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 is 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 introduce an optimization method using polygon templates. 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.

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 use 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] called Time-Space Partitioning (TSP) trees. 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. 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 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 a few seconds 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 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 dataset. 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

*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)

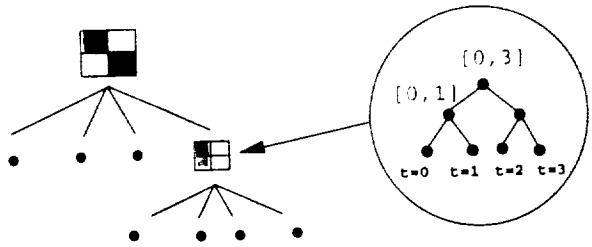

Figure 1: The TSP tree's skeleton is an octree, and each of the TSP tree nodes is a binary time tree. In the example here, the time-varying field has four time steps.

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 to use texture hierarchical and spherical shells to render very large data sets. Both of the methods allow fast volume rendering of large datasets. 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 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].

```
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.

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. 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 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 is 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 Polygon Templates

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 tens or hundreds of octree nodes. The main additional cost is due to the increased number of slicing polygons. There are more slicing polygons compared to 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 does 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 the entire 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 use the incremental slicing algorithm described by Yagel *et al.* [9] to reduce the cost of calculating the slicing polygons.

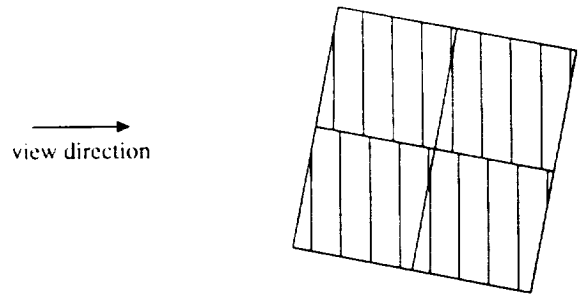

Figure 3: Example of polygon mismatch between subvolumes when polygon templates are used.

We use a third approach that reuses the calculated polygons for several subvolumes. The algorithm generates a set of slicing polygons for each of the different subvolume sizes. It then reuses this set of polygons for each subvolume of the same size by translating the polygons to the actual location of the subvolume. Usually, about a dozen sets of polygons must be generated: eight for the eight possible leaf node subvolume sizes (each dimension may either be full size or a partial size), plus a few larger subvolumes for internal leaf octree nodes that are being rendered using flat-shaded polygons. We reduce the number of vertices that must be sent to the graphics system by generating polygons with three to six vertices instead of only generating triangles.

Using polygon templates speeds up the overall rendering algorithm, since fewer polygons must be calculated. One consequence of using polygon templates is that artifacts are occasionally visible at the subvolume boundaries. The artifacts are due to the slicing polygons not matching up at the boundaries, as shown in Figure 3. We cannot see these artifacts in Figures 4 through 9. The slicing polygons can only match up if polygons are calculated for every subvolume. In the future, we hope to implement Yagel *et al.*'s algorithm [9] to see whether it speeds up the slicing polygon calculations so they are not the bottleneck.

4.2 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.

4.3 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, it can be respectively replaced with flat-shaded polygons or use another time step's volume data without significantly reducing the image quality. 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 not able to run experiments using the scalar-based error metrics with $16 \times 16 \times 16$ 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 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 distance in RGBα space between each voxel's color and the mean color of the set of voxels in question. The 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 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]. 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 $d(\bar{\mathbf{c}}, 0)$, 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:

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

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, $d(\bar{\mathbf{c}}_i, 0)$, 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:

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

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

$$e_{rct} = \frac{1}{n} \sum_i \frac{\hat{\sigma}_i}{\hat{\mu}_i} \quad (13)$$

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 20 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 at most a few seconds 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:

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

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 need to define the total estimated population of a subvolume p_{tot} since we will refer to it in several equations:

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

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 :

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

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.

