Shear-Image Order Ray Casting Volume Rendering

Yin Wu[†], Vishal Bhatia[‡], Hugh Lauer[†]
TeraRecon, Inc.

Larry Seiler[‡]
Mitsubishi Electric Research Laboratories

Abstract

This paper describes shear-image order ray casting, a new method for volume rendering. This method renders sampled data in three dimensions with image quality equivalent to the best of ray-per-pixel volume rendering algorithms (full image order), while at the same time retaining computational complexity and spatial coherence near to that of the fastest known algorithm (shear-warp). In shear-image order, as in shear-warp, the volume data set is resampled along slices parallel to a face of the volume. Unlike shear-warp, but like the texture-based methods, rays are cast through the centers of pixels of the image plane and sample points are at the intersections of rays with each slice. As a result, no post-warp step is required. Unlike texture methods, which realize shear and warp by transformations in a commodity graphics system, the shear-image ray casting methods use a new factorization that preserves memory and interpolation efficiency. In addition, a method is provided for accurately and efficiently embedding conventional polygon graphics and other objects into volumes. Both opaque and translucent polygons are supported.

We also describe a method, included in *shear-image* order but applicable to other algorithms, for rendering anisotropic and sheared volume data sets directly with correct lighting.

The *shear-image* order method has been implemented in the VolumePro™ 1000, a single chip real-time volume rendering engine capable of processing volume data at a pipeline rate of 10^9 samples per second. Figure 1 on the color page shows *a shear-image order* gallery of volumes rendered with different translucency, lighting, and some embedded geometry.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; I.4.10 [Image Processing and Computer Vision]: Image Representation.

Additional Keywords: Volume rendering; ray casting; shear warp; base plane; shear-*image* order; image order.

1 INTRODUCTION

Real-time volume rendering is a technique for creating interactive images of objects and phenomena represented as sampled data in three or more dimensions [7]. It is becoming increasingly important in medical imaging, seismic exploration, and scientific visualization, and it has potential applications in industrial inspection, non-destructive testing, airline security, and any area where it is important to see the internal or hidden structures of the objects under study. While volume rendering algorithms have been known for years, there are three principal challenges to achieving useful, interactive visualization: amassing enough computational power to render images at multiple frames per second; moving huge amounts of data from memory to the processing power; and providing high quality, visually meaningful images.

In this paper, we introduce shear-image order volume rendering. Shear-image order inherits the ray-per-pixel feature of full image order; it is shear order as the x and y-axes of the sample space are sheared from the z-axis of image space (hence the name shear-image order). The shear-image order method eliminates shear-warp's intermediate image and final warp step while preserving the memory access efficiency of shear-warp. It produces high quality images by casting rays directly through the centers of pixels of the image plane. Shear-image order is efficient to compute, requiring four interpolations per sample compared with three interpolations per sample for shear-warp and seven interpolations per sample for full image order. Shear-image order supports the accurate embedding of polygon and other objects, and it enables direct rendering of anisotropic and sheared data sets without the need for resampling. The shear-image order method is implemented in VolumePro™ 1000, a second-generation real-time volume rendering engine.

Section 2 of this paper briefly describes existing volume rendering algorithms — *shear-warp, full image order*, and texture based volume rendering. Section 3 describes the *shear-image order* method, including its coordinate systems and transformations, the algorithm for stepping through the sample points in parallel with volume data for high performance and discussions relating it to different volume rendering methods. Section 4 describes how polygons and other objects can be accurately embedded in the rendered image of the volume. Section 5 discusses anisotropic and sheared volumes, which are common in the medical imaging field. Section 6 presents an overview of the VolumePro 1000 hardware implementation. Section 7 presents conclusions and discusses future work.

2 BACKGROUND

This section provides a brief review of three important algorithms for volume rendering — *shear-warp, full image order*, and texture based volume rendering.

