Efficient Ray Tracing of Volume Data

Marc Levoy

University of North Carolina

Abstract

Volume rendering is a family of techniques for visualizing sampled scalar or vector fields of three spatial dimensions without fitting geometric primitives to the data. A subset of these techniques generate images by computing 2D projections of a colored semi-transparent volume, where the color and opacity at each point is derived from the data using local operators. Since all voxels participate in the generation of each image, rendering time grows linearly with the size of the dataset. This paper presents a front-to-back image-order volume rendering algorithm and discusses two techniques for improving its performance. The first technique employs a pyramid of binary volumes to encode spatial coherence present in the data, and the second technique uses an opacity threshold to adaptively terminate ray tracing. Although the actual time saved depends on the data, speedups of an order of magnitude have been observed for datasets of useful size and complexity. Examples from two applications are given: medical imaging and molecular graphics.

CR categories and subject descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation - *display algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling - *curve, surface, solid, and object representations*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - *visible line/surface algorithms*; E.1 [**Data structures**]: Trees

General terms: Algorithms, Performance

Additional Key Words and Phrases: Scientific visualization, medical imaging, molecular graphics, volume visualization, volume rendering, ray tracing, hierarchical spatial enumeration, voxel, octree

1. Introduction

The increasing availability of powerful graphics workstations in the scientific and computing communities has catalyzed the development of new methods for visualizing discrete multidimensional data. In this paper, we address the problem of visualizing sampled scalar or vector fields of three spatial dimensions, henceforth referred to as *volume data.* We direct our attention to a family of visualization methods called *volume rendering techniques* in which the sample array is displayed directly, i.e. without first fitting geometric primitives to it. We further focus on a subset of these volume rendering techniques in which a color and an opacity is assigned to each voxel, and a 2D projection of the resulting colored semi-transparent volume is computed [9, 21, 29, 33]. The principal advantages of these techniques over other visualization methods are their superior image quality and the ability to generate images without explicitly defining surface geometry. The principal drawback of these techniques is their cost. Since all voxels participate in the generation of each image, rendering time grows linearly with the size of the dataset.

This research was supported by ONR grant N00014-86-K-0680, NIH Division of Research Resources grant RR02170-05, NCI grant P01-CA47982, and IBM.

Author's address: Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599.

This paper presents a front-to-back image-order volume rendering algorithm and discusses two strategies for improving its performance. The first optimization is based on the observation that many datasets contain coherent regions of empty voxels. In the context of volume rendering, a voxel is defined as empty if its opacity is zero. Methods for encoding coherence in volume data include octree hierarchical spatial enumeration [25], polygonal representation of bounding surfaces [10], and octree representation of bounding surfaces [13]. The algorithm presented in this chapter employs an octree enumeration similar to that of [25], but represents the enumeration by a pyramid of binary volumes or *complete octree* [35] rather than by a condensed representation. The present algorithm also differs from [25] in that it renders data in image order, i.e. by tracing viewing rays from an observer position through the octree, while [25] renders in object order, i.e. by traversing the octree in depth-first manner while following a consistent direction through space.

The second optimization is based on the observation that once a ray has struck an opaque object or has progressed a sufficient distance through a semi-transparent object, opacity accumulates to a level where the color of the ray stabilizes and ray tracing can be terminated. The idea of adaptively terminating ray tracing was first proposed in [34]. Many algorithms for displaying medical data stop after encountering the first surface or the first opaque voxel. In this guise, the idea has been reported in [15, 30, 32] and perhaps elsewhere. In the present algorithm, surfaces are not explicitly detected. Instead, they appear in the image as a natural byproduct of the stepwise accumulation of color and opacity along each ray. Adaptive termination is implemented by stopping each ray when its opacity reaches a user-selected threshold level.

The reduction in image generation time obtained by applying these optimizations is highly dependent on the depth complexity of the scene. In this paper, we focus on visualizations consisting of opaque or semi-transparent surfaces. A plot of opacity along a line perpendicular to one of these surfaces typically exhibits a bump shape several voxels wide, and voxels not in the vicinity of surfaces have an opacity of zero. For these scenes, savings of up to an order of magnitude over brute-force rendering algorithms have been observed. For scenes consisting solely of opaque surfaces, the cost of generating images has been observed to grow nearly linearly with the size of the image rather than linearly with the size of the dataset.

