Two-Phase Perspective Ray Casting for Interactive Volume Navigation

Martin Brady, Kenneth Jung, HT Nguyen, and Thinh Nguyen

Microcomputer Research Labs
Intel Corporation

ABSTRACT

Volume navigation is the interactive exploration of volume data sets by "flying" the view point through the data, producing a volume rendered view at each frame. In this paper, we present an inexpensive perspective volume navigation method designed to run on a PC platform with accelerated 3D graphics hardware. We compute perspective projections at each frame, allow trilinear interpolation of sample points, and render both gray scale and RGB volumes by volumetric compositing. Our implementation handles arbitrarily large volumes, by dynamically swapping data within the local depth-limited frustum into main memory as the viewpoint moves through the volume. We describe a new ray casting algorithm that takes advantage of the coherence inherent in adjacent frames to generate a sequence of *approximate* animated frames much faster than they could be computed individually. We also take advantage of the 3D graphics acceleration hardware to offload much of the alpha blending and resampling from the CPU.

Key Words: Volume navigation, volume rendering, 3D medical imaging, scientific visualization, texture mapping.

1 INTRODUCTION

Volume rendering [6] techniques can be used to create informative two dimensional rendered views from large 3D images (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 view of the entire volume. In dealing with large 3D data sets, this approach has several problems. First, it can take a prohibitively long time to render a single image. Interactive update of the point of view is thus precluded. This can be a significant disadvantage, since 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 [9]. In addition, although volumetric compositing techniques display some of the internal data via translucency effects, it can be difficult to discern small complicated internal structures within a large data set when generating an image from the entire volume.

In *volume navigation* [2], the viewing frustum is placed 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). Within medical data, for example, one could simulate an endoscopic examination of structures such as bronchial passages, arteries, or the intestinal tract using 3D radiological images [8,14]. Furthermore, this digital technique can be more flexible, allowing one to traverse complex branching structures, pass through solid objects, or render obscuring objects transparent.

2200 Mission College Blvd., Santa Clara, CA 95052
{Martin_Brady, Kenneth_K_Jung, H_T_Nguyen, Thinh_Q_Nguyen}@ccm.sc.intel.com

A key ingredient for volume navigation is the interactive volume rendering of a sequence of frames in real time. We assume that it is acceptable to generate these frames at reduced resolution, in order to enable interactive navigation. This allows one to quickly browse the data looking for interesting structures, and to take advantage of the depth information conveyed by an animated sequence of frames. Although useful information can 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. Note that most previous fast volume rendering algorithms have assumed parallel projection.

In this paper, we present an inexpensive perspective volume navigation method designed to run on a high-end PC platform with accelerated 3D graphics hardware. We compute perspective projections at each frame, allow trilinear interpolation of sample points, and render both gray scale and RGB volumes. Our implementation handles arbitrarily large volumes by dynamically swapping data within the local depth-limited frustum into main memory as the viewpoint moves through the volume. We describe a new ray casting algorithm that takes advantage of the coherence inherent in adjacent frames to generate a sequence of *approximate* animated frames much faster than they could be computed individually. We also take advantage of the 3D graphics acceleration hardware to offload much of the alpha blending and resampling from the CPU.

In Section 2, we describe previous work in fast volume rendering and navigation. In Section 3 we present our new rendering algorithm. Section 4 describes our implementation and the results obtained. Finally, Section 5 contains some conclusions.

2 PREVIOUS WORK

A number of different methods have been proposed for interactively exploring volume data. One method is to convert the volume data into a surface-based representation using an isosurface extraction algorithm such as Marching Cubes [10,11]. This preprocessing step is time consuming, taking up to a few minutes, 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 though data containing fairly well defined surfaces, such as the colon wall within 3D radiological images [12,16]. The disadvantage of this method is that much of the information contained within the 3D data set is lost in the conversion to isosurfaces. Structures with densities that are not near the isosurface value are not visible. This limits the ability to fully explore a 3D data set by navigating through it.