2.1 Shear-Warp Order

One of the fastest classic algorithms for volume rendering is *shear-warp* [6]. In *shear-warp*, the 3D viewing matrix is factored into "a 3D shear parallel to slices of the volume data, a projection to form a distorted intermediate image, and a 2D warp to produce the final image" [6]. Shear-warp has the advantage of retrieving volume data from memory in a coherent manner, thereby maxi-

[†] Address of Yin Wu and Hugh C. Lauer: TeraRecon, Inc. 300 Baker Avenue, Suite 301, Concord, MA 01742. E-mail: wu@terarecon.com, lauer@terarecon.com, World-wide web: http://www.terarecon.com/products/volumepro_prod.html

[‡] Current address of Vishal Bhatia and Larry Seiler: ATI Research, Inc., 62 Forest Street, Marlboro, MA 01752. E-mail: vbhatia@ati.com, lseiler@ati.com.

mizing the utilization of memory bandwidth. It has the disadvantages of requiring a final warp step and has an additional difficulty of accurately embedding polygons and images of other objects. VolumePro™ 500, the first commercial real-time volume-rendering engine, is a hardware implementation of shear-warp with flexibility in the sampling frequency in the *z*-dimension [9].

Figure 2 illustrates the shear-warp factorization. Voxel positions are shown as dots. The × characters represent sample points of the volume. The pixel locations are the intersections of the gray grid of Figure 2d. In Figure 2a, the axes of the volume data set are permuted so that the z-axis is most nearly parallel to and in the same direction as the rays. The volume is re-sampled along the rays with sample points organized in slices perpendicular to this permuted z-axis, contributing to the memory coherency. Both shear-image order and 2D texture methods share this feature.

Within each slice, samples are organized in grids parallel to the grid of voxels. The volume is transformed by shearing each slice with respect to its neighbors (Figure 2b). Samples are then projected onto a plane parallel to those slices called the *base plane* (Figure 2c). The resulting image is then warped to produce the final image as shows in Figure 2d.

The shear-warp factorization allows the rendering engine to fetch and process volume data in a sequence related to its storage in memory, thereby using both memory bandwidth and recently retrieved voxel data efficiently. In addition, interpolation operations can be shared among adjacent samples in the same slice: four neighbors in each slice share one interpolated value in the permuted *z* direction, and two neighbors share an interpolated value in a second dimension within the slice. Therefore, *shear-warp* method realizes tri-linear interpolation with three multiplications per sample, one multiplication per dimension.

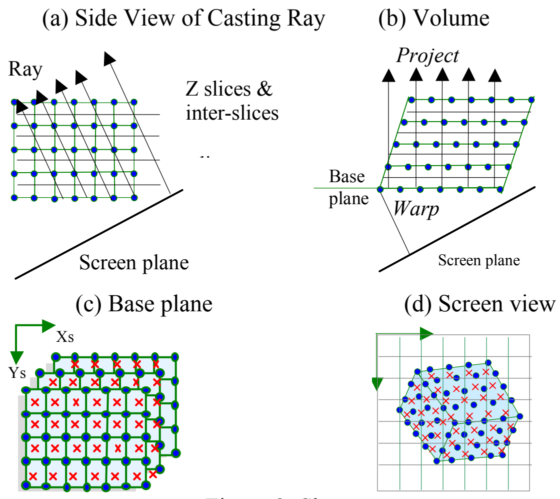

Figure 2: Shear-warp

It is possible to achieve high quality images using shear-warp. However, doing so requires over-sampling the volume data set to compensate for distortion in the warp step and high precision calculation to reduce error propagation between the sampling stages. These requirements impact performance. Furthermore, the mismatch between pixels in the base plane and the image plain makes it difficult to embed polygons into rendered volumes.

2.2 Full Image Order

Another class of ray-casting methods is often called image order, but is referred in this paper as full image order in order to emphasis the difference from shear-image order. In full image order methods, rays are cast through the centers of pixels of the image plane (Figure 3b) and samples are organized into planes parallel to the image plane (Figure 3a). These methods eliminate the need for the final warp step of shear-warp methods, and they can produce high quality images without over-sampling the volume. However, the cost is increased complexity in data handling and buffering and the loss of coherent patterns of memory accesses. Moreover, interpolations cannot, in general, be shared between adjacent samples so seven multiplications are needed to derive sample values by tri-linear interpolation (four in the 1st interpolation dimension, two in the 2nd dimension and one in the 3rd dimension). As a result, full image order methods are not yet competitive in performance with shear-warp methods, even using hardware acceleration.