2. Brute-force algorithm

Let us first consider the brute-force volume rendering algorithm described in [21] and outlined in figure 1. We begin with a 3D array of data samples. For simplicity, let us assume a scalar-valued array forming a cube N voxels on a side. In this paper, we treat voxels as point samples of a continuous function rather than as volumes of homogeneous value. Voxels are indexed by a vector $\mathbf{i} = (i, j, k)$ where $i, j, k = 1, \ldots, N$, and the value of voxel $\mathbf{i}$ is denoted $f(\mathbf{i})$. Using local operators, a scalar or vector color $C(\mathbf{i})$ and an opacity $\alpha(\mathbf{i})$ is derived for each voxel.

Parallel rays are then traced into the data from an observer position as shown in figure 2. Let us assume that the image is a square measuring P pixels on a side, and that one ray is cast per pixel. Pixels and hence rays are indexed by a vector $\mathbf{u} = (u, v)$ where $u, v = 1, \ldots, P$. For each ray, a vector of colors and opacities is computed by resampling the data at W evenly spaced locations along the ray and trilinearly interpolating from the colors and opacities in the eight voxels surrounding each sample location. Samples are indexed by a vector $\mathbf{U} = (u, v, w)$ where (u, v) identifies the ray, and $w = 1, \ldots, W$ corresponds to distance along the ray with $w = 1$ being closest to the eye. The color and opacity of sample $\mathbf{U}$ are denoted $C(\mathbf{U})$ and $\alpha(\mathbf{U})$ respectively. Finally, a fully opaque background is draped behind the dataset, and the resampled colors and opacities are composited with each other and with the background to yield a color for the ray. This color is denoted $C(\mathbf{u})$.

The rendering algorithms in [9] and [21] process data from back to front, while the algorithms in [29] and [33] operate from front to back. In this paper, we work from front to back, compositing the color and opacity at each sample location *under* the ray in the sense of [27]. Specifically, the color $C_{out}(\mathbf{u};\mathbf{U})$ and opacity $\alpha_{out}(\mathbf{u};\mathbf{U})$ of ray $\mathbf{u}$ after processing sample $\mathbf{U}$ is related to the color $C_{in}(\mathbf{u};\mathbf{U})$ and opacity $\alpha_{in}(\mathbf{u};\mathbf{U})$ of the ray before processing the sample and the color $C(\mathbf{U})$ and opacity $\alpha(\mathbf{U})$ of the sample by the transparency formula

$$\hat{C}_{out}(\mathbf{u};\mathbf{U}) = \hat{C}_{in}(\mathbf{u};\mathbf{U}) + \hat{C}(\mathbf{U})(1 - \alpha_{in}(\mathbf{u};\mathbf{U})) \tag{1a}$$

and

$$\alpha_{out}(\mathbf{u};\mathbf{U}) = \alpha_{in}(\mathbf{u};\mathbf{U}) + \alpha(\mathbf{U})(1 - \alpha_{in}(\mathbf{u};\mathbf{U})). \tag{1b}$$

where $\hat{C}_{in}(\mathbf{u};\mathbf{U}) = C_{in}(\mathbf{u};\mathbf{U})\alpha_{in}(\mathbf{u};\mathbf{U})$, $\hat{C}_{out}(\mathbf{u};\mathbf{U}) = C_{out}(\mathbf{u};\mathbf{U})\alpha_{out}(\mathbf{u};\mathbf{U})$, and $\hat{C}(\mathbf{U}) = C(\mathbf{U})\alpha(\mathbf{U})$.

After all samples along a ray have been processed, the color $C(\mathbf{u})$ of the ray is obtained from the expression $C(\mathbf{u}) = \hat{C}_{out}(\mathbf{u};\mathbf{W}) / \alpha_{out}(\mathbf{u};\mathbf{W})$ where $\mathbf{W} = (u,v,W)$. If a fully opaque background is draped behind the dataset at $w' = W + 1$ and composited under the ray after it has passed through the data, then $\alpha_{out}(\mathbf{u};\mathbf{W}') = 1$ where $\mathbf{W}' = (u,v,w')$, and this normalization step can be omitted.