Brady, et al., [4] proposed a method for volume navigation that takes advantage of the inter-frame coherence by rendering a dense set of short parallel ray segments, which are then used to construct entire rays from many neighboring points of view. 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 sufficient, 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. Useful information can be obtained by browsing the data in this way, but the view is highly distorted, and does not look at all like a 'realistic' walk through the volume data from within.

We use the general idea of computing multiple views from stored volume rendered ray segments, but extend this idea to render perspective views. This requires a significantly different approach, since the previous method relies on the fact that stored ray segments are parallel. In addition, we show how sets of ray segments can be grouped and defined as 2D textures in order to allow a standard 3D graphics accelerator to perform the final phase of the rendering. A number of previous researchers have proposed various techniques to take advantage of existing 2D or 3D texture mapping hardware to accelerate volume rendering [5,15,17,18].

3 FAST PERSPECTIVE PROJECTION VOLUME NAVIGATION

3.1 Overview

In our navigation scenario, we assume a uniform volume-sampled 3D data set of size $N_1 \times N_2 \times N_3$. The user can travel through the volume following an arbitrary navigation path. At any given time, only a small subregion of the volume of size $n_1 \times n_2 \times n_3$ is used to produce the current view. (In a highly transparent region, limiting the depth of the view may cause rays to clip before they have saturated. This assumption could be enforced by incorporating an attenuation term to limit the depth of the view.) We assume that the entire volume is too large to fit into main memory, but that the working subregion is small enough to fit. Thus, a dynamic I/O technique is implemented to incrementally update the subregion with data from disk as we navigate.

Although the effective viewing volume for a single frame is limited, it is typically still too large to be rendered at interactive rates by conventional methods. However, in successive steps, view position or direction can be changed only by a small increment. 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. We use this coherence between nearby frames to accelerate their computation. In particular, intermediate results from the computation of a given view are used to quickly compute f successive frames, after which a new, fresh set of intermediate results must be produced. By amortizing this work over the f frames generated, an approximately factor f speedup is obtained in the core rendering portion of the computation. Below we describe our new two-phase perspective ray casting algorithm which is the heart of the method. We then explain how to incorporate it into volume navigation. Finally, we describe some extensions to the technique.

3.2 Two-Phase Perspective Ray Casting Algorithm

Perspective ray casting algorithms often cast a ray at a time, iteratively computing the next sample location, sampling, and compositing the sample onto the current ray. One ray is cast for each pixel in the rendered image. Thus, to render an $m \times m$ view at a depth of d samples per ray requires m^2d sampling steps. Our algorithm is divided instead into two phases. In the first phase, we cast a set of short ray *segments*, 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.

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. We assume for simplicity in the following discussion that the sample rate along the rays is 1 (but in fact this rate can be arbitrarily specified). 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 $S_l(p,v)$, or simply as S_l when position and direction are understood. The set of all segments at all levels is denoted $S(p,v)$, or simply S, respectively.

If we choose to sample each level at the screen size, then Phase 1 results in a set of L planes of segments of size $m \times m$. We then simply blend these planes to form the final view in Phase 2. This would amount to a simple reordering of the ray sample and composite operations from the ray-at-a-time algorithm, and would produce the same output, using the same number of sampling steps. In general, however, we allow each level, l, to be sampled at different rates, casting $W_l \times H_l$ ray segments in level l. Phase 2 then resamples each level to screen resolution and then composits the results to form the final view. (Alternately, the overall resampling work could be reduced by iteratively resampling level l to $W_{l+1} \times H_{l+1}$ and then compositing with level l+1. This turns out to be unimportant, since the resampling step will be performed off the CPU, on the graphics accelerator hardware in our implementation.) Algorithm 1 below gives a high level overview of this technique.