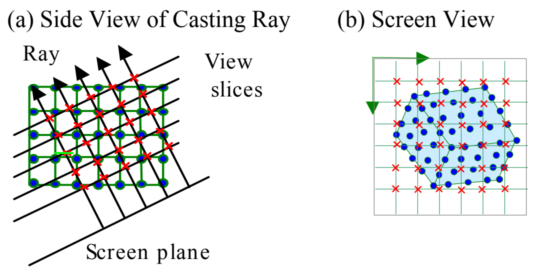

Figure 3: Full Image Order

2.3 Texture Map Methods

Texture map methods have been widely discussed in the literature (see [1], [11]). These can be partitioned into 3D texture methods and 2D texture methods. 3D texture methods are conceptually simple and much like full image order. The volume data set is stored in the 3D texture memory of a conventional polygon graphics system. To render the volume from a particular point of view, a set of polygons collectively called the *proxy geometry* is defined to represent slices parallel to the image plane. These polygons are rasterized and texture mapped by the 3D texture to produce sample values. These samples are then accumulated in a frame buffer using ordinary alpha blending.

This is equivalent to ray casting where all of the rays are advanced in parallel, one slice of proxy geometry at a time. The fragments of each slice are derived from the texture map by tri-linear interpolation, using the interpolation machinery that is inherently part of 3D texture map systems. Each sample point is naturally aligned with a pixel on the image plane, so no further processing is required. It is also easy to embed polygonal objects by simply drawing them along with the proxy geometry.

This method can produce high quality images if the spacing of the slices of proxy geometry is sufficiently close or if pre-integration is used [3]. However, there is little memory coherency in the interpolation process, so performance can be challenging. In addition, there is no natural way for the algorithm to determine whether any particular ray can to be terminated early, for example because it has already reached maximum opacity.

There are two major impediments that prevent 3D texture mapping on commodity graphics chips from being the preferred method of volume rendering. First, modern graphics chips require very specialized, high bandwidth memory devices. Currently, these come in very small sizes — 128 megabits — and cannot be cascaded because of timing demands. Therefore, these graphics chips cannot support the large data sets usually found in medical and geophysical applications, e.g., one gigabyte or more. The second impediment is the estimation of gradients for illumination.

2D texture methods for volume rendering are more complex than 3D texture methods, but they are useful on graphics hardware that has only 2D texture support. Typically, a volume data set is partitioned into slices three times, once for each axis. Then each slice from each dimension is loaded into a separate 2D texture map. To render a volume, proxy geometry is generated once for each slice, rasterized, textured by the map corresponding to that slice and orientation, then accumulated by blending in a frame buffer. The 2D texture methods suffer the same limitations as 3D texture methods. In addition, each volume data set needs to be stored three times, increasing the total memory requirement.

3 SHEAR-IMAGE RAY CASTING

Shear-image ray casting preserves shear-warp's organization of sample points in slices parallel to the slices of the volume, but it casts rays directly through the centers of pixels of the image plane, as in *full image order*. It thereby eliminates the intermediate image and final warp step of shear-warp. It is similar to 2D texture methods in that the rays are cast from the screen plane onto slices parallel to a face of the volume. However, the *shear-image* method explicitly decomposes the 3D viewing transformation into two matrices: a transformation from voxel coordinates to an intermediate coordinate system called *sample space* that is aligned with the pixels of the image or screen; and a transformation to adjust depth values of sample points to reflect their distances from the image plane or other reference point. The explicit decomposition has four beneficial features: (1) sample space is spatially coherent with both image or screen coordinates and with the voxel coordinates; (2) intermediate samples are cast directly from screen plane and can therefore blended without additional error propagation and image distortion; (3) the depth warp allows the embedding of polygons by comparing depths of samples with those of fragments of the polygons; (4) and super sampling factors can be controlled in a very flexible way, allowing in equal sample spacing in three dimensions for any view angle.