The ray tracing, resampling, and compositing steps of the brute-force rendering algorithm are thus summarized as follows:

procedure $TraceRay_1(\mathbf{u})$ **begin**
- $\hat{C}(\mathbf{u}) := 0;$
- $\alpha(\mathbf{u}) := 0;$
- $\mathbf{x}_1 := First(\mathbf{u});$
- $\mathbf{x}_2 := Last(\mathbf{u});$
- $\mathbf{U}_1 := \lfloor Image(\mathbf{x}_1) \rceil;$
- $\mathbf{U}_2 := \lfloor Image(\mathbf{x}_2) \rfloor;$
- {Loop through all samples falling within data}
- **for** $\mathbf{U} := \mathbf{U}_1$ **to** $\mathbf{U}_2$ **do begin**
  - $\mathbf{x} := Object(\mathbf{U});$
  - {If sample opacity > 0,}
  - {then resample color and composite into ray}
  - $\alpha(\mathbf{U}) := Sample(\alpha,\mathbf{x});$
  - **if** $\alpha(\mathbf{U}) > 0$ **then begin**
    - $\hat{C}(\mathbf{U}) := Sample(\hat{C},\mathbf{x});$
    - $\hat{C}(\mathbf{u}) := \hat{C}(\mathbf{u}) + \hat{C}(\mathbf{U})(1 - \alpha(\mathbf{u}));$
    - $\alpha(\mathbf{u}) := \alpha(\mathbf{u}) + \alpha(\mathbf{U})(1 - \alpha(\mathbf{u}));$
  - **end**
- **end**

end $TraceRay_1$.

The *First* and *Last* procedures accept a ray index and return the object-space coordinates of the points where the ray enters and leaves the data respectively. These coordinates are denoted by real vectors of the form $\mathbf{x} = (x,y,z)$ where $1 \le x,y,z \le N$. The *Object* and *Image* procedures convert between object-space coordinates and image-space coordinates. Although these calculations normally require matrix multiplications, they can be simplified for the restricted case of an orthographic viewing projection by retaining the coordinates computed in the previous invocation and using differencing. The *Sample* procedure accepts a 3D array of colors or opacities and the object-space coordinates of a point, and returns an approximation to the color or opacity at that point by trilinearly interpolating from the eight surrounding voxels.

3. Optimized algorithm

The first optimization technique we consider is hierarchical spatial enumeration. For a dataset measuring N voxels on a side where $N = 2^M+1$ for some integer M, we represent this enumeration by a pyramid of $M+1$ binary volumes as shown in figure 3 for the case of $N = 5$. Volumes in this pyramid are indexed by a level number m where $m = 0, \ldots, M$, and the volume at level m is denoted V_m. Volume V_0 measures $N-1$ cells on a side, volume V_1 measures $(N-1)/2$ cells on a side, and so on up to volume V_m, which is a single cell. Cells are indexed by a level number m and a vector $\mathbf{i} = (i,j,k)$ where $i,j,k = 1, \ldots, N-1$, and the value contained in cell $\mathbf{i}$ on level m is denoted $V_m(\mathbf{i})$. We define the size of cells on level m to be 2^m times the spacing between voxels. Since voxels are treated as points, whereas cells fill the space between voxels, each volume is one cell smaller in each direction than the underlying dataset as shown in the figure. We also place voxel $(1,1,1)$ at the front-lower-right corner of cell $(1,1,1)$. Thus, for example, cell $(1,1,1)$ on level zero encloses the space between voxels $(1,1,1)$ and $(2,2,2)$.

We construct the pyramid as follows. Cell $\mathbf{i}$ in the base volume V_0 contains a zero if all eight voxels lying at its vertices have opacity equal to zero. Cell $\mathbf{i}$ in any volume V_m, $m > 0$, contains a zero if all eight cells on level $m-1$ that form its octants contain zeros. In other words, let $\{1,2,...,k\}^n$ be the set of all n-vectors with entries $\{1,2,...,k\}$. In particular, $\{1,2,...,k\}^3$ is the set of all vectors in 3-space with integer entries between 1 and k. We then define

$$V_0(\mathbf{i}) = \begin{cases} 1 & \text{if } \alpha(\mathbf{i}+\Delta\mathbf{i}) = 1 \text{ for } \mathbf{i} \in \{1,2,...,N-1\}^3 \text{ and any } \Delta\mathbf{i} \in \{0,1\}^3 \\ 0 & \text{otherwise} \end{cases} \tag{2a}$$

and

$$V_m(\mathbf{i}) = \begin{cases} 1 & \text{if } V_{m-1}(2\mathbf{i}-\Delta\mathbf{i}) = 1 \text{ for } \mathbf{i} \in \{1,2,...,(N-1)/(m+1)\}^3 \text{ and any } \Delta\mathbf{i} \in \{0,1\}^3 \\ 0 & \text{otherwise} \end{cases} \tag{2b}$$

for $m = 1, \ldots, M$.

