Interactive Volume Navigation

Martin L. Brady, *Member, IEEE*, Kenneth K. Jung, *Member, IEEE*,
H.T. Nguyen, *Member, IEEE*, and Thinh PQ Nguyen *Member, IEEE*

Abstract—Volume navigation is the interactive exploration of volume data sets by "flying" the viewpoint through the data, producing a volume rendered view at each frame. We present an inexpensive perspective volume navigation method designed to be run on a PC platform with accelerated 3D graphics hardware. The heart of the method is a two-phase perspective raycasting algorithm that takes advantage of the coherence inherent in adjacent frames during navigation. The algorithm generates a sequence of approximate volume-rendered views in a fraction of the time that would be required to compute them individually. The algorithm handles arbitrarily large volumes by dynamically swapping data within the current view frustum into main memory as the viewpoint moves through the volume. We also describe an interactive volume navigation application based on this algorithm. The application renders gray-scale, RGB, and labeled RGB volumes by volumetric compositing, allows trilinear interpolation of sample points, and implements progressive refinement during pauses in user input.

Index Terms—Volume navigation, volume rendering, 3D medical imaging, scientific visualization, texture mapping.

---

1 Introduction

Volume-rendering techniques can be used to create informative two-dimensional (2D) rendered views from large three-dimensional (3D) images, or *volumes*, such as those arising in scientific and medical applications. In a typical volume-rendering scenario, rays are cast from an observation point outside the volume through the entire volume to obtain a 2D view of the whole data set. In dealing with large 3D data sets, this approach has several limitations. First, it can be difficult to discern small, complicated internal structures within a large data set when generating an image from the entire volume. Volumetric compositing techniques can be used to display some of the internal data via translucency effects, and cutting planes can aid in removing some of the occlusions. However, it can still be difficult to locate and isolate internal structures using these techniques. Second, an animated sequence of views sometimes reveals more information than a set of static images, even if the quality of the individual frames is reduced. However, interactive update of the point of view is often precluded for large volumes due to the prohibitively long time needed to render a single image. *Volume navigation* [2], [3] addresses these limitations by placing a viewing frustum of limited depth inside the volume data set. The volume acts as a virtual environment in which the user can navigate (translate and rotate the point of view). Navigating through the volume allows more flexibility in avoiding occluding objects and focusing on structures of interest. Limiting the field of view also reduces the computational burden in rendering an individual frame.

The methods we consider are best suited for exploring large, complex, optically dense volumes and searching for relatively small features of interest. The data set should contain regions of low opacity through which to navigate the viewpoint. In order to convey a sense of smooth motion, the views should be produced at interactive rates. The complete data set may exceed the system's memory, but the amount of data enclosed by the view frustum is assumed to fit within main memory. The goal is to quickly browse the volume looking for areas of interest using interactive 3D movements. We also assume that some reduction in the quality of individual frames can be tolerated when moving at this high rate. During any pause in motion, the view should be progressively improved, so that once a possible area of interest is located, a high-resolution view from the current viewpoint is obtained.

An application for this technique arises in the analysis of 3D medical data. Volume navigation can be used to simulate an endoscopic examination of structures such as bronchial passages, blood vessels, or the intestinal tract using 3D radiological images [8], [16], [19], [21], [10]. The "hollow" areas of anatomy through which a catheter would normally move serve as the regions of low opacity required for navigation. This digital simulation can be more flexible than the physical procedure, allowing one to traverse complex branching structures, pass through solid objects, or render obscuring objects transparent.

In this paper, we describe an inexpensive volume navigation method designed for a standard PC platform with accelerated 3D graphics hardware. The computation is accelerated by taking advantage of the fact that views are along a continuous path, and adjacent viewing sites differ very little. Much of the work done to produce a view at a given site can be stored and reused to produce approximate views at nearby locations, thus amortizing the cost to render a view over many frames. A raycasting algorithm is presented that exploits this strategy to generate a sequence of approximate views in far less time than would be required to compute them individually. Furthermore, in order to avoid delays for loading voxel data from disk, a *subcube* of data enclosing the view frustum is maintained in main memory and updated incrementally as the frustum moves.

• *The authors are with the Microcomputer Research Labs, Intel Corporation. E-mail: Martin.Brady@intel.com.*

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 107050.

1077-2626/98/$10.00 © 1998 IEEE

An overview of the volume navigation problem and our approach is given in Section 2. The two-phase algorithm for perspective volume rendering is presented in Section 3 and applied to volume navigation in Section 4. The incremental data I/O algorithm that completes the core of our interactive navigation strategy is described in Section 5. In Section 6, we discuss previous work in accelerating perspective raycasting and relate it to our algorithms. Section 7 describes the implementation and analysis of a Windows® NT-based volume navigation application using our algorithms, followed by some concluding observations in Section 8. A preliminary version of this work that primarily described the two-phase perspective raycasting algorithm appeared in [4].

2 Volume Navigation

A volumetric data set consists of an $N_x \times N_y \times N_z$ three-dimensional array of voxels. The sampling intervals along the three coordinate axes are σ_x, σ_y, σ_z, respectively. We assume that the voxels are isotropic, i.e., $\sigma_x = \sigma_y = \sigma_z = \sigma$ in this paper. Distances are written in units of σ, the voxel spacing. This assumption simplifies some of the details of the algorithms. However, the algorithms extend naturally to anisotropic data.

The view frustum is much smaller than the volume, and the viewpoint is typically placed within the volume at an arbitrary position, p, and orientation, v. During navigation, an arbitrary path is traced through the volume as a sequence of volume rendered views at a discrete sequence of viewing sites, (p_i, v_i), $i = 1, 2, 3, \ldots$ whose values are not known in advance. The depth of view is fixed, i.e., rays are cast to a distance d from the viewpoint. In a highly transparent region, this may cause the rays to clip before they have saturated. In order to see a structure that is further than d from the viewpoint, the user must move toward it.

In general, a navigation trajectory will tend to string together a sequence of incremental steps in a given direction or incremental rotations about a given axis. This coherence between nearby frames can be used to accelerate their computation. Brady, et al. [3] proposed an algorithm for volume navigation in which a dense set of short parallel ray segments are rendered, and are then used to construct complete rays from many neighboring points of view. Intermediate results from the computation of a given view are used to quickly compute f successive frames, after which a fresh set of intermediate results must be produced. By amortizing this work over the f frames generated, approximately a factor f speedup is obtained in the core-rendering portion of the computation. The method is restricted to parallel-projected volume rendering, however. Note that parallel projection is commonly used for rendering "external" views of volume data and is often sufficient in that case, since the distance from the viewer to the data is relatively large. This assumption is violated in the extreme in the case of volume navigation. Although useful information can certainly be obtained from parallel projections in this scenario, perspective projection is required if the views are to look natural, since most of the information is near the viewer. We extend the general idea of computing multiple views from stored volume-rendered ray segments to perspective geometry. This requires a significantly different approach since the previous method relies heavily on the fact that stored ray segments are parallel. The two-phase perspective raycasting algorithm is described in the following section.

3 Two-Phase Perspective Volumetric Raycasting Algorithm

Perspective raycasting algorithms usually cast rays from the viewpoint through each pixel in the view plane [11], [13]. Each ray is computed iteratively, by determining the next sample location, *sampling*, and then *compositing* the sample onto the current ray. "Sampling" refers to all of the steps required to obtain red, green, blue, and opacity values at a specific point. This includes interpolation of the voxel data, mapping the voxel value to RGBA, and possibly the application of additional shading or lighting equations (see [6]). Any front-to-back rendering operation can be applied in the "Compositing" step. The volumetric compositing equation [18] is used in the implementations in this paper.

The raycasting algorithm is divided into two phases. In the first phase, short ray *segments* are cast, computing a composite color and transparency (i.e., one minus opacity) for each segment. These segments are then used to construct approximations of the full rays in the second phase. For simplicity, it is assumed in the following discussion that the sampling interval along the rays is equal to σ (the voxel sampling interval), but in fact, this rate can be arbitrarily specified.

In Phase 1, the sample points are divided into L levels, $0 \le l < L$, based on their distance from the viewpoint. The distance of the first sample of level l from the viewpoint is defined as D_l. Level 0 consists of a set of ray segments cast from the viewpoint ($D_0 = 0$) to a distance $D_1 - 1$. Level 1 consists of a set of ray segments cast from distance D_1 to distance $D_2 - 1$, and so forth. Under this assumption, each segment in level l consists of $D_{l+1} - D_l$ samples. We refer to the set of segments in level l computed from position p in viewing direction v as $S_l(p, v)$. The set of all segments at all levels is denoted $S(p, v)$. These terms are shortened to S_l or S, respectively, when position and direction are understood.