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

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

$$e_{acs} = \frac{\tilde{\sigma}}{\tilde{\mu}} \quad (19)$$

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 at most about 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:

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

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]. 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 two Cartesian grid data sets. The Delta Wing dataset is a CFD computation of a delta wing aircraft flying at a high angle of attack, and was performed on a single curvilinear grid. We resampled the data set's density values onto a Cartesian grid for our experiments. The resampling was performed twice 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 dataset is also a CFD computation, but 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 passes over the wing. 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 floating point values and thus require 4 bytes per voxel.

Both data sets were run with a *sparse* transfer function that many of the voxels transparent, and reveals the main features of the data. We also ran experiments with the Delta Wing data 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 more difficult for the spatial error metric to classify subvolumes as coherent.

The experiments were run using OpenGL on a SGI Onyx2 workstation using one of two 195 MHz MIPS R10000 processors, 512MB of main memory, and InfiniteReality graphics with 64MB of texture memory. All of the runs used a minimum subvolume size of 32 × 32 × 32 voxels (except for boundary subvolumes), and were rendered at a resolution of 640 × 480. The experimental runs for each data set were run using the same viewpoint for all the frames, as shown in Figures 4 through 8. The tables show values that are averages of the per-frame values.

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. 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 [14, 15]. The RGB to L*u*v* conversions assumed a D_{65} white point.

6.2 Results

Tables 3 and 4 show the average rendering speed for several combinations of model, error metric, error tolerance, transfer function, and use of TSP trees. The non-TSP tree algorithm could not be run with all of the time steps for the two larger data sets, the large Delta Wing and the F18. These runs with all 12 time steps failed due to lack of memory, so the entries are for runs using 4 time steps. We were not able to determine whether the failure was due to our inefficient use of the Volumizer library or to the library requiring more memory than was available.

Table 5 give statistics about average the amount of coherence exploited by the algorithm. The 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 give as percentages of all the voxels to avoid possible distortions caused by overweighting the small subvolumes at the edges of the volume. The next column 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.

Data Set	# Time Steps	Dimensions	Texture Size (MB)
Small Delta Wing	30	111 × 126 × 51	20.4
Large Delta Wing	12	222 × 253 × 103	66.2
F18	12	402 × 135 × 103	64

Table 1: Experimental datasets.

Data Set	Transfer Function	Scalar	Ref. Color	Appr. Color
Small Delta Wing	sparse	21	262	0.21
	filled	20	320	0.21
Large Delta Wing	sparse	231	966	1.0
	filled	253	1167	3.0
F18	sparse	246	949	1.0

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

Our results in Table 3 show that the TSP tree algorithm with the color error metrics and the sparse transfer function has higher performance than the non-TSP-tree algorithm when the textures have previously been loaded into texture memory. The TSP tree algorithm runs at 30 ms per frame, twice as fast with the small Delta Wing, which runs at 60.6 ms per frame. With the large Delta Wing, the TSP-tree algorithm runs four times faster than the non-TSP-tree: they respectively run at 33 and 129 ms per frame. The speed difference is largely due to the 29% of the data that did not need to be rendered because the TSP tree detected that it was transparent. The TSP-tree F18 runs with the color error metrics was also faster than the non-TSP-tree runs (30 versus 388 ms) even though it had to handle 12 time steps of data instead of four.

The zero-error runs with the Delta Wing with the filled transfer function are harder to interpret (see Table 3). We cannot explain the reason that the TSP tree was faster with the small Delta Wing because, as shown in Table 5, the TSP tree algorithm used all of the volume data. The difference may be due to different amounts of optimization in the implementations. The TSP-tree and non-TSP-tree large Delta wing runs cannot be compared since they are for different numbers of time steps. The TSP tree runs are slow because the graphics system is moving textures in and out of texture memory during rendering. This texture is too large even though the 7% of voxels rendered as polygons reduces the 66MB of textures to 61MB, less than the 64MB capacity, because there are two inefficiencies: the texture subvolumes require that the border voxels be replicated, and the graphics system may not be able to use all of the texture memory due to memory fragmentation.