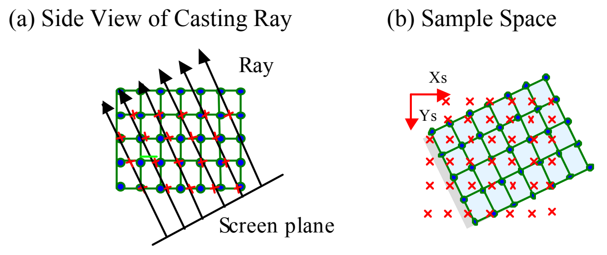

Figure 4: Shear-Image

In Figure 4, sample points (×) are organized into slices parallel to a face of the volume (i.e., normal to the permuted z axis), as shown in Figure 4a, similar to Figure 2a. Samples within each slice are located at points where the rays passing through pixels on the image plane intersect the slice. Figure 4b shows the sample grid x_s and y_s are not parallel to the permuted voxel grid x_v and y_v. Indeed, the shear-image method has the same screen view image as full image order (Figure 3b): the sample grid is aligned with the screen pixels. However, due to the shear between screen and sample space in the permuted z dimension, a separate calculation is needed to obtain the depth of each sample.

The memory coherence of shear-image method also applies to gradient interpolation. Gradients are estimated at voxel points by central difference or another convolution. Gradients at sample points are derived by tri-linear interpolation, in parallel with deriving sample values. Our method considers the gradient calculation for anisotropic data, which is described further in Section 5.

The *shear-image* algorithm operates in two parts. The first part operates like *shear-warp* by stepping through volume memory one slice at a time. It interpolates between adjacent voxel slices to obtain slices of what we call *z-interpolated samples* — that is, sample values resulting from interpolation only in the z-dimension of the volume. Since there has not yet been any interpolation in the other two dimensions, these z-interpolated samples are organized in a grid parallel to the x- and y-dimensions of the volume.

The second part of the algorithm steps through each slice of z-interpolated samples in the x- and y-dimensions of the sample space. It enumerates the sample points (i.e., the intersections of rays with the slice), and it derives color and opacity values for each one from the z-interpolated samples of that slice. These samples are accumulated along their respective rays, thereby producing rendering of the volume directly on the image plane.

The algorithm also associates with each sample point a *depth value* to measure the distance from the sample point to the eye or some other reference. These depth values correspond to the z-values of traditional polygon graphics and are used for embedding polygons in the rendered image as described in Section 4.

3.1 Coordinate Systems and terms

To precisely define *shear-image* order, we illustrate various coordinate systems and transformations as well as the terms used in this paper. These are illustrated in Figure 5.

World space, *Camera* or *Eye space*, *Normalized Device coordinates*, and *Image space* are coordinate systems commonly understood in computer graphics [4]. Likewise, an application specifies the position, orientation, scale, and view of the volume object to be rendered using the *Model*, *View*, *Projection*, and *Viewport* transformations of the underlying graphics environment.

For *shear-image order* rendering, we introduce *voxel space* as the coordinate system of the 3D array of voxel data; voxels are at integer positions in this space, regardless of the coordinate system in which the data was obtained. *Object space* is a 3D Carte-

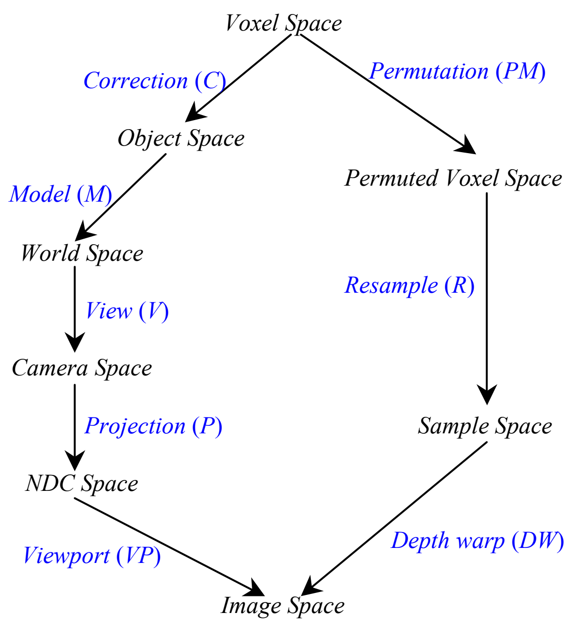

Figure 5: The coordinate systems of shear-image order rendering

sian coordinate system of the volume based on a local origin. The *Correction* transformation maps voxel space to object space, correcting for anisotropy and shear in the sampling process. *Permuted voxel space* is a permutation of voxel space in which the axes are rearranged so that the *z*-axis is most nearly parallel to the rays and in the same direction. It is introduced primarily to avoid singularities in solving for the unknowns in the following sections. *Sample space* is the coordinate system of sample points; each sample is located at an integer position in sample space.