If we choose to sample each level at the screen size, then Phase 1 results in a set of L two-dimensional arrays of segments, each of size $m \times m$. These planes can then simply be alpha blended to form the final view in Phase 2. This would amount to a simple reordering of the sample and composite operations from the basic brute-force raycasting algorithm [11], and would produce the same output, using the same number of sampling steps. In general, however, each level can be sampled at different resolutions. Let $W_l \times H_l$ specify the horizontal and vertical resolution of the segments cast in level l. Phase 2 resamples each level to screen resolution and then composites the resulting $m \times m$ arrays to form the final view. (Alternatively, the overall resampling work

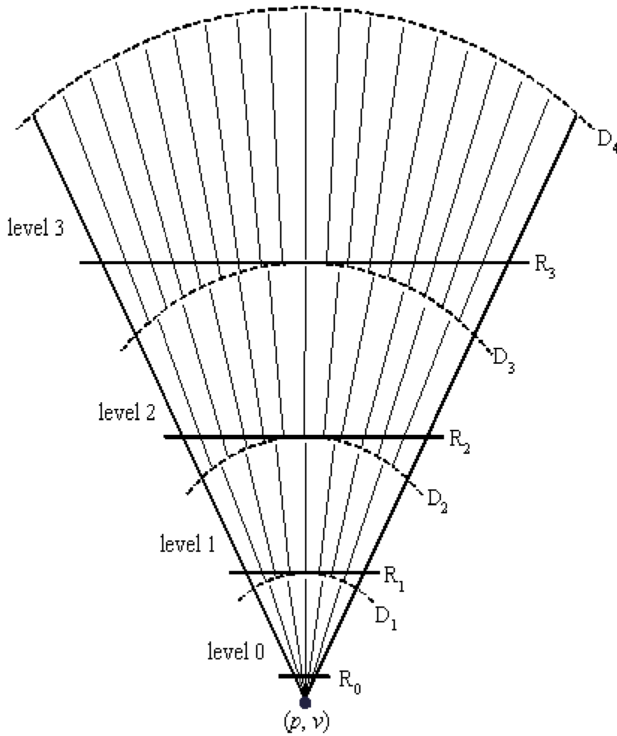


Fig. 1. 2D illustration of the segments computed by the two-phase algorithm. Solid horizontal lines represent the rectangles into which the segments are to be mapped.

can be reduced by resampling each level l to $W_{l+1} \times H_{l+1}$ and then compositing with level $l + 1$, iterating from $l = 0$ to $L - 2$. This is not used in our implementation since the resampling step is performed via 2D texture mapping on the graphics accelerator hardware.) Algorithm 1 below gives a high level overview of the two-phase algorithm.

ALGORITHM 1. Two-Phase Perspective Raycasting Algorithm.

/* Phase 1 */
for l **from** 0 **to** $L - 1$
  /* Generate a $W_l \times H_l$ array of segments at level l */
  for d **from** D_l **to** $D_{l+1} - 1$
    Sample a $W_l \times H_l$ array of rays at distance d from the viewpoint.
    Composite the sample array onto the back of the array of level l segments, S_l.
  end for
end for

/* Phase 2 */
for l **from** 0 **to** $L - 1$
  Resample the segments S_l at screen resolution.
  Composite onto the back of the current view.
end for

Observe that in brute-force perspective raycasting, the lateral sampling rates vary in proportion to the distance from the viewpoint. An advantage of the two-phase algorithm is that it allows a form of adaptive sampling. Each of the layers can be sampled in the horizontal and vertical directions at a rate near that of the underlying data (see Fig. 1). Novins et al. [17], proposed a somewhat similar adaptive sampling approach, with the primary objective of avoiding undersampling of distant sample points by splitting rays and averaging their results. Conversely, our main concern is to avoid oversampling in the regions near the viewpoint, since we are immersed in the data and most of the information is nearby. Thus, we generally set the resolution of each level so that the lateral sampling rates are similar at all levels. Each level's dimensions, $W_l \times H_l$, are then proportional to its distance from the viewpoint, D_l. This means that the first few levels, being near the viewpoint, may cast very few rays. This results in some reduction in the total rendering time. More significant for volume navigation is the fact that most of the reduction in the sampling time is from levels near the viewpoint.

Phase 2 of Algorithm 1 can be phrased as a 2D texture-mapping problem. Each array of rendered segments $S_l(p, v)$ from Phase 1 is defined as a 2D texture, T_l, in Phase 2. For each level ($0 < l < L$), a rectangle R_l is drawn at distance D_l, perpendicular to the view direction, and texture T_l is mapped into the rectangle (see Fig. 1). Level 0 is an exception, since $D_0 = 0$ and thus R_0 would not be visible from viewpoint p. Its rectangle is drawn at distance 1, although anywhere between 0 and D_1 will suffice. The rectangles are drawn in sequence from front to back, and the texture attributes are set so that the rectangles are alpha-blended onto the view to produce a volume-rendered frame.

The two-phase raycasting algorithm is compared to brute-force perspective raycasting below. Brute-force perspective raycasting is a special case of the two-phase algorithm in which $W \times H$ perspective-correct rays are cast to depth D within a single level. For comparison, consider a two-phase partition of L levels of equal-length segments, i.e., $D_i = iD/L$, $i = 0, 1, \ldots, L - 1$. The resolution of each level is chosen to maintain a similar lateral sampling rate at every level:

$$W_i = \frac{D_{i+1}}{D} W = \frac{i+1}{L} W \quad (1)$$

$$H_i = \frac{D_{i+1}}{D} H = \frac{i+1}{L} H \quad (2)$$

The total number of sampling steps for brute-force perspective raycasting (without early ray termination or other acceleration techniques) is $W \cdot H \cdot D$. Two-phase raycasting with the parameters defined above requires

$$\sum_{i=0}^{L-1} W_i H_i (D_{i+1} - D_i) =$$

$$\sum_{i=0}^{L-1} \left(\frac{i+1}{L} W\right)\left(\frac{i+1}{L} H\right)\frac{D}{L} = W \cdot H \cdot D\left(\frac{1}{3} + \frac{1}{2L} + \frac{1}{6L^2}\right) \quad (3)$$

sampling steps in the first phase. In addition, the second phase requires $W \cdot H(L - 1)$ pixel resampling and blending steps.

The performance of the standard and two-phase perspective raycasting algorithms is shown in Table 1 for 8-bit gray and 32-bit RGB-label data. Times were measured on one CPU of a dual-CPU, 300 MHz Intel Pentium®

TABLE 1
COMPARISON OF RENDERING TIMES FOR TWO-PHASE VS. BRUTE-FORCE PERSPECTIVE RAYCASTING

			Brute-force		Two-phase (L=10)			
Data type	D	W, H	Samples (millions)	Time (sec.)	Samples (millions)	Phase 1 (sec.)	Phase 2 (sec.)	Total (sec.)
32-bit RGBL	160	160	4.1	5.54	1.8	2.43	0.04	2.47
		320	16.4	21.72	6.3	8.96	0.17	9.13
		480	65.5	48.49	25.2	19.67	0.31	19.98
8-bit gray	160	160	4.1	3.54	1.8	1.44	0.04	1.48
		320	16.4	13.30	6.3	5.75	0.17	5.92
		480	65.5	29.74	25.2	13.0	0.31	13.31

The field of view is 60º (both horizontal and vertical). (Intel Pentium® II, 300 MHz, 256 MB RAM, NeTpower ULTRAfx2™.)

II platform, running the Windows® NT 4.0 operating system. The system was equipped with 256 MB RAM and a NeTpower ULTRAfx2™ graphics accelerator card. The field of view is 60º, the number of levels is 10, and each segment consists of 16 samples. W and H are set so that the maximum ray separation is approximately σ, $\sigma/2$, or $\sigma/3$, i.e., one, two, or three rays per voxel. These values are chosen to illustrate the algorithms starting at the minimum resolution required to sample all of the voxels, and compare to successively better resolutions. The depth of D is the largest that does not exhaust our system's main memory. With this set of parameters, the two-phase algorithm issues 0.385 times as many sampling steps as the brute-force approach. The total computation time is reduced by a factor of about 0.43 on average. Note that the second phase, implemented as a 2D texture-mapping problem, is accelerated in the graphics hardware.

In the two-phase algorithm, some of the ray segments are obtained by bilinear interpolation from the neighboring segments. In contrast, a correct segment is obtained by interpolating individual sample points and then applying the rendering operation to the samples. Since the compositing operation is nonlinear, reordering the interpolation and the composite steps will incur some error. However, we expect some amount of coherence to exist between segments that are separated by less than one voxel. Levoy proposed an adaptive method to exploit this coherence to accelerate volumetric raytracing [12]. A comparison between Levoy's algorithm and our work is described in Section 6. Kreeger et al. [9] have concurrently developed a technique which divides the volume into levels based on exponential distance from the viewpoint. It creates simple and regular resampling patterns, utilizing a slice-order algorithm for volume access and thus partitions the levels along volume slices.

Figs. 2 and 3 contrast the quality of the two-phase algorithm with brute-force, for a labeled RGB and a gray scale volume, respectively. They use the same set of parameters used to produce Table 1. The first column contains the brute-force raycast, the second contains the image produced by the two-phase algorithm, and the third contains their absolute difference, multiplied by 10 so that the values are visible.

Fig. 2 was produced using the Visible Human™ Male RGB dataset, plus 8-bit labels segmenting the anatomy. The data was downsampled to 1 mm resolution in all three axes, to a size of $584 \times 340 \times 1{,}878$. Opacity parameters were set to display bone, nerves, circulatory and respiratory systems. The view is from the neck, looking down the spine. In Fig. 2, the three rows represent the three different resolutions 160, 320, and 480. The images have been printed at different resolutions so that they appear the same size in the diagram. Notice that in all three, a few small areas of sharp difference appear near edges which border pixels that have clipped at the back of the frustum.