We now reformulate the ray tracing, resampling, and compositing steps of our rendering algorithm to use this pyramidal data structure. For each ray, we first compute the point where the ray enters the single cell at the top level. We then traverse the pyramid in the following manner. When we enter a cell, we test its value. If it contains a zero, we advance along the ray to the next cell on the same level. If the parent of the new cell differs from the parent of the old cell, we move up to the parent of the new cell. We do this because if the parent of the new cell is unoccupied, we can advance the ray further on our next iteration than if we had remained on a lower level. This ability to advance quickly across empty regions of space is where the algorithm saves its time. If, however, the cell being tested contains a one, we move down one level, entering whichever cell encloses our current location. If we are already at the lowest level, we know that one or more of the eight voxels lying at the vertices of the cell have opacity greater than zero. We then draw samples at evenly spaced locations along that portion of the ray falling within the cell, resample the data at these sample locations, and composite the resulting color and opacity into the color and opacity of the ray.

The second optimization technique we consider is adaptive termination of ray tracing. Our goal is to quickly identify the last sample location along a ray that significantly changes the color of the ray. Returning to equation (1a), we define a significant color change as one in which $C_{out}(\mathbf{u};\mathbf{U}) - C_{in}(\mathbf{u};\mathbf{U}) > \varepsilon$ for some small $\varepsilon > 0$. Since $\alpha_{in}(\mathbf{u};\mathbf{U})$ in equation (1b) increases monotonically along the ray, no significant color changes occur beyond the point where $\alpha_{out}(\mathbf{u};\mathbf{U})$ first exceeds $1-\varepsilon$. This becomes our termination criterion. Higher values of ε reduce rendering time, while lower values reduce image artifacts. For the datasets used in this paper, $\varepsilon = .05$ usually represents a satisfactory compromise.

Combining both of these optimizations gives us the following algorithm:

procedure *TraceRay*$_2$($\mathbf{u}$) **begin**

$\hat{C}(\mathbf{u}) := 0;$

$\alpha(\mathbf{u}) := 0;$

$\mathbf{x} := First(\mathbf{u});$

$m := m_{max};$

{Loop until beyond data or opacity > threshold}
while $InBounds(\mathbf{x})$ **and** $\alpha(\mathbf{u}) \leq 1 - \varepsilon$ **do begin**

$\mathbf{i} := Index(m, \mathbf{x});$

{If high level cell contains a one, drop a level}
if $V_m(\mathbf{i})$ **and** $m > m_{\min}$ **then** $m := m - 1;$

else begin

{If level zero cell contains a one, render it}
if $V_m(\mathbf{i})$ **then** $RenderCell(\mathbf{u}, \mathbf{x}, Next(m, \mathbf{x}, \mathbf{u}));$

{Advance to next cell and maybe jump to higher level}
while $Parent(m, Index(m, Next(m, \mathbf{x}, \mathbf{u}))) \neq Parent(m, \mathbf{i})$ **and** $m < M$ **do begin**

$\mathbf{i} := Parent(m, \mathbf{i});$

$m := m + 1;$

end

$\mathbf{x} := Next(m, \mathbf{x}, \mathbf{u});$

end

end

end *TraceRay*$_2$

procedure $RenderCell(\mathbf{u}, \mathbf{x}_1, \mathbf{x}_2)$ **begin**

$\mathbf{U}_1 := \lceil Image(\mathbf{x}_1) \rceil;$

$\mathbf{U}_2 := \lfloor Image(\mathbf{x}_2) \rfloor;$

{Loop through all samples falling within cell}
for $\mathbf{U} := \mathbf{U}_1$ **to** $\mathbf{U}_2$ **do begin**

$\mathbf{x} := Object(\mathbf{U});$

{If any of eight surrounding voxels have opacity > 0,}
{then resample color and opacity and composite into ray}
if $V_0(Index(0, \mathbf{x}))$ **then begin**

$\hat{C}(\mathbf{U}) := Sample(\hat{C}, \mathbf{x});$

$\alpha(\mathbf{U}) := Sample(\alpha, \mathbf{x});$

$\hat{C}(\mathbf{u}) := \hat{C}(\mathbf{u}) + \hat{C}(\mathbf{U})(1 - \alpha(\mathbf{u}));$

$\alpha(\mathbf{u}) := \alpha(\mathbf{u}) + \alpha(\mathbf{U})(1 - \alpha(\mathbf{u}));$

end

end

end *RenderCell*.