In this paper, the axes of permuted voxel space are labeled x_v, y_v, and z_v. The axes of image space are denoted in this paper by x_i, y_i, and D (for depth). The axes of sample space are labeled x_s, y_s, and z_s. A crucial identity is that $x_s = x_i$, and $y_s = y_i$. The origin of sample space corresponds to ($x_i = 0, y_i = 0, D = depth0$), where *depth0* is the depth value of the near plane of the viewport.

Figure 5 shows two hierarchies of transformations mapping voxel space to image space. The Correction transformation and the traditional transformations of computer graphics are on the left. The three transformations of shear-image order rendering are on the right, namely, the *Permutation* transformation for rearranging the axes, the *Resampling* transformation for mapping permuted voxel coordinates to sample space, and the *Depth* transformation for mapping the z_s-coordinate of sample space to the *D*-axis of image space.

3.2 Derivation of rendering parameters

When an application requests the rendering of a volume by the *shear-image order* method, it must specify the *Correction* transformation, the graphics transformations on the left side of Figure 5, *depth0*, and the spacing of sample points along rays.[1] From Figure 5, it can be seen that

$$VP \times P \times V \times M \times C = DW \times R \times PM. \qquad (1)$$

That is, a mapping of a point or vector in voxel space by the conventional graphics transformations, augmented by the Correction transformation, must produce the same result as the transformations implemented by the *shear-image* order engine.

Let $M_p = VP \times P \times V \times M \times C \times (PM)^{-1}$. Observe that M_p is a linear mapping of one three-dimensional space into another and therefore can be expressed in homogeneous coordinates as a matrix comprising four column vectors. The first three vectors define how unit vectors in the three dimensions of the source space (i.e., permuted voxel space) map into the target space (i.e., image space). The fourth vector maps the origin of the source space to a point in the target space. Therefore, M_p can be represented as

$$M_p = \begin{bmatrix} dx_ix_v & dx_iy_v & dx_iz_v & X^0{}_{i,v} \\ dy_ix_v & dy_iy_v & dy_iz_v & Y^0{}_{i,v} \\ dDx_v & dDy_v & dDz_v & D^0{}_{i,v} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad (2)$$

where dx_ix_v, dx_iy_v, dx_iz_v, etc., represent incremental changes in the *x*-dimension of image space causes by unit steps in the *x*-, *y*-, and *z*-dimensions of voxel space, respectively, and where $X^0{}_{i,v}$, $Y^0{}_{i,v}$, and $D^0{}_{i,v}$, are image space coordinates of the origin of voxel space.

From Equation 1, $M_p = DW \times R$. From the definition of sample space, the x_s- and y_s-values are exactly the same as the x_i- and y_i-values of image space. By definition, the depth transformation *DW* only transforms the z_s-axis and leaves the x_s- and y_s-axes unchanged. Therefore, *DW* can be written as

$$DW = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ dDx_s & dDy_s & dDz_s & depth0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (3)$$

In Equation 3, dDz_s and *depth0* are known from the spacing of sample points along rays and the definition of the origin of sample space, respectively. This leaves only dDx_s and dDy_s as unknowns.

Similarly, *R* transforms permuted voxel space into sample space and can therefore be written as the matrix

$$R = \begin{bmatrix} dx_sx_v & dx_sy_v & dx_sz_v & X^0{}_{s,v} \\ dy_sx_v & dy_sy_v & dy_sz_v & Y^0{}_{s,v} \\ 0 & 0 & dz_sz_v & Z^0{}_{s,v} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (4)$$

From Equations 2 and 3 and the definition of sample space, it can be seen that the first two rows of *R* are identical to the first two rows of M_p, with x_s and y_s being substituted for x_i and y_i, respectively. The third row of *R* has zeros in the first two columns because sample slices are parallel to voxel slices — i.e., a unit step in either the x_v- or y_v-dimension of permuted voxel space produces no change in the z_s-dimension of sample space. This leaves two unknowns in *R*, namely dz_sz_v and $Z^0{}_{s,v}$.