Fig. 3 is a $256 \times 256 \times 154$ CT image of a canine thorax with 0.703 mm × 0.703 mm × 0.703 mm voxel spacing, interpolated from forty slices of 3 mm thickness. In this case, only the 480 resolution images are shown. The material was given a large proportion of diffuse reflectivity to highlight the walls of the trachea. Notice that errors seem to be concentrated around sharp changes in the diffuse reflection.

4 INTERACTIVE NAVIGATION

A major advantage of the two-phase algorithm is that many of the segments computed in Phase 1 can be used to construct approximate frames from viewpoints near the base position. We use them for viewpoints within a radius δ of the base position, and within angle θ of the original viewing direction for which the segments were computed. In particular, at each step we recompute the first λ levels, but reuse the last $L - \lambda$ levels. Note that since the levels near the viewpoint contain fewer segments, these are inexpensive to compute. Thus, given a set of segments $S_l(p, v)$, $\lambda \le l < L$, an approximate view within (δ, θ) of (p, v) can be quickly computed.

A straightforward implementation of this idea would initialize the $S_l(p, v)$ data structure from the current viewpoint. Then a sequence of frames can be quickly produced under interactive control, until the position or orientation exceeds one of the thresholds. At this point, a fresh set of segments $S_l(p', v')$ is computed, and the process continues. Unfortunately, this produces jerky motion due to the periodic pauses to recompute the segment data structure. Instead, we amortize the time to compute new segments assuming that we can reliably estimate the location at which the update will be required. This is true for long sweeps of translation in a fixed direction or rotation about a fixed axis. We describe acceleration techniques for these two cases below.

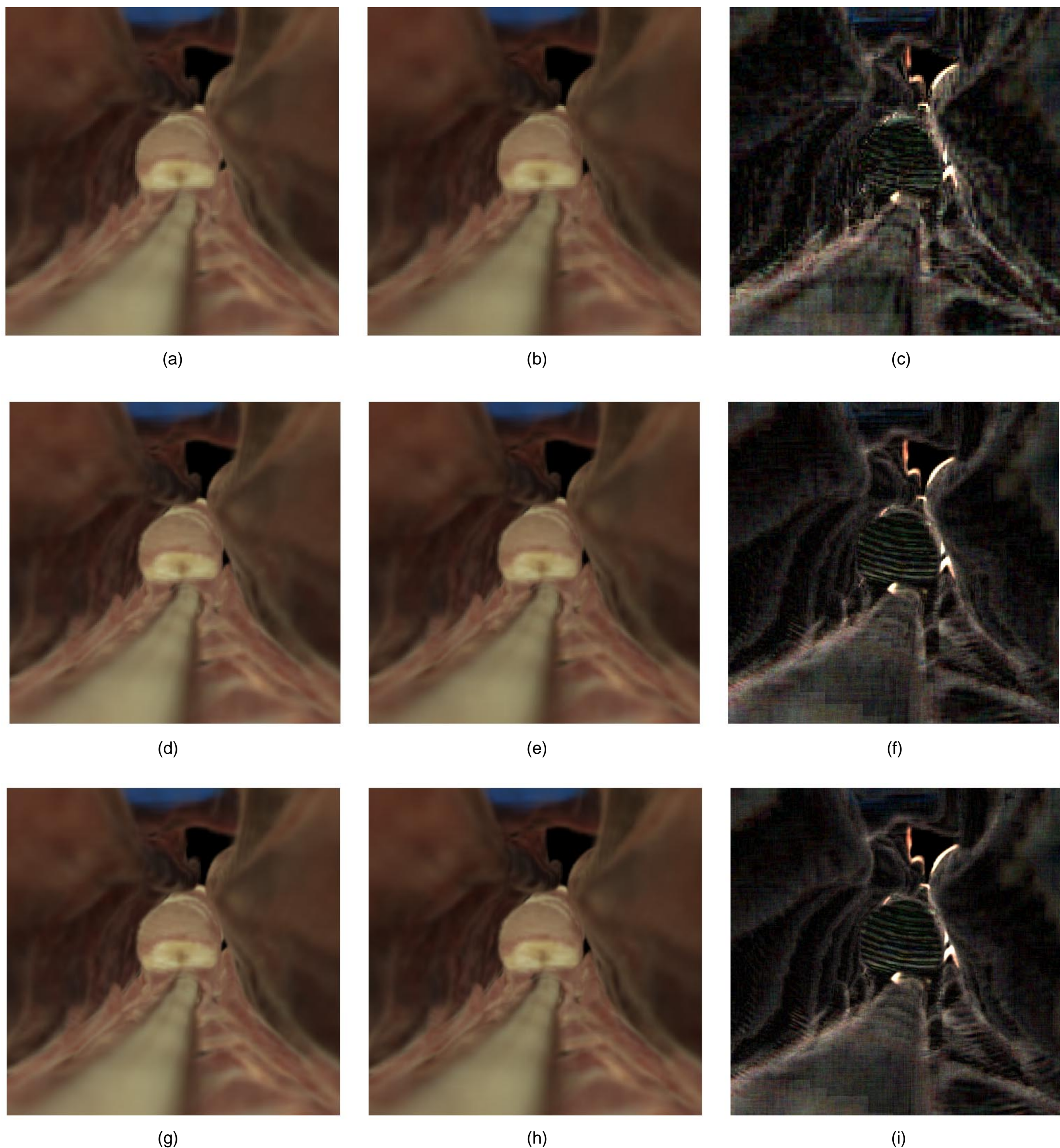
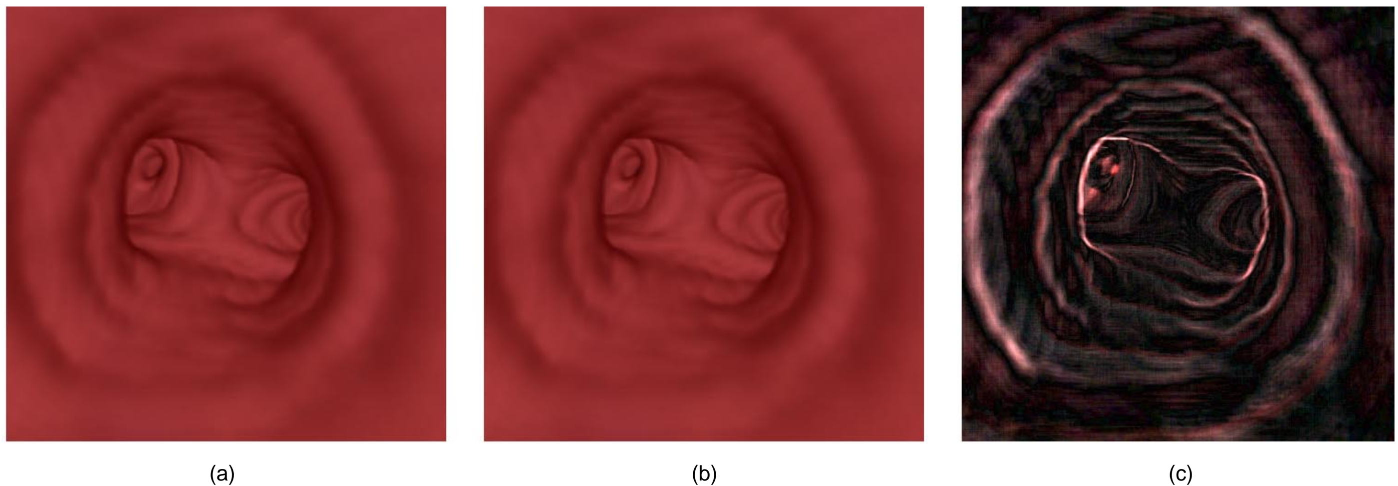


Fig. 2 Comparison of rendering of lableled RGB data. (a) (d) (g) The first column shows a brute-force perspective raycast. (b) (e) (h) The second column shows the results of the two-phase algorithm. (c) (f) (i) The third column is the absolute difference between the first two, multiplied by 10 to make the differences visible. (a) Brute-Force, W = 160. (b) Two-Phase, W = 160. (c) Difference X 10, W = 160. (d) Brute-Force, W = 320. (e) Two-Phase, W = 320. (f) Difference X 10, W = 320. (g) Brute-Force, W = 480. (h) Two-Phase, W = 480. (i) Difference X 10, W = 480.

4.1 Viewpoint Translation

Consider the case of forward movement. If we move a distance Δp in each frame, then a total of $f = 2\delta/\Delta p$ frames are within the scope of a single data structure, $S(p, v)$. The next predicted data structure is $S(p + 2\delta, v)$. Therefore, we amortize the cost to compute $S(p + 2\delta, v)$ by completing $1/f$ of the new data structure in each step. Then, after f steps, we swap in the new data structure and repeat.