The *Index* procedure accepts a level number and the object-space coordinates of a point, and returns the index of the cell that contains it. The *Parent* procedure accepts a level number and cell index, and returns the index of the parent cell. The *Next* procedure accepts a level number and a point on a ray, and computes using a method similar to [2] the coordinates of the point where the ray enters the next cell on the same level. The *RenderCell* procedure composites the contribution made to a ray by the specified interval of volume data. The algorithm terminates when the ray leaves the data as detected by the *InBounds* procedure.

Figure 4 shows in two dimensions how a typical ray might traverse a hierarchical enumeration. The level zero cell corresponding to each non-empty voxel is denoted by a shaded box. The largest empty cell enclosing each empty voxel is denoted by an unshaded box. The sequence of points computed by the *Next* procedure are denoted by circular dots. In regions containing many non-empty level zero cells, the spacing between these dots is close to the spacing between voxels. We are therefore led to ask the question: why not simply resample the data at these points? We observe, however, that these points are not evenly spaced along the ray. If the data is resampled at such non-uniformly spaced points, a noise component may be added to the resulting image [7]. To avoid these artifacts, we superimpose a set of evenly spaced sample locations as shown by the rectangular tick marks in the figure, then limit ourselves to resampling the data at these locations.

Assuming that we are rendering a non-empty dataset, most cells on the top levels of the pyramid will contain ones. It is therefore inefficient to begin our traversal there. For the datasets used in this paper, traversal costs were minimized by setting $m_{\max} = M - 2$ for all values of M. Assuming an orthographic viewing projection, the cost of advancing a ray from one cell to the next by computing ray-cell intersections is higher than the cost of advancing the ray from one sample location to the next using differencing. It is therefore inefficient to descend to level zero. Instead, we descend to some higher level, loop through the sample locations falling within that cell, and render those for which $V_0(\mathit{Index}\,(0,\mathbf{x})) = 1$. For the current implementation, $m_{\min} = 2$ yields the best results.

The memory required for the optimized algorithm is $2N^3$ bytes to hold a monochrome color and opacity for each voxel, $(8^{M+1}-1)/7$ bits to hold the pyramid of binary volumes, and P^2 bytes to hold a monochrome output image. Condensed representations of the pyramid such as linear octrees [12] are possible, although the amount of memory saved would be small compared to the size of the color and opacity arrays, and the cost of accessing a cell would generally be higher.

4. Comparison to ray tracing of geometrically defined scenes

The tracing of rays through coherent regions of empty voxels in volume data is analogous to the tracing of rays through expanses of empty space in geometrically defined scenes. This problem has received much attention in the computer graphics literature (see [4] for an excellent survey), and it is useful to compare the present algorithm to strategies for speeding up ray tracing of geometric scenes.

One such strategy is to place bounding volumes around primitives or groups of primitives. Rays are tested first against these volumes, and if a volume is hit, then against its contents. Bounding volume schemes that have been tried include spheres [34], parallelepipeds [28], extruded extents [19], and convex hulls [20]. These methods can be applied to volume data by fitting geometric primitives to the sample array. Primitives that have been used for this purpose include opaque cubes [18], polygonal meshes constructed from 2D contours [10], and voxel-sized polygons generated directly from 3D data samples [24]. The principal drawback of this approach is that fitting of primitives requires making a binary classification of the data. As a result, these methods often exhibit false positives (spurious surfaces) or false negatives (erroneous holes in surfaces), particularly in the presence of small or poorly defined features.

An alternative strategy is to subdivide space into disjoint cells and to associate with each cell a list of primitives that fall wholely or partially inside it. Rays are advanced incrementally through the scene, moving from cell to cell. When a ray enter a cell that contains primitives, the ray is tested against those primitives; when a ray enters a cell marked as empty, the ray is simply advanced to the next cell. Variants of this approach include uniform subdivision of space into a regular 3D grid of cubic cells [11], adaptive subdivision into parallelepipeds, generalized cubes, or tetrahedra [8], and adaptive hierarchical subdivision into cubic cells of varying size using octrees [14]. These methods can be applied to a geometric description of the volume data by fitting primitives as described above, or they may be applied directly to the sample array. Specifically, if we treat each data sample as a cell, the resulting regular 3D grid of cells is analogous to uniform subdivision of a geometric scene. Similarly, octree representations of volume data are analogous to adaptive hierarchical spatial subdivisions of geometric scenes.

