics could allow an even faster response by only computing the metrics needed for the current time step, and computing the other time steps' metrics later.

7 Conclusions and Future Work

We have presented a fast volume rendering algorithm using 3D texture hardware for visualizing large-scale time-varying data sets. Utilizing a hierarchical data structure called the TSP tree, we are able to exploit the spatial and temporal coherence that exists in time-varying fields and substantially reduce the amount of texture memory that is required. The fast volume rendering is achieved by rendering a combination of flat-shaded and solid-textured polygons, where flat-shaded polygons are used to represent those regions having high spatial coherence, and the solid-textured polygons are used to represent regions having high variation, both in spatial and temporal domains. We have presented a fast incremental slicing algorithm that reduces the overhead of generating additional parallel sample planes due to the use of multiple subvolumes. In addition, we have developed color-based error metrics that more accurately identify spatial and temporal coherence compared to the scalar based error metrics used by most of the existing hierarchical volume rendering techniques. Our fast approximate color-based error metric, which is orders of magnitudes faster than a naïve color-based error metric, enables the user to change the transfer function interactively. Finally, we have presented results from experimental studies that show that we can overcome the limitation of texture memory capacity and significantly speed up the time-varying volume rendering using 3D texture hardware.

One area of possible future work is improved error metrics. One possible improvement is to compute color differences in a perceptual color space instead of RGBα space. This might give more accurate error metrics. While the RGB to perceptual color space conversion adds computation, the additional computation is minimal for the approximate color-based error metrics because only the transfer function needs to be converted. Other error metric work includes adding population estimates to the approximate temporal color-based error metric, and evaluating error metrics that do not use α-weighting. A second area of future work would be to explore the effect of the subvolume size. If smaller subvolumes were used, more coherence would be exposed, but the algorithm would most likely need to manage texture memory as was done by Sloan *et al*. [12]. Finally, there may be advantages to using the color-based error metrics with a TSP-based ray casting volume renderer.

Acknowledgments

This work was supported in part by NASA contracts NAS2-14303 and DTTS59-99-D-00437/A61812D. We would like to thank Neal Chaderjian, Ken Gee, Scott Murman and Ravi Samtaney for providing the data sets. We also thank Chris Henze, Pat Moran and other members in the Data Analysis Group at NASA Ames Research Center for their helpful comments and technical support.

References

[1] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In *Proceedings of Visualization '99*, pages 371–377. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[2] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[3] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proceedings of SIGGRAPH 91*, pages 285–287. ACM SIGGRAPH, 1991.

[4] J. Wilhelms and A. Van Gelder. Multi-dimensional tree for controlled volume rendering and compression. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 27–34. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[5] K. Akeley. RealityEngine graphics. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 109–116, August 1993.

[6] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 91–98, 1994.

[7] O. Wilson, A. Van Gelder, and J. Wilhelms. Direct volume rendering via 3D textures. *UCSC Technical Report, UCSC-CRL-94-19*, 1994.

[8] T. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. *UNC Technical Report, TR93-0027*, 1993.

[9] R. Yagel, D. M. Reed, A. Law, P.-W. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *1996 Volume Visualization Symposium*, pages 55–62. IEEE Computer Society Press, Los Alamitos, CA, October 1996.

[10] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive textured-based volume visualization. In *Proceedings of Visualization '99*, pages 355–361. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[11] S. Fang, R. Srinivasan, S. Huang, and R. Raghavan. Deformable volume rendering by 3D texture mapping and octree encoding. In *Proceedings of Visualization '96*, pages 73–80. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[12] P.-P. Sloan and C. Hansen. Parallel lumigraph reconstruction. In *Proceedings 1999 IEEE Parallel Visualization and Graphics Symposium*, pages 7–14. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[13] SGI, Mountain View, CA. *OpenGL Volumizer Programmer's Guide*, 1998. Document Number 007-3720-001.

[14] D. I. Meiron and R. Samtaney. 3D simulations of the Richtmyer-Meshkov instability with re-shock. *Bulletin of the American Physical Society*, 43(9):2104.

[15] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Fundamentals of Interactive Computer Graphics*, page 584. Addison-Wesley Publishing Company, second edition, 1990.

[16] C. Poynton. Frequently asked questions about color. http://www.inforamp.net/~poynton/Poynton-color.html, December 1999.

[17] SGI, Mountain View, CA. *Onyx2 Reality, Onyx2 InfiniteReality, and Onyx2 InfiniteReality Technical Report*, August 1998. http://www.sgi.com/onyx2/tech_report.pdf.

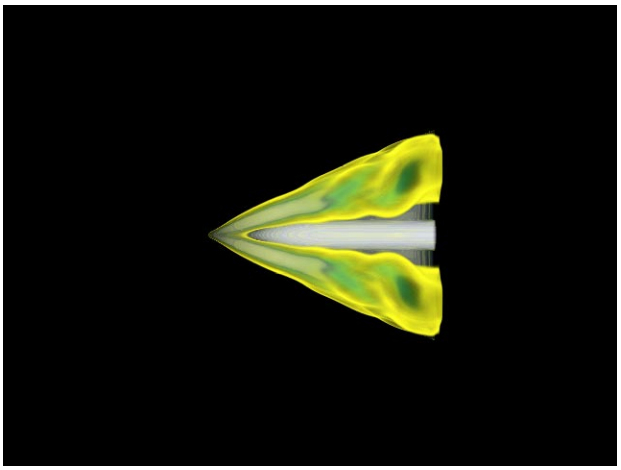

Figure 10: Large Delta with zero error tolerance and the sparse transfer function.