A pseudocode overview of the forward translation algorithm is shown below in Algorithm 2 and illustrated in Fig. 4. Notice that while the positions of the first λ levels move about with the viewpoint, the last $L - \lambda$ levels are fixed, relative to the base position p. Thus, the number of samples within the first λ levels depends upon its offset from p. This is handled by varying the depth of level $\lambda - 1$ by the amount of the offset, δ. Second, for each step forward,



(a) (b) (c)

Fig. 3 Rendering comparison of gray-scale data. (a) The result of a brute-force perspective raycasting. (b) The two-phase algorithm. (c) The difference between the two multiplied by 10. (a) Brute-Force, W = 480. (b) Two-Phase, W = 480. (c) Difference X 10, W = 480.

TABLE 2.
COMPARISON OF RENDERING TIMES FOR COMPLETE TWO-PHASE PERSPECTIVE RAYCASTING WITH THE TIMES FOR INCREMENTAL FORWARD MOVEMENT AND INCREMENTAL ROTATION

$L = 10$, $\lambda = 2$, $\delta = 8$, $f = 16$, $\theta = 3°$			Complete two-phase raycasting		Two-phase incremental translation		Two-phase incremental rotation	
Data type	D	*W*, *H*	Samples (millions)	Time (sec.)	Samples (millions)	Time (sec.)	Samples (millions)	Time (sec.)
32-bit RGBL	160	160	1.8	2.47	0.12	0.26	0.097	0.27
		320	6.3	9.13	0.47	1.09	0.39	1.01
		480	25.2	19.98	1.88	2.15	1.56	2.17
8-bit gray	160	160	1.8	1.48	0.12	0.16	0.097	0.16
		320	6.3	5.92	0.47	0.63	0.39	0.61
		480	25.2	13.31	1.88	1.26	1.56	1.24

The field of view is 60º (both horizontal and vertical). (Intel Pentium® II, 300 MHz, 256 MB RAM, NeTpower ULTRAfx2™.)

one plane of samples should be rendered onto the back of the last level, so that the total viewing depth remains constant. This is achieved by adding a level L to the back of the data structure that is updated at every step. We have omitted this detail in Algorithm 2 to avoid overcomplicating the discussion. Finally, notice that for positions behind the base position (p, v), rays near the border of the frustum may pass outside the border of $S(p, v)$. To avoid this, we compute S at a slightly larger field of view than the viewing frustum to accommodate all positions within radius δ of the base. Specifically, for a square field of view of angle ϕ, each rectangle R_l is expanded by a border of $\delta \cdot \tan(\phi/2)$ on each side.

ALGORITHM 2. Incremental forward translation.

Move viewing frustum forward to *p+offset*.
/* Phase 1 */
/* Recompute S_0 to $S_{\lambda-2}$ from position *p+offset* */
for l **from** 0 **to** $\lambda - 2$
  for *depth* **from** D_l **to** $D_{l+1} - 1$
    Sample an array of $W_l \times H_l$ points at distance *depth* from position *p+offset*.
    Composite onto the back of level l segments, S_l.
  end for
end for
/* Recompute $S_{\lambda-1}$ from position *p+offset* */
for *depth* **from** $D_{\lambda-1}$ **to** $D_\lambda - 1$ + *offset*
  Sample an array of $W_{\lambda-1} \times H_{\lambda-1}$ points at distance *depth* from position *p+offset*.
  Composite onto the back of level l segments, S_l.
end for
/* Amortized update of next data structure */
Compute $1/f$ of the segments $S_l(p + 2\delta, v)$, $\lambda \le l < L$.
if *offset* $> \delta$
  for l **from** λ **to** $L - 1$
    Swap $S_l(p + 2\delta, v)$ in for S_l.
  end for
  offset $= -\delta$
end if

/* Phase 2 *
for l **from** 0 **to** $L - 1$
  Resample segments S_l at screen resolution at position *p+offset*.
  Composite onto the back of current view.
end for

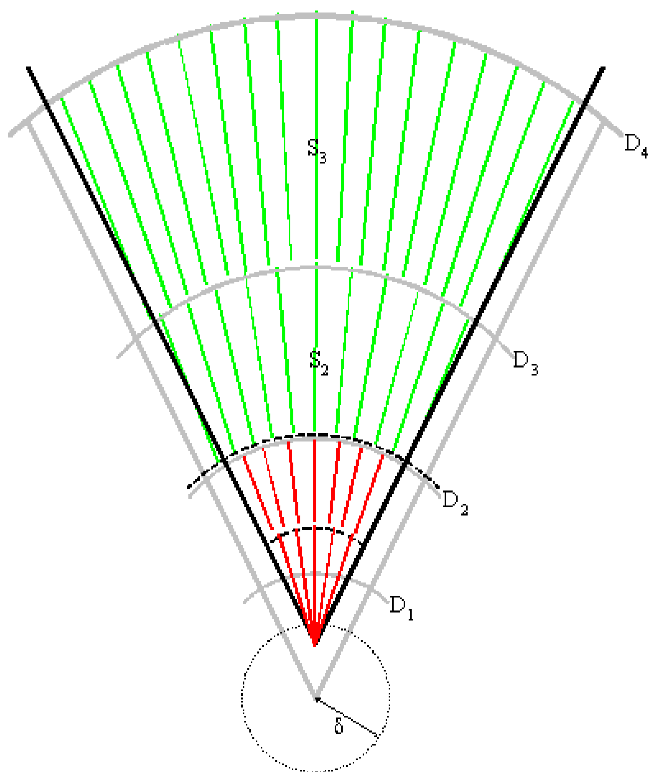


Fig. 4. 2D depiction of forward translation. Gray lines represent the view from the original position, while black lines indicate the position of the translated view. In this example, $\lambda = 2$; i.e., the first two levels are recomputed (the recomputed segments are depicted in red), while the last two (green) are simply resampled from the segment data structure from the new position. Note that the depth of level λ is reduced by the offset distance, so that level $\lambda - 1$ samples just up to the front of the original S_2.

The times to compute an incremental forward step using Algorithm 2 are compared to two-phase raycasting from scratch in Table 2, using $\lambda = 2$ and $f = 16$. The radius of the segment data structure scope is set to $\delta = 8$, meaning that each forward step moves the viewpoint forward by one voxel. We observe approximately a factor of 9 speedup from amortizing the cost to compute the back $L - \lambda$ levels over 16 steps.

For a fixed step size Δp, the value of f trades off image quality for frame rate. Higher values of f amortize the segment computation over greater number of frames. However, this corresponds to a higher maximum offset, δ. At a forward offset position (p+*offset*, v), the orientation of segments $S(p, v)$ on the periphery of the view do not match that of the rays cast from the current position. This orientation error increases with the offset. Consider a segment s in level l that is to be approximated at view position p+*offset* by resampling $S_l(p, v)$ (see Fig. 5). The segment is obtained by bilinearly interpolating the four segments nearest to s, based on their distance from s. However, segment s is not parallel to any of the neighboring segments in $S_l(p, v)$, so it is difficult to define such a distance. One could take an average distance of the individual sample points in the segments, but the resulting color and opacity of a segment is a nonlinear function of the individual sample values, highly dependent on individual opacities, and tends to emphasize the nearby samples. In our implementation, each level $S_l(p, v)$ is mapped into a rectangle R_l at distance D_l, perpendicular to v. Thus, the resampling depends upon relative distances within the plane of this rectangle. Near the center of the view, where the difference in orientation of s from its samples is small, the distances are based on the position of the front of the segments. Toward the edges of the view, the weighting is closer to the middle of the segments. This tends to favor resampling accuracy in the center of the view.

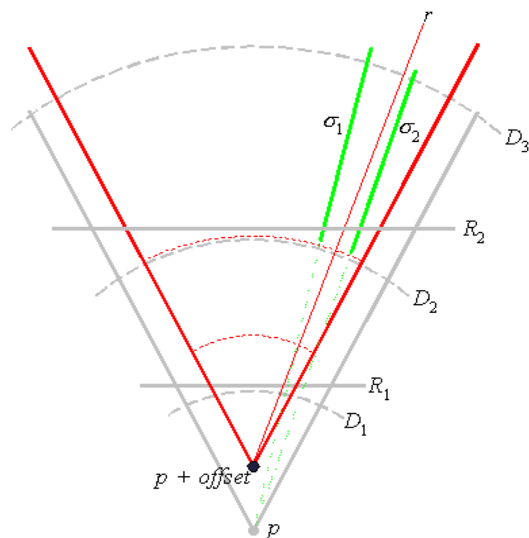


Fig. 5. A 2D illustration of the resampling of S during translation. A ray r cast from position p+*offset* (red) is resampled in level 2 using segments σ_1 and σ_2 cast from position p (gray). The bilinear interpolation is based on the relative positions of r, σ_1, and σ_2 projected into R_2.

Fig. 6 shows the labeled RGB view of the Visible Human™ Male computed at a maximum translational offset of $\delta = 8$ (recall that the distance units are $\sigma = 1$ mm in this case). The difference between this offset image and the brute-force image, shown in Fig. 6b, indicates that additional error beyond that which arose from the two-phase algorithm is distributed throughout the image, near areas of high gradient.