The analogy between spatial enumeration of volume data and spatial subdivision of a geometric scene is not exact, however, and comparisons made in the literature between competing schemes for subdividing geometric scenes do not scale well when applied to volume data. In particular, spatial subdivisions of geometric scenes typically consist of hundreds of cells each containing many primitives [6], whereas volume datasets consist of tens of millions of spatially ordered cells each containing a single data sample. Several researchers [2, 6, 11] have reported that, for the geometric scenes they have tested, uniform subdivision outperforms hierarchical subdivision. For the volume datasets considered in this paper, a hierarchical data structure is more efficient.

5. Implementation and results

To understand how the algorithm behaves on typical scenes, let us consider some examples. The characteristics of three datasets are given in table 1. The first is a computerized tomography (CT) study of a human skull mounted in a lucite head cast. To demonstrate the effect of semi-transparent surfaces on the performance of the algorithm, this dataset was rendered twice, once with a semi-transparent air-lucite boundary surface (figure 5), and once with a completely transparent boundary surface (figure 6). The second dataset is a portion of an electron density map of Staphylococcus Aureus ribonuclease. A volume rendering of an isovalue surface from this map is shown in figure 7. The polymer backbone crosses the image from bottom to top, and two Tyrosine residues with their characteristic six-atom benzene rings can be seen extending to the left and right sides of the backbone. A color-coded stick representation of the molecular structure has been superimposed on the image to aid in its interpretation. To study the growth of rendering cost with respect to dataset size, this dataset was rendered at three different spatial resolutions, the largest of which is shown in the figure. The last dataset is a CT study of a complete human head, a volume rendering of which is shown in figure 8.

The time required to calculate voxel opacities for a dataset is proportional to the number of voxels it contains. The time required to calculate voxel colors is proportional to the number of non-empty voxels (voxels whose opacity is non-zero). For the 256 x 128 x 113 voxel jaw with semi-transparent skin (figure 5), these two steps took 45 seconds and 15 seconds respectively on a Sun 4/280. The time required to construct a pyramid of binary volumes is proportional to the number of cells the pyramid contains. The computation of each cell requires accessing eight spatially adjacent locations in a 4D array and performing seven logical "or" operations. For figure 5, this step took about 30 seconds. Since the pyramid depends on the array of opacities rather than on the original data, it must be recomputed whenever these opacities change. The pyramid is independent of observer position, however, and can be used to efficiently generate multiple views from a single set of opacities.

The combined costs of ray tracing, resampling, and compositing for all three datasets are summarized in table 2. Separate entries are provided for the brute-force algorithm, the optimized algorithm with adaptive termination of ray tracing disabled by setting $\varepsilon = 0$, and the fully optimized algorithm with $\varepsilon = .05$. As the table shows, hierarchical enumeration reduced rendering time by a factor of between 2.0 and 5.0 for this data, and adaptive termination of ray tracing added another factor of between 1.3 and 2.2. We also observe that adding a semi-transparent surface to the rendering of the skull fragment decreased the amount of time saved but did not eliminate the savings completely. We finally note that doubling the spatial resolution of the electron density map increased rendering time by roughly a factor of eight for the brute-force algorithm and five for the optimized algorithm.

To help us interpret these results, the cost of generating figure 8 has been broken down into its constituent parts. Using the brute-force rendering algorithm described in section 2, the cost of finding all non-empty samples along a ray is proportional to the length of the ray clipped to the boundaries of the dataset. For the observer position used in figure 8, a visualization of this cost is shown in figure 9a. Brighter pixels represent more work. The image is essentially an X-ray of a cube of uniform density. The cost of resampling and compositing the non-empty samples along a ray is proportional to the number found along the ray. For the dataset under consideration, a visualization of this cost is shown in figure 9b. This image is essentially an X-ray of a binary representation of the data. As expected, it is brightest along silhouettes where rays pass through large amounts of bony material. The total cost of rendering figure 8 using the brute-force algorithm is a weighted sum of figures 9a and 9b.