Algorithm 1. Two-Phase Perspective Ray Casting.

```
/* Phase 1 */
for l from 0 to L–1
   /* Generate a W_l × H_l array of segments at level l. */
   for depth from D_l to D_{l+1}–1
      Sample an array of W_l × H_l points, at distance depth.
      Composite the array onto the back of the set of level l
            segments, S_l.
   end for
end for
/* Phase 2 */
for l from 0 to L–1
   Resample segments S_l at screen resolution.
   Composite onto the back of current view.
end for
```

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. Observe that in a standard ray cast, the lateral sampling rates vary in proportion to the distance from the viewpoint. Novins, et al. [13], proposed a somewhat similar adaptive sampling approach, with the primary objective of avoiding undersampling of distant sample points by splitting rays

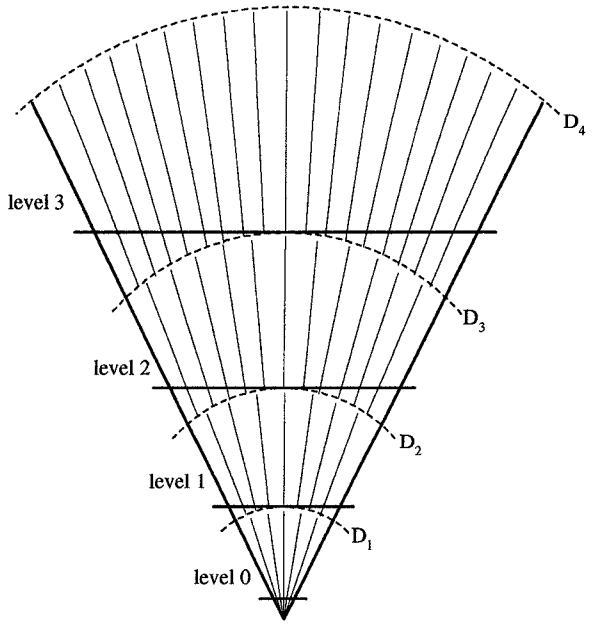

Figure 1. 2D illustration of the segments computed by the two-phase algorithm. Solid horizontal lines represent the rectangles that the segments are to be mapped into.

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.

In our use of the two-phase algorithm, we generally set the resolution of each level so that the lateral sampling rates are similar at all levels. A level's dimensions, $W_l \times H_l$, are then proportional to its distance from the viewpoint, D_l. This means that the first levels, being near the viewpoint, may cast very few rays. The result is some reduction in total sampling time, although more important for volume navigation is the fact that there is a large reduction in the sampling time for ray segments near the viewpoint (and little change for the most distant segments).

The two-phase algorithm can be naturally partitioned between the CPU and 3D graphics accelerator hardware, as described below. In essence, Phase 1 is performed on the CPU, and Phase 2 can be done entirely on the accelerator. Each level, l, of rendered segments from Phase 1 is defined as a 2D texture, T_l, in Phase 2. For each level, we draw a rectangle R_l at distance D_l, perpendicular to the view direction, and map texture T_l into the rectangle (see Figure 1). (Level 0 is an exception, since $D_0 = 0$. We choose to draw its rectangle at distance 1, although anywhere between 0 and D_1 will do.) 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. One can balance the time spent by the CPU and the accelerator by adjusting the number of levels.

Note that this scheme requires only 2D texture mapping, and is thus amenable to PC-based workstations with commercially available graphics accelerators. In contrast, many of the previous texture mapped volume rendering methods have required 3D texture mapping hardware, and most have been dependent upon or skewed toward a parallel rendering geometry.

3.3 Interactive Navigation

The 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 original position. We use them for viewpoints within a radius δ of the original 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 $S_l(p,v)$, $\lambda \le l < L$, we quickly compute an approximate view within (δ,θ) of (p,v).

A straightforward implementation of this idea would be to initialize the $S_l(p,v)$ data structure from the current position. 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 we will require the update. This is true for long sweeps of translation in a fixed direction or rotation about a fixed axis. (This is precisely the case in which smooth motion is most important. If one is changing between translation and rotation, it generally requires a change in user input, and during this time a short delay (~1 sec.) in updating the data structure is more tolerable.)

3.3.1 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 pseudo code overview of the forward translation algorithm is shown below in Algorithm 2 and illustrated in Figure 2. Notice that while the positions of the first λ levels move about with the viewpoint, the last $L-\lambda$ 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, 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 (p,v), rays on 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 border of $\delta \tan(\phi/2)$ units on each side.