4.2 Viewpoint Rotation

For rotation, a data structure $S(p, v)$ is computed for a field of view $f\Delta v/2$ degrees wider than the size of the viewing frustum in the direction of rotation, where Δv is the size of the incremental rotations. This allows a sequence of f rotations before rays reach the edge of the data structure. In each step, $1/f$ of the next data structure, $S(p, v + f\Delta v)$, is computed. This is illustrated in Fig. 7, and outlined in Algorithm 3. Notice that levels 0 through $\lambda - 1$ do not need to be recomputed at each step, assuming that the field of view is increased by $f\Delta v/2$. Refer to the rightmost columns in Table 2 for measurements of the performances of incremental update. These numbers are quite similar to those measured for forward translation. The number of steps is $f = 16$ and the rotation step size $\Delta v = 0.325°$.

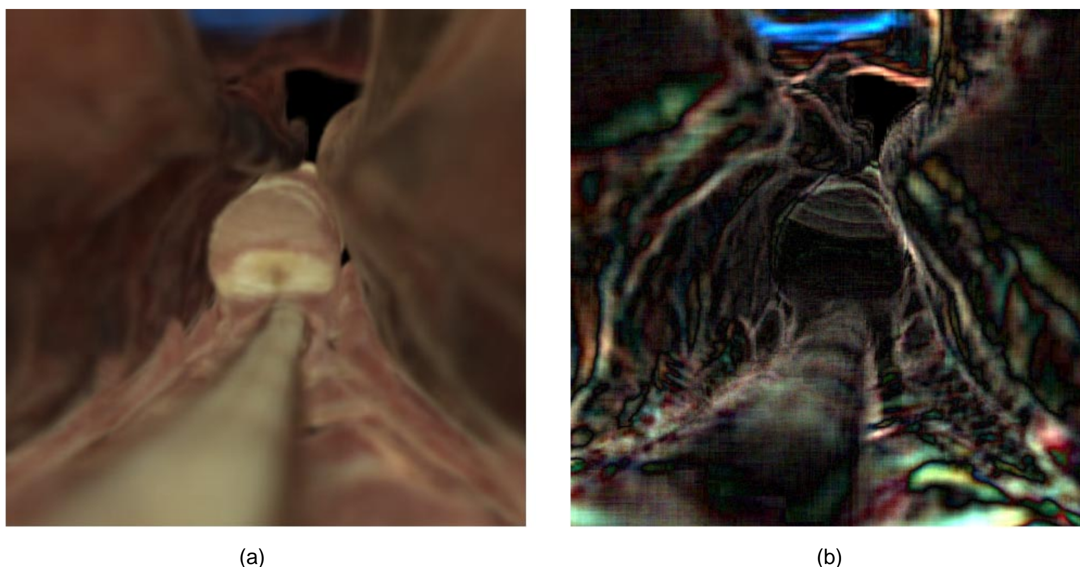


Fig. 6. Illustration of translational offset error. The same view as in Fig. 2 is rendered using the two-phase algorithm with translational offset of 8σ. the difference with respect to brute-force raycasting (Fig. 2g) is shown in (b). (a) Two-phase algorithm, 8σ forward-offset, W = 480. (b) Difference x 10, W = 480.

ALGORITHM 3. Incremental rotation about a fixed axis.

Rotate viewing frustum to v+*offset* at position p.
/* Phase 1 */
/* Update next data structure */
Compute $1/f$ of the segments $S_l(p, v + 2\theta)$, $\lambda < l < L$
if *offset* > θ
  Recompute levels 0 through $\lambda - 1$ from orientation $v + 2\theta$ at position p.
  for l **from** λ **to** $L - 1$
    Swap $S_l(p, v + 2\theta)$ in for S_l.
  end for
  offset = $-\theta$
end if

/* Phase 2 */
for l **from** 0 **to** $L - 1$
  Resample segments S_l at screen resolution at orientation v+*offset*.
  Composite onto the back of current view.
end for

Rotations tend to have greater resampling accuracy at large offsets than translations, because no error in the orientations of the segments occurs. If the point of view remains at p during rotation, the orientations of the segments at the new viewing direction are consistent with those already in the data structure. Thus, the relative distance between a segment at the new viewing direction and its four nearest neighbors is well defined, and is handled properly by the resampling on the projected rectangle. Fig. 8 shows the labeled RGB view of the Visible Human™ Male computed at a maximum rotational offset of $\theta = -3.0°$. In Fig. 8b, the absolute difference from the brute force raycast shows little variation beyond that already attributed to the two-phase algorithm.

5 SUBCUBE MAINTENANCE

Voxel data sets suitable for navigation are typically quite large and may not fit into main memory. Instead, data within the current view frustum must be paged in from disk during navigation. In particular, an $n \times n \times n$ subcube of the volume enclosing the view frustum is maintained within main memory. The subcube must be updated as the view position and orientation change without significant impact on the frame rate. For example, consider a subcube with origin located at (x_0, y_0, z_0) within an isotropic volume data set. To move the subcube one voxel in the positive x direction, an $n \times n$ yz-plane of data must be extracted from the volume and inserted at the right side of the cube, and an $n \times n$ plane $(x = x_0)$

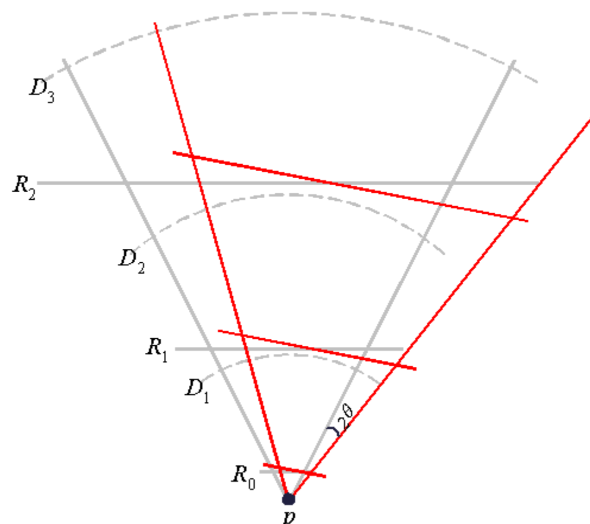
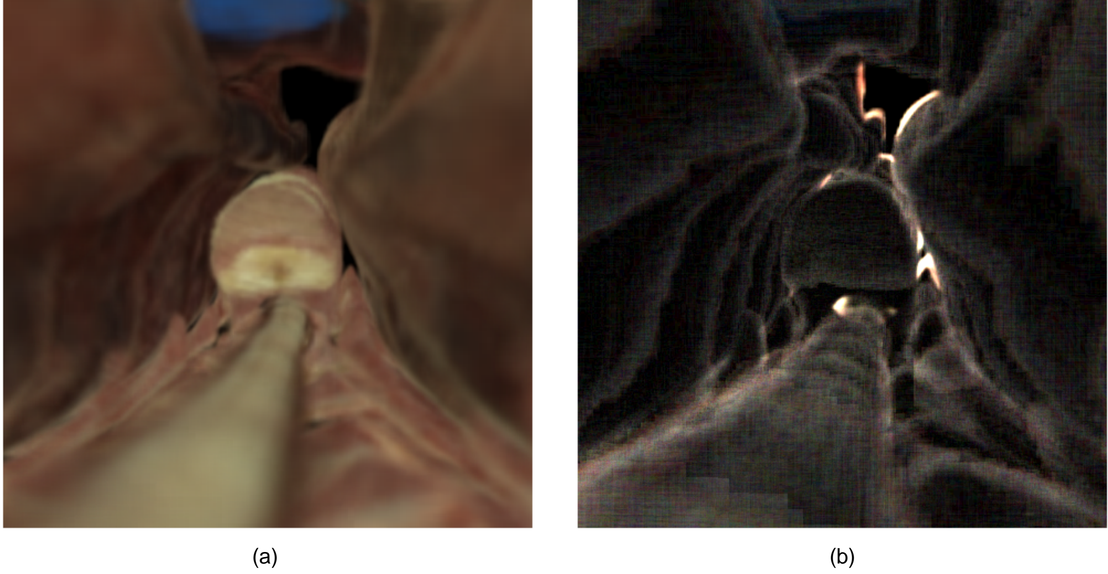


Fig. 7 The initial segment data structure is depicted in gray, and should be 2θ wider that the field of view. The next segment data structure for right rotation is depicted in red.



(a) (b)

Fig. 8 Illustration of rotational offset error. The same view as in Fig. 2 is rendered using the two-phase algorithm with rotational offset of –3 degrees. The difference between this (a) and the brute-force raycasting (Fig. 2g) is shown is (b). (a) Two-phase, –3 degree rotational offset, W = 480. (b) Difference x 10, W = 480.

must be deleted from the left side of the cube. These two steps can be combined by writing the new plane of data into the locations vacated by the deleted plane.

The organization of the volume on disk and the subcube in memory greatly affect the efficiency of subcube update operations. Mapping functions must be specified for both which generate linear address offsets from spatial coordinates. Let $V(x, y, z)$ and $C(x, y, z)$ denote the mapping functions for the volume and subcube, respectively.