The runs using color-based error metrics clearly give better performance than the ones that use scalar-based metrics. The most dramatic examples are with the large Delta Wing with the transparent transfer function, and with the F18. The cached rendering runs using color error metrics are more than 100 times the speed of the scalar error metric runs (see Table 3). The timings for scalar- versus color-based error metrics are 5.4 s versus 33 ms for the large Delta Wing, and 3.6 s versus 28 ms for the F18.

The non-cached runs are not as impressive. The zero-error TSP-tree runs are slower than the non-TSP-tree runs. This is due to the additional overhead of creating and managing many small textures compared to fewer large textures. We may be able to decrease the difference in speed by optimizing our texture creation routines since they have not yet been optimized. If moderate image error is acceptable, the non-cached runs can run faster than the zero-error

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 Wing	237	60.6	804	46.0	488	30.2	487	30.0	232	60.6	806	45.1	805	45.0	807	45.1
Large Delta Wing	1.49s*	129*	20.0s	5.40s	2.30s	33.0	2.42s	33.0	1.54s*	134*	18.1s	4.78s	16.5s	4.16s	21.8s	4.90s
F18	1.43s*	388*	17.1s	3.62s	1.50s	28.0	2.01s	28.0	—	—	—	—	—	—	—	—

Table 3: Average rendering times with zero error. The columns give times, in milliseconds unless otherwise indicated, when using the scalar, reference color, and approximate color metrics for when textures are cached (C) and not cached (NC). *With the non-TSP implementation, the large Delta Wing and the F18 could only be run with 4 time steps.

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
slight	344	44.7	352	29.0	325	29.3	347	44.4	364	43.3	340	43.3
moderate	152	30.1	166	30.2	149	30.0	156	44.9	174	44.8	160	43.5

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

Model	Transfer Function	Error Tolerance	Scalar				Reference Color				Approximate Color			
			% NR	% P	% RT	% CT	% NR	% P	% RT	% CT	% NR	% P	% RT	% CT
Small Delta Wing	sparse	zero	0	0	0	100	29	0	0	71	29	0	0	71
		slight	15	0	35	50	29	0	22	49	29	0	24	47
		moderate	29	0	51	20	29	0	50	21	29	0	49	20
	filled	zero	0	0	0	100	0	0	0	100	0	0	0	100
		slight	0	15	35	50	0	2	48	50	0	15	37	48
		moderate	0	29	51	20	0	2	77	21	0	29	50	21
Large Delta Wing	sparse	zero	0	0	0	100	66	0	0	34	66	0	0	34
	filled	zero	0	0	0	100	0	7	0	93	0	7	0	93
F18	sparse	zero	0	0	0	100	63	0	10	27	63	0	0	37

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

non-TSP-tree case.

Another result is that the behavior of the reference and approximate error metrics are quite similar. The numbers in the respective columns of the tables are within 10% except with the Delta Wing using the filled transfer function. In the non-zero error cases, the approximate color metrics finds more spatial coherence and less temporal coherence than the reference color metrics. The difference in computation speeds is dramatic, as shown in Table 2. The approximate error metrics can be calculated between approximately 300 to 1000 times faster, and run at interactive speeds for the data sets we considered. Allowing the error metrics to be recalculated at interactive rates is important because it must be done when the transfer function is changed, and changing the function 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.

The values in Tables 3 and 4 show that the different error tolerance allow the user to trade quality for speed. There is over a factor of three difference in performance between all the corresponding zero-error and moderate-error runs for the non-cached small Delta Wing using the sparse transfer function. There is little difference between the zero-error and moderate-error cached runs with this data set because there is little room for improvement: the zero-error cases run at over 20 and 30 Hz when using the color error metrics.

The use of polygon templates, where the generation of polygons is shared between between subvolumes, allows an improvement from 168 ms to 33 ms per frame when the textures are cached (see Table 6). There is little or no advantage in using the polygon templates when the textures have not been cached or with the filled transfer function since the polygons are a small fraction of the overall work. The slight artifacts due to the polygon mismatch at subvolume boundaries cannot be noticed in Figures 4 through 9.