Algorithm 2 provides nearly a factor of f speedup Phase 1 of the algorithm over rendering each frame independently. Higher values of f increase the error in the approximation, however, and this error is difficult to characterize. Consider a segment s in level l that is to be approximated at position $p+offset$ by

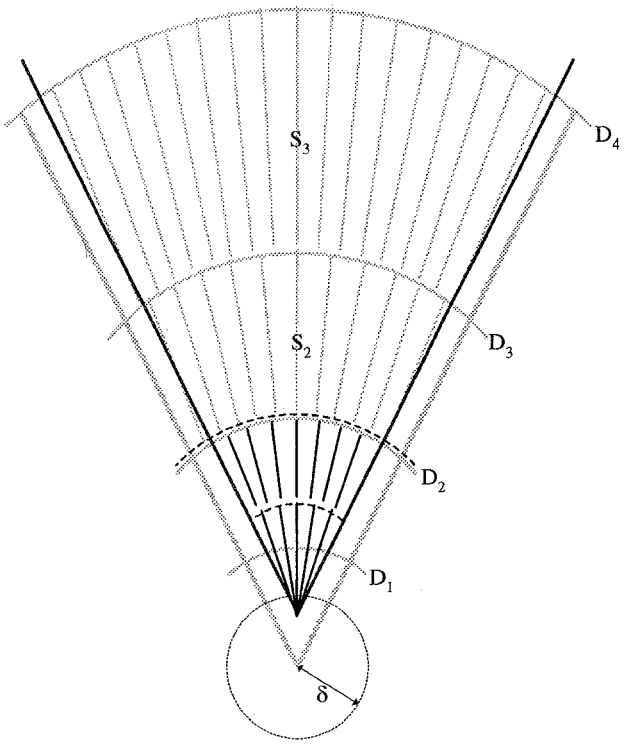

Figure 2. 2D depiction of forward translation. Gray lines represent the view from the original position, while black indicate the position of the translated view. In this example, $\lambda = 2$; i.e., the first two levels are recomputed (the recomputed segments are depicted in black), while the last two 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.

resampling $S_l(p,v)$ (see Figure 3). Ideally, we would resample by determining a weight for segments near 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 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 non-linear 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. Towards 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.

3.3.2 Viewpoint Rotation

For rotation, we must begin with a data structure $S(p,v)$, 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

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 $S_l(p+2\delta,v)$, $\lambda < l < L$.
if *offset* $> \delta$
  for l **from** $\lambda-2$ **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

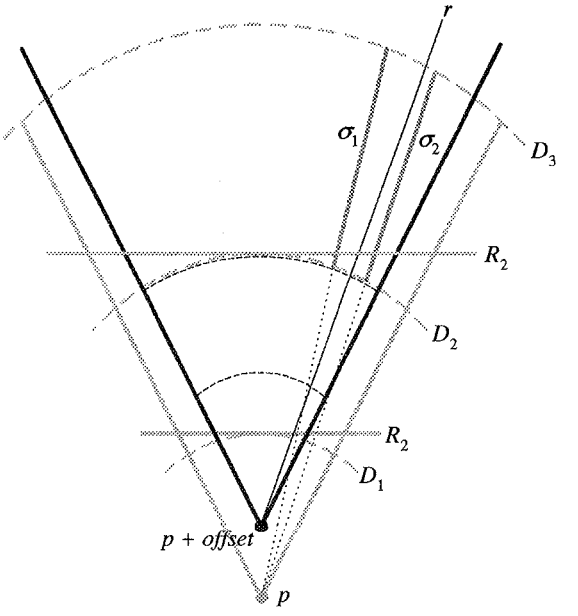

Figure 3. A 2D illustration of the resampling of S during translation. A ray r cast from position *p+offset* (black) is resampled in level 2 using segments σ_1 and σ_2 cast from position p (gray). The (bi)linear interpolation is based on the relative positions of r, σ_1, and σ_2 projected into R_2.