First, consider the retrieval of volume data from disk. The time to read an $n \times n$ plane tends to be dominated by the latency involved in initiating block transfers. This is because the data within an arbitrary an $n \times n$ plane is not contiguous. For example, in a standard 3D array layout ($V(x, y, z) = x + N_x y + N_x N_y z$), one must usually seek to the beginning of each row, and in the worst case—a *yz*-plane—must seek for every voxel. The overhead can be amortized somewhat by loading more than one plane at a time. To further reduce this overhead, we use a block-oriented mapping in which $b \times b \times b$ blocks of the volume are stored in contiguous addresses. The subcube is then maintained on *b*-voxel boundaries. (N_x, N_y, N_z, and n must be divisible by b, which may require padding the volume data set by a small amount.) A b-unit move along any axis loads n^2/b^2 new blocks from disk, and each block contains b^3 contiguous voxels. In general, the block-oriented mapping is described by

$$V_b(x, y, z) = b^3\left\lfloor \frac{x}{b} \right\rfloor + (x)_{\bmod b} + N_x b^2 \left\lfloor \frac{y}{b} \right\rfloor + (yb)_{\bmod b^2} + N_x N_y b \left\lfloor \frac{z}{b} \right\rfloor + \left(zb^2\right)_{\bmod b^3} \quad (4)$$

For the first voxel in a block, in which x, y, and z are multiples of b, this reduces to

$$V_b(x, y, z) = b^2 x + N_x b y + N_x N_y z \quad (5)$$

The goal for the layout of the subcube is a mapping that is independent of subcube position, so that data already loaded need not be moved when the subcube moves. Furthermore, the mapping should be simple to compute since it will be computed every time a sample is accessed. The mapping that we propose is

$$C(x, y, z) = \left(x + ny + n^2 z\right)_{\bmod n^3} \quad (6)$$

For any $n \times n \times n$ subcube with origin located at (x_0, y_0 z_0), it is simple to show that C maps the n^3 subcube voxels, $I(x, y, z)$ for $x_0 \le x < x_0 + n$, $y_0 \le y < y_0 + n$, $z_0 \le z < z_0 + n$ one-to-one onto the integers 0, 1, 2, …, $n^3 - 1$. Therefore, each voxel of any subcube has a unique address, and new planes of data written into the subcube will overwrite the planes that are to be deleted from the previous subcube position. Note also that if n is chosen to be a power of two, C is particularly inexpensive to compute.

The performance of the block-oriented subcube update is contrasted below in Table 3 with that of a standard array layout. The time to load $16 \times 256 \times 256$ and $256 \times 256 \times 16$ slabs of voxels are measured, corresponding to motion along the x and z axes, respectively. Times were averaged over many successive loads. The timings vary greatly depending upon the likelihood that a given block transfer loads data already residing on the system file cache. If one is moving around within a small subregion of the volume, the hit rate can be essentially 100 percent. In contrast, a long, straight flight down the *z*-axis tends to incur file cache misses on every slab load. We bound these behaviors by presenting “best” and “worst” case paths which tend to maximize and minimize, respectively, file cache performance.

TABLE 3
COMPARISON OF 3D BLOCK LOAD TIMES FOR *X*- AND *Z*-ORIENTED BLOCKS OF DATA FROM A LARGE VOLUME FILE

	Block-oriented (b = 16)		3D array	
	$16 \times 256 \times 256$ (msec.)	$256 \times 256 \times 16$ (msec.)	$16 \times 256 \times 256$ (msec.)	$256 \times 256 \times 16$ (msec.)
"Warm" file cache	34	30	930	120
"Cold" file cache		455	940	1,100

3D array is a standard x-y-z layout.

Note that in the x-direction, a 3D array layout generates $n^2 = 256^2$ separate, noncontiguous data transfers. The overhead required to initiate so many block transfers appears to dominate the block load, independent of whether the data is in the file cache or retrieved from disk (see the next-to-last column of Table 3). Along the z-axis, the number of block reads is reduced to $bn = 256 \times 16$ for an array layout. The block-oriented layout requires at most $(n/b)^2 = 256$ block transfers along any of the axes. Thus, when the file cache performance is good, the transfer times reduced by more than an order of magnitude. In the worst case, the block-oriented transfer rate becomes dominated by file cache misses, although this is still a factor of two improvement over the array layout.

Finally, note that the subcube to some extent decouples the layout of the volume data set from the data organization required by the two-phase raycasting algorithm. For instance, blocks could be stored in a compressed format. (The volume mapping function would need to be modified.) During a subcube update, blocks would be loaded, decompressed, and then inserted into the subcube. This technique trades the reduction in disk space and I/O bandwidth requirements against increased CPU cycles. As a second example, nonisotropic data could be stored in the block-oriented format, and resampled to the subcube resolution after loading from disk.

6 DISCUSSION

Numerous techniques have been proposed for accelerating the volumetric raycasting process. Many of these focus on avoiding unnecessary sampling steps. We discuss some of the main techniques in the context of our volume navigation scenario.

Levoy [13] proposed early termination of rays cast front-to-back based on their accumulated opacity. This method can produce dramatic improvements when raycasting through an entire optically dense volume. We would expect this technique to be less effective when navigating within a depth-limited frustum, since rays are already short. Furthermore, the method does not appear to be compatible with our two-phase algorithm. Short segments are rendered independently, and are far less likely to saturate than long rays. Thus, the potential gain from early termination of the segments is much smaller. In addition, it is difficult for a segment to take advantage of the accumulated opacity between the viewpoint and the front of that segment, because the segment will be reused from different positions that may present less accumulated opacity.

Many methods have been suggested for skipping over transparent regions of the volume by encoding information about the data's coherence into the volume. This tends to be most effective when the voxel opacities are known apriori. Levoy [13] describes an algorithm for raycasting though an octtree-encoded data structure. Zuiderveld et al. [24] encode transparent voxels with the distance, d, to the nearest nonempty voxel, thus allowing the next d voxels along a ray to be skipped. Such techniques may be able to further improve the speed of our algorithm. We have experimented with a very limited form of encoding in our RGB-label implementation. We reserve one label value in each voxel $I(x, y, z)$ to indicate that the current voxel and seven neighbors, $I(x + i, y + j, z + k)$, $i, j, k \in \{0, 1\}$ are all transparent. We use this information to skip most of the computation associated with a single sample in this empty region.

Levoy [12] proposed an adaptive refinement technique that takes advantage of coherence within adjacent rays. In Levoy's method, a sparse grid of rays is cast initially. In regions of sharp contrast, the number of rays cast is adaptively increased, while in highly coherent areas of the image the inter-ray pixels are simply interpolated. Note that this is the coherence that our two-phase algorithm uses to justify the casting of fewer rays near the viewpoint. In our algorithm, the inter-segment resolution is nonadaptive however. It may be possible to improve image quality near the viewpoint by computing extra segments in high contrast regions using this technique.

One method that has been used to accelerate the computation is to convert the volume data into a surface-based representation using an isosurface extraction algorithm such as Marching Cubes [15], [14]. This preprocessing step is time consuming, but is done only once. The resulting isosurface representation is compatible with the standard 3D rendering pipeline, and thus takes advantage of standard 3D graphics acceleration hardware. This method has proven useful for navigating through data that contains reasonably well defined surfaces, such as the colon wall within 3D radiological images [16], [21]. A disadvantage of this method is that much of the information contained within the 3D data set is lost in the conversion to isosurfaces.

Various techniques have been suggested that use 2D or 3D texture-mapping hardware to accelerate volume rendering [5], [20], [22], and [23]. Many of these require 3D texture mapping hardware, which is not often available on PC graphics accelerators. Our implementation uses only 2D texture mapping, which is often implemented even in relatively inexpensive PC graphics hardware.