Using hierarchical enumeration, the cost of finding all non-empty samples along a ray is proportional to the number of iterations through the outer loop in the $TraceRay_2$ procedure plus the number of tests of level zero cells performed in the *RenderCell* procedure. A visualization of this cost is shown in figure 10a. This image is essentially an X-ray of an octree. The cost of resampling and compositing the non-empty samples is shown in figure 10b. Since the use of hierarchical enumeration alone does not reduce the number of samples composited, figure 10b is identical to figure 9b. The total cost of rendering figure 8 using hierarchical enumeration is a weighted sum of figures 10a and 10b.

Adaptive termination of ray tracing reduces the number of non-empty samples which must be found. For $\varepsilon = .05$, a visualization of the reduced cost is shown in figure 11a. In regions where fewer samples are processed, resampling and compositing costs drop as well, as shown in figure 11b. The total cost of rendering figure 8 using both optimization techniques is a weighted sum of figures 11a and 11b.

6. Conclusions

An algorithm for efficiently visualizing sampled scalar or vector fields of three spatial dimensions has been described. The algorithm employs both hierarchical spatial enumeration and adaptive termination of ray tracing to reduce rendering costs. Any opacity assignment operator that partitions a volume dataset into coherent regions of opaque and transparent voxels is a candidate for this algorithm. Although the amount of time saved depends on the depth complexity of the partitioned scene, savings of more than an order of magnitude have been observed for many datasets.

If there is coherence present in a dataset, there may also be coherence present in its projections. This is particularly true for data acquired from sensing devices, where the acquisition process often introduces considerable blurring. We can take advantage of this coherence by casting a sparse grid of rays, less than one per pixel, and adaptively increasing the number of rays in regions of high image complexity. Images may be formed from the resulting non-uniform array of sample colors by interpolation and resampling at the display resolution. In many cases, this optimization reduces rendering time by another order of magnitude [23]. Alternatively, a sequence of successively more refined images can be generated at equally spaced intervals of time by casting more rays, adding the resulting colors to the sample array, and repeating the interpolation and resampling steps.

A strategy used to speed up ray tracing of geometrically defined scenes that has not been addressed in this paper is to group together rays emanating from similar locations and traveling in similar directions. Specific techniques include the light buffer of [16], the ray coherence algorithm of [26], and the ray classification algorithm of [3]. In the present algorithm, an orthographic viewing projection is used, and shadowing, reflection, and refraction are not supported. All rays consequently travel in the same direction. Many volume rendering systems offer a perspective viewing projection, however, and the author has developed algorithms for casting shadows through volume data [22]. Directional data structures might be useful in these cases. Other ray tracing techniques that might be applicable to volume rendering include generalized rays such as beams [17], cones [1], and pencils [31], statistical optimizations such as distributed ray tracing [7], and frame-to-frame coherence [5].

Acknowledgements

The author wishes to thank Profs. Henry Fuchs, Stephen M. Pizer, Frederick P. Brooks Jr., and Turner Whitted of the Computer Science Department and Drs. Julian Rosenman and Edward L. Chaney of the Radiation Oncology Department for their advice and support. Thanks are also due to John Gauch for many enlightening discussions, and to the anonymous reviewers for their useful suggestions. The CT scans used in this paper were provided by the Radiation Oncology Department at North Carolina Memorial Hospital. The electron density map was provided by Dr. Chris Hill of the University of York Chemistry Department, and was reformatted and brought on-line with the help of Mark Harris of the University of North Carolina.

References

[1] Amantides, J. Ray tracing with cones. *Computer Graphics* 18, 3 (July, 1984), 129-135.

[2] Amantides, J. and Woo, A. A fast voxel traversal algorithm for ray tracing. *Proc. Eurographics '87,* G. Marechal, Ed. Elsevier Science Publishers B.V., North-Holland, 1987, pp. 3-10.

[3] Arvo, J. and Kirk, D. Fast ray tracing by ray classification. *Computer Graphics* 21, 4 (July, 1987), 55-64.

[4] Arvo, J. and Kirk, D. A survey of ray tracing acceleration techniques. *SIGGRAPH '88 course notes* (Atlanta, GA, August, 1988), vol. 7.

[5] Badt, S.,Jr. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer* 4 (1988), 123-132.

[6] Cleary, J.G., and Wyvill, G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer* 4 (1988), 65-83.