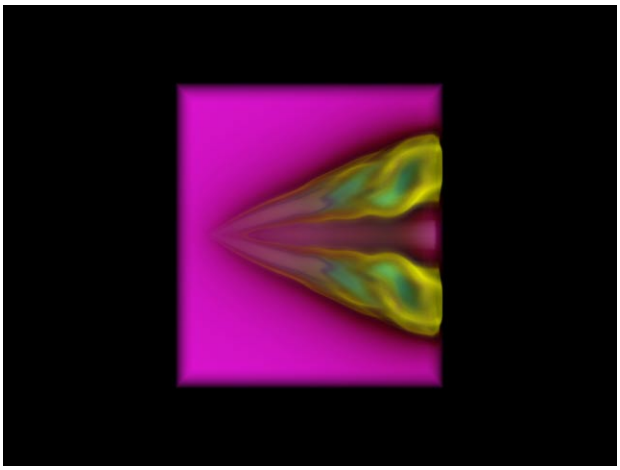

Figure 11: Large Delta with zero error tolerance and the filled transfer function.

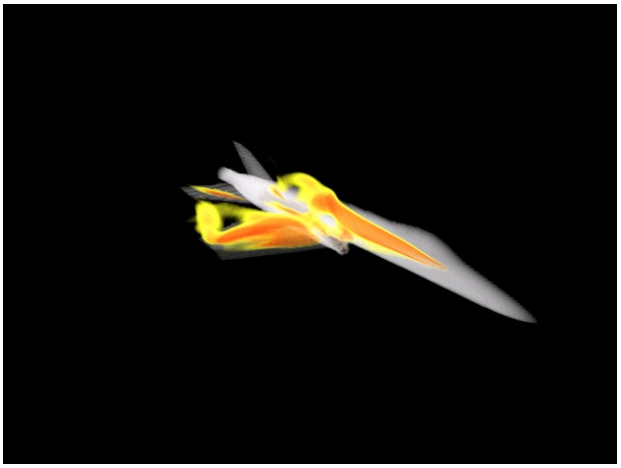

Figure 12: F18 with zero error tolerance.

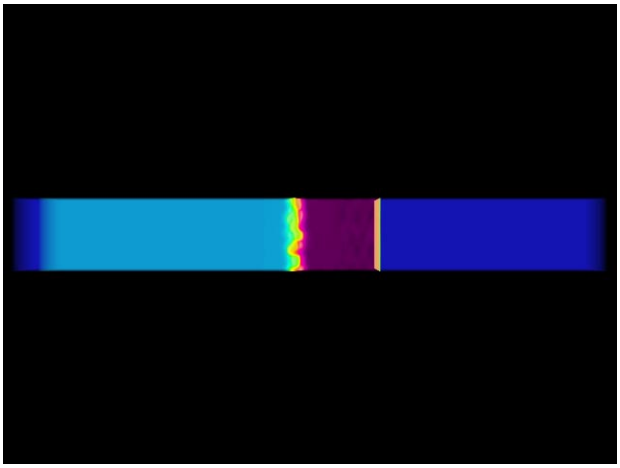

Figure 13: Shock with zero error tolerance.

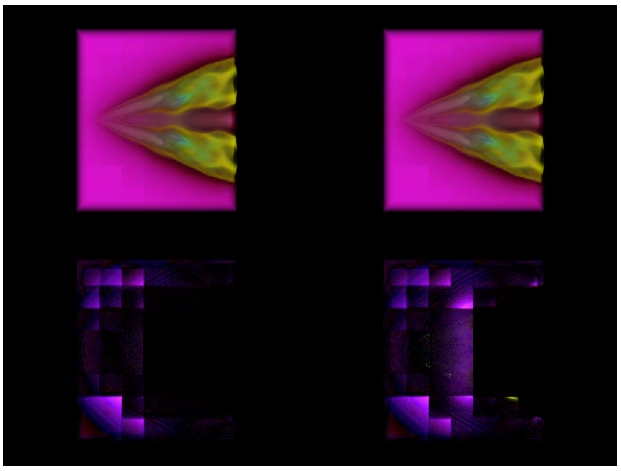

Figure 14: Large Delta with slight (left) and moderate (right) error and the filled transfer function, plus contrast-enhanced differences from the zero-error-tolerance image.

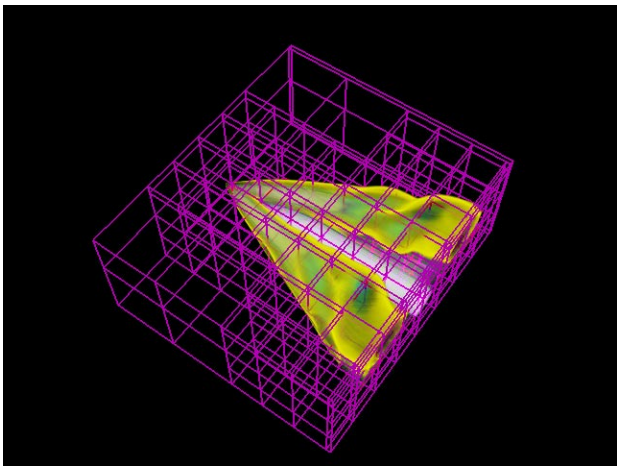

Figure 15: Large Delta with sparse transfer function and lines showing the subvolume boundaries. Large subvolumes have high spatial coherence.