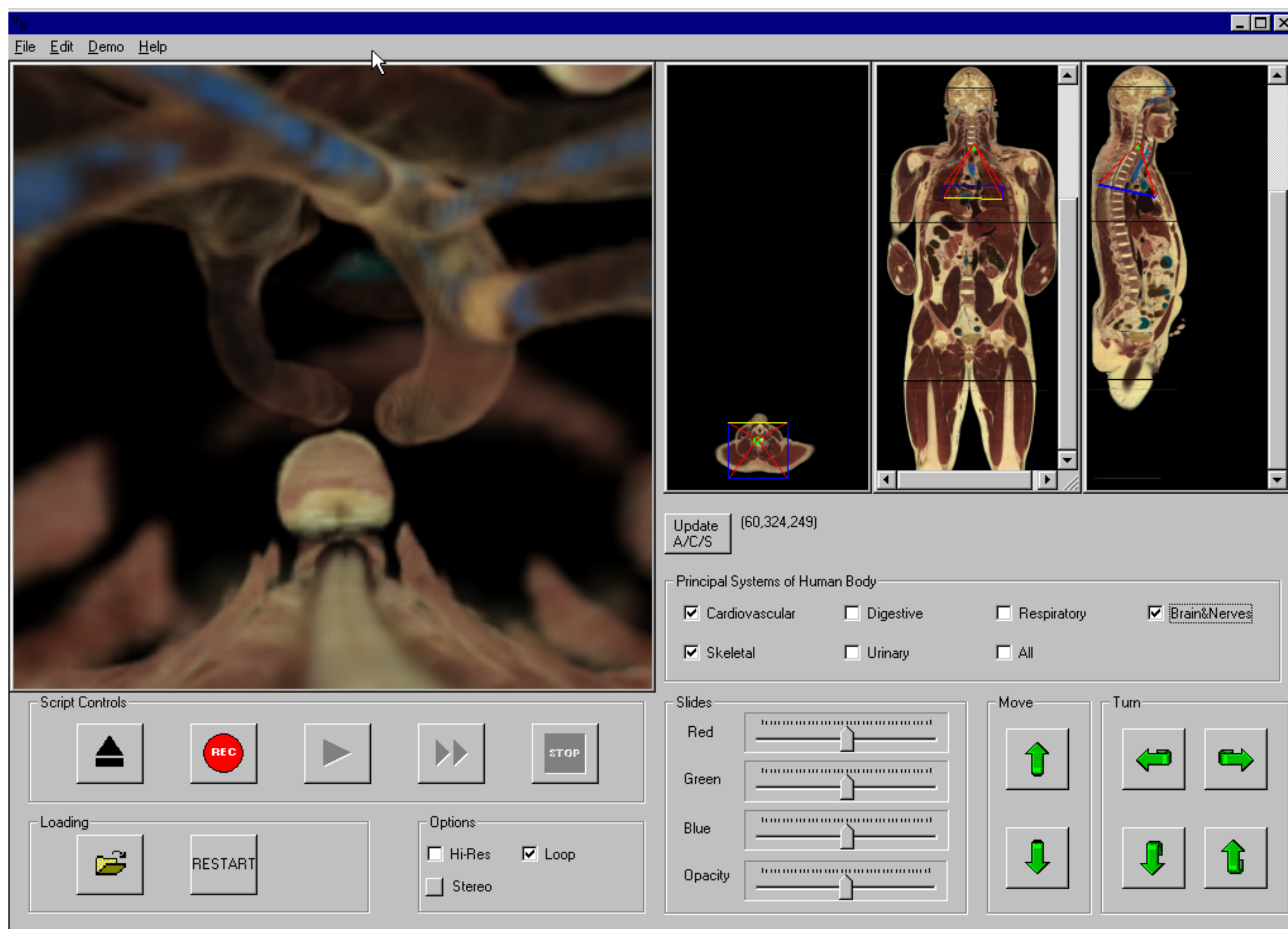


Fig. 9. Screen capture of our volume navigation application.

7 Volume Navigation Application

A volume navigation application was implemented in order to evaluate the methods presented. A view of the main navigation console is shown in Fig. 9. The navigation window, in the upper left of the console, displays the volume navigation view. Axial, Coronal, and Sagittal (ACS) slices from the current position are displayed in separate panes. These slices are updated only during a pause in user input, to avoid impacting the navigation speed. Controls allow the user to move forward or backward, or to rotate right, left, up, or down. Full control of the volume rendering parameters, such as number of levels, resolution and depth of each level, sampling rate, navigation window size, movement increment, distance between data structure updates, color maps, lighting, etc., can be modified in auxiliary windows. The application can take 8-bit gray scale, 24-bit color, and 32-bit color-plus-label voxel data types as input. Sampling can be trilinear or nearest neighbor at each level.

A restricted form of lighting is implemented. It assumes that the light is a point source positioned at the viewpoint. The lighting computation is simplified because the light and ray directions are the same for all rays. The amount of diffuse reflection can thus be estimated at any sample point as the difference of the point's two neighbors along that ray. We do not normalize the gradient in this computation. Larger point differences generate greater diffuse reflections. More-general lighting models are compatible with the algorithm, but highly accurate models are less critical during fast, reduced-resolution navigation relative to the extra computation incurred.

During a pause in user input, the application can be set to use the idle time to render and display progressively higher resolutions of the current position. Additional delay is used to progressively extend the depth of the rays by blending samples from a reduced-resolution volume onto the back of the rays.

7.1 Implementation

The volume navigation application is a multithreaded implementation, whose major rendering blocks are illustrated in Fig. 10. Recast and Update comprise Phase 1 of the algorithm. Recast recomputes the first λ levels of segments from the current viewpoint. Update completes $1/f$ of the segment rendering of levels λ through $L-1$ from the predicted future location. Their computations consist primarily of determining sample positions, trilinear resampling, lighting, and blending. Display corresponds to Phase 2. It resamples and blends each level at the screen resolution. In our implementation, Display is computed almost entirely on the

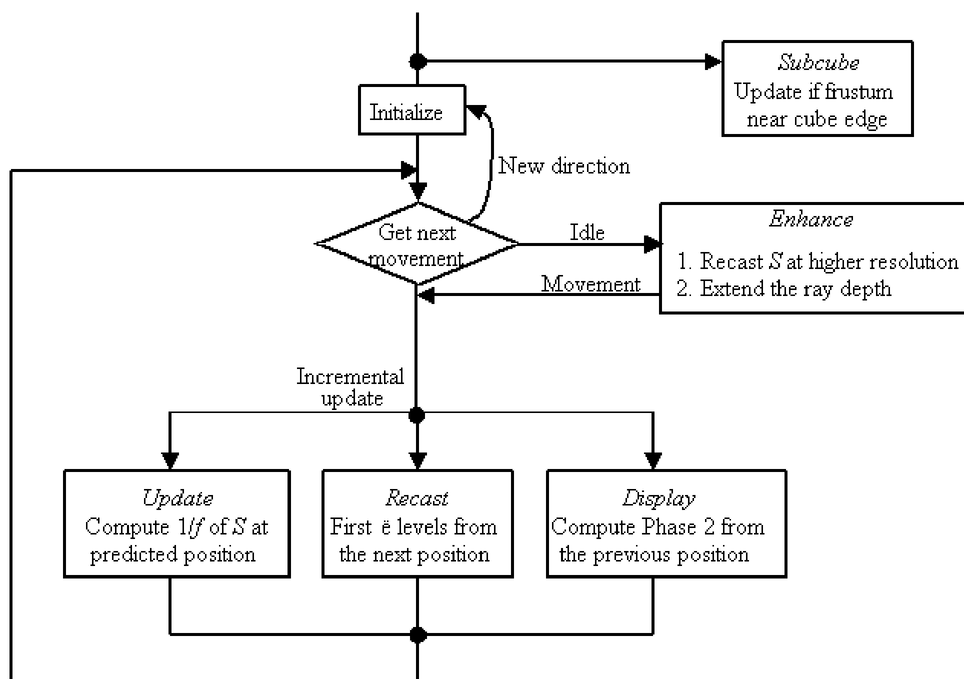


Fig. 10. Illustration of the major blocks of the multithreaded volume navigation application.

TABLE 4
COMPARISON OF TOTAL UPDATE TIMES FOR TWO-PHASE PERSPECTIVE RAYCASTING WITH THE TIMES FOR INCREMENTAL FORWARD MOVEMENT AND INCREMENTAL ROTATION

Data type	Two-phase incremental translation (msec.)	Two-phase Incremental rotation (msec.)
32-bit RGBL	153	152
8-bit gray	105	106

The field of view is 60º (both horizontal and vertical). (Intel Pentium® II, 300 MHz, 256 MB RAM, NeT-power ULTRAfx2™.) Additional rendering parameters are L = 10, $\lambda = 2$, $\delta = 8$, f = 16, D = W = H = 160.

graphics acceleration hardware. The segment data is assembled into textures and dumped onto the accelerator for resampling and alpha blending.

The three main rendering blocks are assigned three different threads, and can be executed concurrently. The three threads synchronize with each other after each incremental movement. Update tends to dominate the computation time, and so it is internally multithreaded in order to improve the load balance. Note that the Recast and Update computations are independent, but Display requires the results from Recast. To allow these two routines to operate in parallel, Display produces the frame from the previous site, (p_i, v_i) while Recast is computing the new segments for the next position, (p_{i+1}, v_{i+1}). When a change in the current motion occurs, the predicted future location is no longer correct, and so the segment data structure is reinitialized for the new motion.

A fourth major block, Subcube, maintains the subcube in main memory. This block receives its own thread at the start of the program and runs throughout the application as an independent routine, monitoring the current user position, and updating the subcube when the user begins to get too close to one of the subcube boundaries. It is essential that I/O runs on an independent thread, even on a single-CPU platform, because its computations must be amortized over many frames.

During a pause in user input, the view is refined in the Enhance block in two ways. First, a new set of segments is computed at increased resolution (the factor increase can be specified). Next, we begin extending the ray depth by adding an extra layer of segments onto the back of the current view, and progressively updating the view. These extended-depth segments are obtained by interpolating from a down-sampled copy of the volume that fits entirely within main memory (this copy is used also to obtain the ACS slices in the auxiliary windows).

7.2 Performance Measurement

The volume navigation application was evaluated on a dual-CPU, 300 MHz Intel Pentium® II platform. The platform was equipped with 256 MB RAM and a NeT-power ULTRAfx2™ graphics accelerator card, running the Windows® NT 4.0 operating system. Times were measured for both 8-bit gray scale data (using both color and opacity look up tables) and 24-bit RGB plus 8-bit label data (the labels are used to look up opacity). Lighting was applied to each sample, with the light is positioned at the view-point.

TABLE 5
AVERAGE CYCLES FOR SAMPLING AND BLENDING ONE RAY SAMPLE

	8-bit gray (cycles)	24-bit RGB (cycles)	32-bit RGBL (cycles)
Addressing	29	29	82
Trilinear interpolation	61	331	87
Shading	24	45	54
Alpha blending	25	25	11
Total	139	430	234

Table 4 shows the times for incremental movement within the multithreaded application. Translation and Rotation represent the average time to compute a single frame during a long sequence of forward and rotate steps, respectively. These represent "perfect" prediction by the algorithm in computing the next segment data structure. When the direction of motion changes, the data structure must be recomputed, using Algorithm 1.