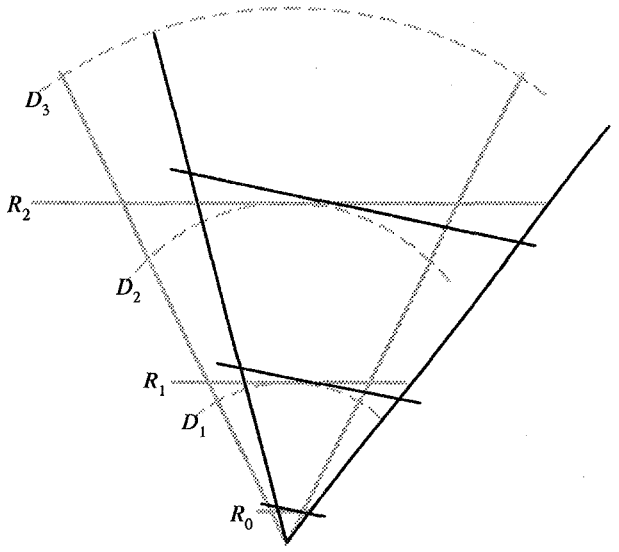

Figure 4. The initial segment data structure is depicted in gray, and should be 2θ wider than the field of view. The next segment data structure for right rotation is depicted in black.

step, we compute $1/f$ of the next data structure, $S(p,v+f\Delta v)$. This is illustrated in Figure 4, and outlined in Algorithm 3. Notice that we do not need to recompute levels 0 through $\lambda-1$ at each step, assuming that their field of view is increased by $f\Delta v/2$.

The accuracy of resampling is less an issue for rotations. Given that the point of view remains at p during rotation, the orientation of the segments at the new rotation are consistent with those in the data structure, and so relative distance between the segments is well defined, and is handled properly by the resampling on the projected rectangle.

3.4 Extensions and Enhancements

There are several possible extensions to enhance this interactive volume navigation technique. We describe these briefly below.

The fast approximation technique that we are using assumes that all frames are viewed for no more than $1/10^{th}$ of a second, and therefore need not be of extremely high quality. However, during a pause in motion, lower quality views become more noticeable. However, this idle time can be used to compute and display incremental improvements to the original view (see, e.g., [3] for progressive refinement techniques). 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. (This could be done in several sweeps, in which an intermediate improvement is computed in the first sweep, then a higher quality, etc.)

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 which maintains the current volume region in RAM to obtain more distant samples. However, after a band of data near the current