Finally, for the TSP-tree color error metric cases, the use of texture caches allows improvements in performance by allowing the texture generation to be performed only the first time a time step is displayed with a given transfer function instead of every time the time step is displayed. The difference in performance ranges between about one and two orders of magnitude. This is a useful result since it is common for users to enable animation in time.

7 Conclusions and Future Work

We have presented a fast volume rendering algorithm using 3D texture hardware for visualizing large-scale time-varying datasets. Utilizing a time-varying 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 to represent regions that have high variation, both in spatial and temporal domains. We exploit the property that there is

Transfer Function	With Templates		Without Templates	
	NC	C	NC	C
Sparse	2.4	0.168	2.4	0.033
Filled	16.8	4.05	21.8	4.9

Table 6: Average rendering speed for the large Delta Wing with and without polygon templates, in seconds. These runs used the approximate color error metrics with zero error tolerance. Key: NC = non-cached, C = cached.

only a limited number of subvolumes with different sizes in the hierarchical data structure by using polygon templates, which 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 to evaluate error metrics that do not use α-weighting.

An additional area of future work would be to implement the incremental scan conversion algorithm described by Yagel *et al.* [9]. This faster polygon generation algorithm may allow us to avoid using polygon templates without slowing down the implementation. This is possible because the current implementation's performance is limited by the graphics subsystem, which means that some additional computation would not reduce the performance. This would remove the artifacts between the subvolume boundaries. A third area of future work would be to explore the advantages of the color-based error metrics with a ray casting volume renderer.

Acknowledgments

This work was supported in part by NASA contract NAS2-14303. We would like to thank Neal Chaderjian, Scott Murman and Ken Gee for providing the datasets. We also thank 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] Silicon Graphics, Inc., Mountain View, CA. *OpenGL Volumizer Programmer's Guide*, 1998. Document Number 007-3720-001.

[14] 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.

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

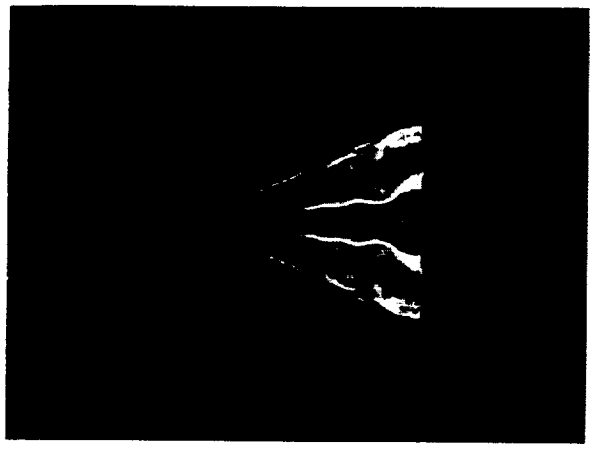

Figure 4: Small Delta Wing with no error and the sparse transfer function.

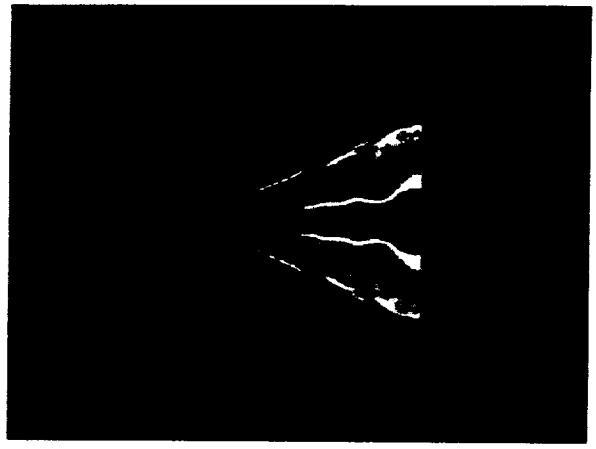

Figure 5: Small Delta Wing with slight error and the sparse transfer function.