[7] Cook, R.L. Stochastic sampling in computer graphics. *ACM Transactions on Graphics* 5, 1 (January, 1986), 51-72.

[8] Dippe, M. and Swensen, J. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics* 18, 3 (July, 1984), 149-158.

[9] Drebin, R.A., Carpenter, L., and Hanrahan, P. Volume rendering. *Computer Graphics* 22, 4 (August, 1988), 65-74.

[10] Fuchs, H., Kedem, Z.M., and Uselton, S.P. Optimal surface reconstruction from planar contours. *Communications of the ACM* 20, 10 (1977), 693-702.

[11] Fujimoto, A., Tanaka, T., and Iwata, K. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications* 6, 4 (April, 1986), 16-26.

[12] Gargantini, I. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing* 20 (1982), 365-374.

[13] Gargantini, I., Walsh, T.R.S., and Wu, O.L. Displaying a voxel-based object via linear octtrees. *Proc. SPIE* 626 (1986), 460-466.

[14] Glassner, A.S. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications* 4, 10 (October, 1984), 15-22.

[15] Goldwasser, S. Rapid techniques for the display and manipulation of 3-D biomedical data. *NCGA ’86 tutorial* (Anaheim, CA, May, 1986).

[16] Haines, E.A. and Greenberg, D.P. The light buffer: a shadow-testing accelerator. *IEEE Computer Graphics and Applications* 6, 9 (September, 1986), 6-16.

[17] Heckbert, P.S. and Hanrahan, P. Beam tracing polygonal objects. *Computer Graphics* 18, 3 (July, 1984), 119-127.

[18] Herman, G.T. and Liu, H.K. Three-dimensional display of human organs from computer tomograms. *Computer Graphics and Image Processing* 9 (1979), 1-21.

[19] Kajiya, J.T. New techniques for ray tracing procedurally defined objects. *Computer Graphics* 17, 3 (July, 1983), 91-102.

[20] Kay, T.L. and Kajiya, J.T. Ray tracing complex scenes. *Computer Graphics* 20, 4 (August, 1986), 269-278.

[21] Levoy, M. Display of surfaces from volume data. *IEEE Computer Graphics and Applications* 8, 3 (May, 1988), 29-37.

[22] Levoy, M. *Display of Surfaces from Volume Data.* Ph.D. dissertation, University of North Carolina at Chapel Hill, May, 1989.

[23] Levoy, M. Volume rendering by adaptive refinement. *The Visual Computer* 6 (1990). In press.

[24] Lorensen, W.E. and Cline, H.E. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics* 21, 4 (July, 1987), 163-169.

[25] Meagher, D. Geometric modeling using octree encoding. *Computer Graphics and Image Processing* 19 (1982), 129-147.

[26] Ohta, M. and Mamouru, M. Ray coherence theorem and constant time ray tracing algorithm. *Proc. CG International ’87,* T.L. Kunii, Ed. 1987, pp. 303-314.

[27] Porter, T. and Duff, T. Compositing digital images. *Computer Graphics* 18, 3 (July, 1984), 253-259.

[28] Rubin, S.M. and Whitted, T. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics* 14, 3 (July 1980), 110-116.

[29] Sabella, P. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics* 22, 4 (August, 1988), 51-58.

[30] Schlusselberg, D.S. and Smith, W.K. Three-dimensional display of medical image volumes. *Proc. NCGA ’86* (Anaheim, CA, May, 1986), vol. III, pp. 114-123.

[31] Shinya, M., Takahashi, T., and Naito, S. Principles and applications of pencil tracing. *Computer Graphics* 21, 4 (July, 1987), 45-54.

[32] Trousset, Y. and Schmitt, F. Active-ray tracing for 3D medical imaging. *Proc. Eurographics '87*, G. Marechal, Ed. Elsevier Science Publishers B.V., North-Holland, 1987, pp. 139-149.

[33] Upson, C. and Keeler, M. VBUFFER: visible volume rendering. *Computer Graphics* 22, 4 (August, 1988), 59-64.

[34] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM* 23, 6 (June, 1980), 343-349.

[35] Yau, M. and Srihari, S.N. A hierarchical data structure for multidimensional digital images. *Communications of the ACM* 26, 7 (July, 1983), 504-515.

Submitted June, 1988; revised February, 1989; accepted April, 1989