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+2θ), λ < l < L
if offset > θ
   Recompute levels 0 through λ–1 from orientation v+2θ
      at position p.
   for l from λ to L–1
      Swap S_l(p,v+2θ) in for S_l.
   end for
   offset = –θ
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
```

depth is composited, it can be deleted to make room for the next band.

It is inexpensive to create stereo pairs using the current 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 left eye's rays.

4 EVALUATION OF THE ALGORITHM

4.1 Implementation

Our implementation consists of four major blocks which are performed to compute each frame of output:

Recast. Recast the first λ levels of segments.

Update. Incrementally update $1/f$ of the predicted future levels λ through $L-1$.

Display. Resample and blend each level at the screen resolution.

I/O. If necessary, read slice(s) of raw data from disk and insert into the active cube of data.

Recast and *Update* are performed on the CPU, and consist primarily of trilinear resampling, lighting, and blending calculations to construct ray cast segments. Opacity is obtained by lookup on the luminance of the sample data. Lighting calculations include ambient and diffuse terms. The computation was simplified in our implementation by assuming a point source located at the viewpoint (i.e., a "miner's helmet" that moves and rotates with the viewer). The gradient computation was reduced to taking a single difference along the ray and scaling the diffuse factor by this term.

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

I/O is more difficult to characterize. Initially, the time is dominated by reads and seeks from the hard disk. In this case, *I/O* dominates the other three blocks. However, the operating system performs file caching, and once this data has been traversed, *I/O* accesses hit the RAM and the access times become relatively small, except for occasional misses, until one moves into a new region of the data file. For our measurements, the cache was "warmed" by traversing the data before taking measurements.

The implementation is dependent on numerous parameters, including L (number of levels), depth of each level, sample rate for each level, field of view, λ (number of levels that are recomputed at every step), f (number of steps between recomputation of the remaining levels), and m (screen size). We selected a set of parameters that seemed a reasonable balance between frame rate and image quality, but due to the large search space, better settings undoubtedly exist. The sampling rates for each level were set sufficiently high so that lateral distance between samples was always less than one unit. The parameters used are given in Table 1.

Table 1. List of parameters used for volume navigation implementation.

$L = 8$
field of view = 30°
$\lambda = 3$
$f = 16$
$m = 512$

Level	start depth (D_l)	$W_l = H_l$
0	0	8
1	8	16
2	16	25
3	32	40
4	64	50
5	80	60
6	96	70
7	112 (end depth = 127)	80

The total number of voxels enclosed by the view frustum is approximately 200,000. The total number of trilinearly-interpolated samples used to create the initial display and data structure is 342,160. We measured times on both 8-bit gray scale data (using both color and opacity look up tables) and 24-bit RGB data (requiring an opacity table, indexed by luminance computed from the sampled RGB value on the fly).

4.2 Performance Measurement

The volume navigation technique was implemented and evaluated on single and quad-CPU platforms. Single processor measurements were made on an Intel 300 MHz Pentium® II processor system. The platform was equipped with 128 MB RAM and an Intergraph Intense 3D Pro graphics accelerator card. The multiprocessor measurements were taken on an Intergraph TDZ-610 platform, containing four 200 MHz Intel Pentium® Pro processors. The platform was equipped with 256 MB RAM, and Intergraph RealiZm Z-25 3D graphics acceleration hardware with 64 MB of texture memory. Both machines ran the Windows® NT 4.0 operating system.

Table 2 contains the main timing results for the algorithm. The code was written as five threads which synchronize in each frame. Two threads handled the *Update* block in parallel, one handled *Recast*, one *Display*, and one *I/O*. We compare the single-CPU implementation and the multiprocessor versions, on both 8-bit gray and 24-bit RGB data. Translation and Rotation represent the average time to compute a single frame using the segment data structure. The time to recompute the view without the benefit of the data structure using our two-phase algorithm is given for comparison. Finally, we show the time for a standard "brute force" front to back ray cast, in which the number of rays is held constant at 80 by 80, the same as the resolution at the last level of our algorithm. This was done by setting our algorithm parameters for a single level, $W_0 = H_0 = 80$, which is recast at every frame. Thus, the single Recast thread dominates the computation, and effectively runs on just one CPU.

Table 2. Times for incremental forward step, incremental rotational step, and time to recompute the entire view from scratch using Algorithm 1. This is compared to the time for a "brute force" perspective ray cast, which cast all rays at the highest resolution used in Algorithm 1. (* The brute force ray cast is dominated by a single thread, and thus effectively runs on a single CPU of the Intergraph platform in this implementation.)

	Pentium® II 300 MHz		Quad Pentium® Pro 200 MHz	
	RGB (msec)	Gray (msec)	RGB (msec)	Gray (msec)