The main rendering subroutine, responsible for interpolating, shading, and alpha blending an individual sample point was hand optimized in assembler, making use of the Intel MMX™ instruction set. The average cycle counts for these operations are broken down in Table 5 for 8-bit gray, 24-bit RGB, and 32-bit labeled RGB data. In the gray scale sample and blend routine, the opacity is obtained by table look up on the interpolated gray data. In RGB, a gray value is computed from the interpolated red, green, and blue channels, and opacity is then mapped through a look up table. RGBL sampling uses a special data encoding to mark empty voxels whose neighbors are also empty. A sample point lying in this neighborhood does not contribute to the ray and can be skipped. We measure that on average, 32 percent of our voxel samples are skipped in way. Thus, the total time is lower than the RGB sampling time.

8 CONCLUSIONS

The algorithms presented provide considerable speedup over brute-force perspective volumetric raycasting. They provide a unique balance between speed and quality for interactive volume rendering by taking advantage of interframe coherence. There are a large number of parameters available to trade off these two goals, and we expect that the settings we have chosen can be improved.

For each successive frame during a pause, we successively recompute high-resolution renderings for level 0, then level 1, etc., and update the view at each step. Thus, the longer the idle time, the farther back into the view the high quality rendering proceeds. Instead of, or in addition to increasing the quality of the rendered segments, one can incrementally increase the depth of the viewing frustum. This can be implemented by adding a level $L + 1$ to the back of the view. We then iteratively sample at depth D_l, $D_l + 1$, etc., and update level $L + 1$ in each iteration. This will require modification of the dynamic I/O procedure that maintains the current volume region in RAM to obtain samples beyond the end of the normal view frustum. However, after a band of distant data is composited, it can be deleted to make room for the next band.

Notice that it is inexpensive to create stereo pairs using the segment data structure S, as long as the two eye positions and orientations are within (δ, θ) of the central view. This requires computing the less-expensive front levels, 0 through $\lambda - 1$, at two different positions. The higher levels, λ through $L - 1$, are computed at a single central position shared for both eye positions. The technique of reprojecting short ray segments has been used by [7] to accelerate stereo volume rendering. Earlier work by [1] accelerated the computation of right-eye views by reprojecting individual samples from the rays of the left eye.

ACKNOWLEDGMENTS

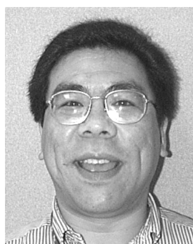
During the course of this work, we have benefited from helpful discussions with Lichan Hong, Bill Higgins, and Krishnan Ramaswamy.
Thanks to the National Library of Medicine for providing the Visible Human™ Dataset used in our measurements and images, and to CIeMed for the segmentation of the Visible Human ™ Dataset.

REFERENCES

[1] S. Adelson and C. Hansen, "Fast Stereoscopic Images with Ray-Traced Volume Rendering," *Proc. 1994 Symp. Volume Visualization*, pp. 3-10 and p. 125, Oct. 1994.
[2] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang, "VolVis: A Diversified Volume Visualization System," *Visualization '94*, pp. 31-38, Washington D.C., Oct. 1994. IEEE Computer Society Press.
[3] M.L. Brady, W.E. Higgins, K. Ramaswamy, and R. Srinivasan, "Interactive Navigation Inside 3D Radiological Images," *1995 IEEE Biomedical Visualization Symp.*, pp. 33-40 and p. 85, Oct. 1995.
[4] M.L. Brady, K. K. Jung, H. T. Nguyen, and T. Q. Nguyen, "Two-Phase Perspective Ray Casting for Interactive Volume Navigation," *Visualization '97*, pp. 183-189 and p. 541, Oct. 1997.
[5] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *Proc. 1994 Symp. Volume Visualization*, pp. 91-98 and p. 131, Oct. 1994.
[6] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics*, vol. 22, no. 4, pp. 65-74, Aug. 1988.
[7] T. He and A. Kaufman, "Fast Stereo Volume Rendering," *Proc. Visualization '96*, pp. 49-56, Oct. 1996.
[8] L. Hong, A. Kaufman, Y. Wei, A. Viswambharan, M. Wax, and Z. Liang, "3D Virtual Colonoscopy," *Proc. 1995 IEEE Biomedical Visualization Symp.*, pp. 26-32, Oct. 1995.
[9] L. Hong, S. Muraki, A. Kaufman, D. Bartz and T. He, "Virtual Voyage: Interactive Navigation In The Human Colon," *Computer Graphics*, pp. 27-34, Aug. 1997. ACM/SIGGRAPH Press.
[10] K. Kreeger, I. Bitter, F. Dachille, B. Chen, and A. Kaufman, "Adaptive Perspective Ray Casting," To appear in *Proc. Symp. Volume Visualization*, Research Triangle Park, N.C., Oct. 1998.

[11] M. Levoy, "Display of Surfaces From Volume Data," *IEEE Computer Graphics and Applications,* vol. 8, no. 3, pp. 29-37, May 1988.
[12] M. Levoy, "Volume Rendering by Adaptive Refinement," *Visual Computing,* vol. 6, no. 1, pp. 2-7, Feb. 1990.
[13] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Trans. Graphics,* vol. 9, no. 3, pp. 245-261, July 1990.
[14] W. Lorensen, "Marching Through the Visible Man," *Proc. Visualization '95*, Oct. 1995. IEEE CS Press.
[15] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics,* pp. 163-189, July 1987.
[16] W. Lorensen, F. Jolesz, and R. Kikinis, "The Exploration of Cross-Sectional Data with a Virtual Endoscope." R. Satava and K. Morgan, eds., *Interactive Technology and the New Medical Paradigm for Health Care*, pp. 221-230. Washington D.C.: IOS Press, 1995.
[17] K.L. Novins, F.X. Sillion, and D.P. Greenberg, "An Efficient Method for Volume Rendering Using Perspective Projection," *Computer Graphics,* vol. 24, no. 5, pp. 285-288, Nov. 1990.
[18] T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics,* vol. 18, no. 3, pp. 253-260, July 1984.
[19] K. Ramaswamy and W.E. Higgins, "Endoscopic Exploration and Measurement in 3D Radiological Images," *SPIE Medical Imaging 1996: Image Processing*, vol. 2,710, pp. 511-523, Feb. 1996.
[20] A. Van Gelder and K. Kim, "Direct Volume Rendering with Shading via Three-Dimensional Textures," *Proc. 1996 Symp. Volume Visualization*, pp. 23-30 and p. 98, Oct. 1996.
[21] D. Vining, D. Gelfand, and R. Bechtold, E. Scharling, E. Grishaw, and R. Shifrin, "Technical Feasibility of Colon Imaging with Helical CT and Virtual Reality," *Presented at the Annual Meeting of the American Roentgen Ray Society*, New Orleans, Apr. 1994.
[22] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics,* vol. 25, no. 4, pp. 275-283, July 1991.
[23] R. Yagel, D. Reed, A. Law, P.-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. 1996 Symp. Volume Visualization*, pp. 55-62 and p. 101, Oct. 1996.
[24] K. Zuiderveld, A. Koning, and M. Viergever, "Acceleration of Ray-Casting Using 3D Distance Transforms," *Proc. Visualization Biological Computing,* vol. 1,808, pp. 324-335, Chapel Hill, NC, Oct. 1992.

Martin L. Brady received the BS, MS, and PhD degrees in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1982, 1984, and 1987, respectively. He has worked as a research scientist at Lockheed's Palo Alto Research Lab, and as an assistant professor in the Department of Computer Science and Engineering at the Pennsylvania State University. Since 1996, he has worked as a research scientist in the Microcomputer Research Labs at Intel Corporation in Santa Clara, California. His research interests include volume visualization, computer graphics, parallel algorithms, and image processing.



Kenneth K. Jung received the MSEE from the University of Illinois at Urbana-Champaign in 1983, BSEE from Arizona State University in 1981. He has worked as a research scientist at Lockheed's Palo Alto Research Lab, and as an MTS at AT&T Bell Labs at Whippany, N.J. Since 1995, he has worked as a research scientist in the Microcomputer Research Labs at Intel Corporation in Santa Clara, California. His research interests include volume visualization, signal processing, and compression.

H.T. Nguyen received the BS in electrical engineering and MS in computer science from the University of Arkansas at Fayettville in 1981 and 1982, respectively. He has worked as a research scientist at Lockheed's Palo Alto Research Lab and Center for Information-Enhanced Medicine, National University of Singapore. He joined the Microcomputer Research Labs at Intel Corporation in Santa Clara, California as a research scientist in 1996. His research interests include volume visualization, three-dimensional user interface, biomedical imaging, and parallel algorithm and architecture.

Thinh PQ Nguyen received the BSc degree in computer engineering from the University of Washington in 1995. He is currently a member of the Microcomputer Research Labs at Intel Corporation. He has also worked in the Intel Development Lab and the Merced Processor group. His current interests include graphics, visualization, parallel computing, and networking.