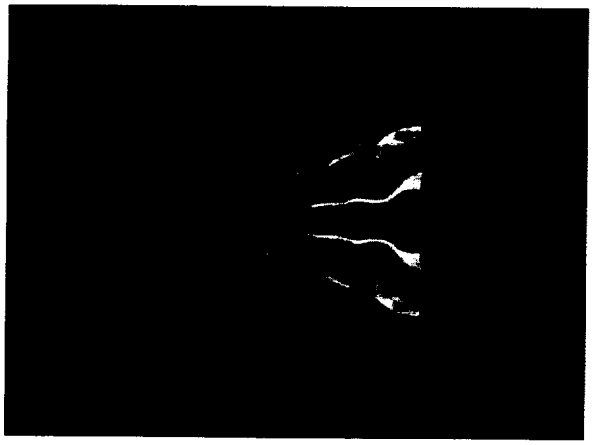

Figure 6: Small Delta Wing with moderate error and the sparse transfer function.

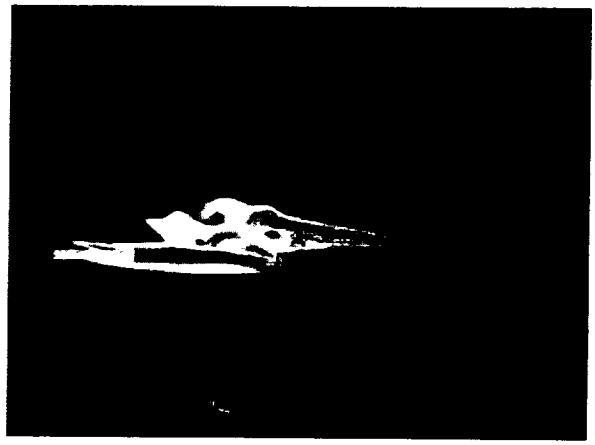

Figure 7: F18 with no error and the sparse transfer function.

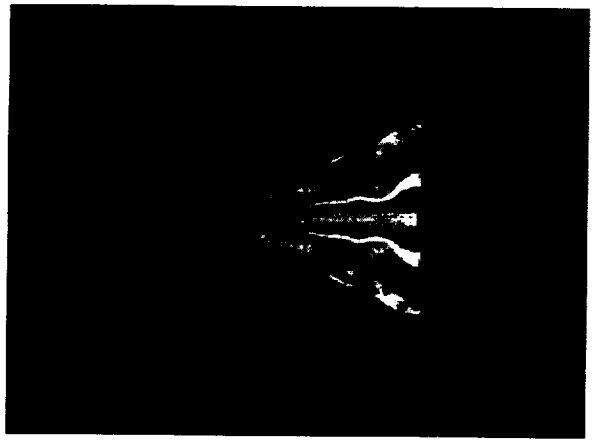

Figure 8: Delta Wing with no error and the filled transfer function.

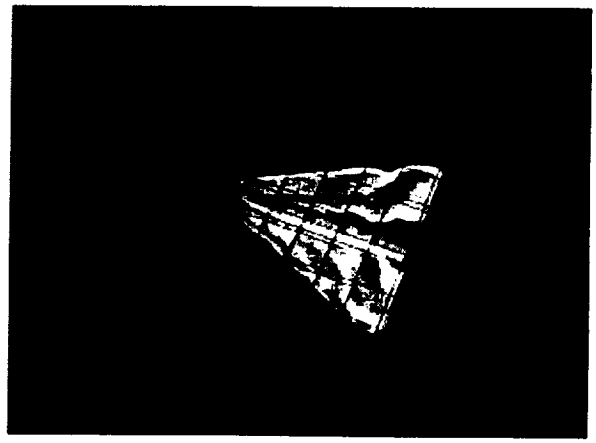

Figure 9: Large Delta Wing with sparse transfer function and lines showing the subvolume boundaries. Large subvolumes have high spatial coherence.