Translation	275	207	101	92
Rotation	290	210	120	97
Recompute	1356	696	782	392
Brute Force	7521	3435	11950*	5957*

For a more detailed breakdown of the times required for each of the major subroutines, we measured the average time per frame for each of the four major blocks, as shown in Table 3, for both RGB and 8-bit gray scale data. These times were taken on both the Pentium® II and on a single CPU of the Intergraph platform, using a single threaded version of the code. Note that much of the display computation is performed on the graphics accelerator, and can be overlapped with CPU computation, even on a single processor platform.

Table 3. Average times, in milliseconds, for the major blocks of the algorithm. This times were measured on a single CPU on the Pentium® II and Intergraph platforms, by running a single threaded version of the code.

	Pentium® II 300 MHz		Pentium® Pro 200 MHz	
	RGB (msec)	Gray (msec)	RGB (msec)	Gray (msec)
Recast	40	18	64	31
Update	85	53	131	80
Display	150	150	71	71

4.3 Images

We present for comparison images produced by our algorithm, shown in the color plate in Figures 5 and 6. The pictures were generated using the Male Visible Human Dataset™, courtesy of the National Library of Medicine. The original data is 24-bit RGB, sampled at .33mm by .33 mm by 1mm. It has been downsampled to 1mm by 1mm by 1mm in our examples. We use the rendering parameters given in Section 4.1.

Figure 5 shows a position within the superior vena cava. Figure 5a is a standard perspective ray cast of 80 by 80 rays, the

control image that we compare to our approximate views. Figure 5b shows a rendering using our two-level algorithm at the same position. In this case, the view point is at the center of the radius δ range, i.e., the offset is zero. Figure 5c shows a rendering at the same position, but with the maximum possible offset (8) for the set of segments. This is a worst case position for our approximation.

Figures 5d, 5e, and 5f are difference images. In each case, the magnitude of red, green, and blue channel differences is taken, and multiplied by a factor 10 in order to make the values visible. Figure 5d depicts the difference between the standard (5a) and two-phase (5b) images, i.e., the error due to the adaptive sampling of the image. Note that much of the error occurs at nearby structures within the image, where the sampling resolution was lowest. Figure 5e gives the difference between the two-phase algorithm at zero offset (5b) and the worst-case offset (5c), i.e., the error due to re-using the segment data structure for views at a distance δ from the position from which the segments were cast. The errors are more widely dispersed in this case. Figure 5f displays combination of these errors, the difference between Figures 5a and 5c.

A second example, with higher-contrast edges was used in Figure 6. The data set was segmented to remove all tissue except bone. The rendering is of the thoracic vertebrae. Figure 6a shows the standard perspective ray cast, Figure 6b is a rendering using the two-phase algorithm, and the difference is shown in Figure 6c multiplied by 5.

5 CONCLUSIONS

Our algorithm provides a unique balance between speed and quality for interactive volume rendering by taking advantage of inter frame coherence. Our implementation is by no means optimized, but serves to verify the promise of the technique. 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.

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 all of 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," *1994 Symp. On 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] L. Bergman, H. Fuchs, E. Grant, S. Spach, "Image Rendering by Adaptive Refinement," *Computer Graphics* **20** (4), pp. 29-37, Aug. 1986.

[4] 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.

[5] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *1994 Symp. On Volume Visualization*, pp. 91-98 and p. 131, Oct. 1994.

[6] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics* **22** (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," *1995 IEEE Biomedical Visualization Symp.*, pp. 26-32, Oct. 1995.

[9] D. Laur and P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *Computer Graphics* **25** (4), pp. 285-288, July 1991.

[10] W. Lorensen, "Marching Through the Visible Man," *Proc. of Visualization '95*, IEEE Press, Oct. 1995.

[11] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, pp. 163-189, July 1987.

[12] W. Lorensen, F. Jolesz, and R. Kikinis, "The Exploration of Cross-Sectional Data with a Virtual Endoscope." In R. Satava and K. Morgan (eds.), Interactive Technology and the New Medical Paradigm for Health Care, pp. 221-230. IOS Press, Washington D. C., 1995.

[13] K. L. Novins, F. X. Sillion, and D. P. Greenberg, "An Efficient Method for Volume Rendering Using Perspective Projection," *Computer Graphics* **24** (5), pp. 285-288, Nov. 1990.

[14] K. Ramaswamy and W. E. Higgins, "Endoscopic Exploration and Measurement in 3D Radiological Images," *SPIE Medical Imaging 1996: Image Processing*, vol. 2710, pp. 511-523, Feb. 1996.

[15] A. Van Gelder and K. Kim, "Direct Volume Rendering with Shading via Three-Dimensional Textures," *Proc. 1996 Symp. on Volume Visualization*, pp. 23-30 and p. 98, Oct. 1996.

[16] 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.

[17] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics* **25** (4), pp. 275-283, July 1991.

[18] 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. on Volume Visualization*, pp. 55-62 and p. 101, Oct. 1996.