Taking advantage of the fact that the lower left quadrant of *R* is zero, Equations 3 and 4 can be solved for dDx_s and dDy_s by

$$\begin{bmatrix} dDx_s \\ dDy_s \end{bmatrix} = \begin{bmatrix} dx_sx_v & dy_sx_v \\ dx_sy_v & dy_sy_v \end{bmatrix}^{-1} \begin{bmatrix} dDx_v \\ dDy_v \end{bmatrix}. \qquad (5)$$

Finally, *R* can be solved for its two remaining unknowns, namely

$$dz_sz_v = \frac{dDz_v - (dDx_s \times dx_sz_v) - (dDy_s \times dy_sz_v)}{dDz_s}, \text{ and}$$

$$Z^0{}_{s,v} = \frac{Z^0{}_{i,v} - (dDx_s \times X^0{}_{s,v}) - (dDy_s \times Y^0{}_{s,v}) - depth0}{dDz_s}. \qquad (6)$$

This derivation leads to a simple and efficient implementation, as described in the next section.

3.3 *Shear-image* order algorithm

The heart of *shear-image* order is a digital difference algorithm that steps through sample space one slice at a time, and within slices one row at a time, and within rows one sample at a time — i.e., it increments in the z_s-dimension most slowly, the y_s-dimension next, and the x_s-dimension most quickly. This is in contrast to many other ray casting algorithms, which follow one ray to its end before considering another ray. The increments for stepping turn out to be the entries of the inverse of *R*, i.e., R^{-1}.

$$R^{-1} = \begin{bmatrix} dx_vx_s & dx_vy_s & dx_vz_s & originX_v \\ dy_vx_s & dy_vy_s & dy_vz_s & originY_v \\ 0 & 0 & dz_vz_s & originZ_v \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (7)$$

Figure 6 shows the *shear-image order* algorithm in the form of two co-routines. In this figure, *k* is the number of slices that must be buffered ahead in order to estimate gradients and interpolate voxel values and gradients in the z_v-dimension. The two co-routines are:

[1] The spacing of samples along rays may be adjusted to trade off rendering speed *vs.* image quality. A reasonable default is that the distance between adjacent samples on a ray be approximately equal to the distance between adjacent rays.

- A *voxelStepper* routine that steps through the volume data set in slice major order in permuted voxel space. It reads ahead by *k* voxel slices, estimates gradients at each voxel point, and forms slices of *z*-interpolated samples and gradients that it sends to the other co-routine, the *sampleStepper*. By this means, the *voxelStepper* can process volume data in a memory-coherent way at very high speeds.
- A *sampleStepper* routine that processes each slice in the order of the *x*- and *y*-axes of sample space, which is aligned with pixels of the image plane. It keeps track of the permuted voxel coordinates of each sample, performs visibility tests on the sample point, and derives the sample value by 2D interpolation of the *z*-interpolated samples and gradients in the slice buffer passed by the *voxelStepper*.

The sample stepper can also test for early termination of a group of rays. If all of the rays of the group reach a threshold of opacity, it stops processing that group and notifies the voxel stepper to skip over the remaining voxels associated with that group.

3.4 Discussion

In volume rendering, images of interior structures are generated by assigning different opacities to different types of tissue or materials. The interfaces between different opacity levels create the appearance of 3D shapes. In our experience, the two most important factors in achieving high quality, visually meaningful images are the *ability to cast rays through the centers of the pixels* of the image plane and *a good illumination function*. Ray-per-pixel rendering avoids the artifacts and degradation that result from repeated resampling, especially with gradient calculation. In this section, we compare the *shear-warp*, *full image order*, *texture methods* and *shear-image order* volume rendering with respect to image quality, illumination, and memory efficiency. Table 1 on the next page shows a summary of these comparisons.

Full image order cay casting provides a natural sampling for ray-per-pixel image quality. Besides high image quality in general, it is easy to generate Multi-Planar Reconstructions (MPR's), the cross sections of the human body that are widely used in medical diagnosis. Because the fragments of the proxy geometry are naturally aligned with pixels on the image plane, it is also easy to embed polygon objects in a view of a volume. The cost of full image order rendering is loss of memory coherency and the seven interpolation operations needed for each component of each sample. 3D texture volume rendering is the only cost effective implementation of *full image order* to date on modern 3D graphics chips, and these have other limitations (see Section 2.3).

By comparison, s*hear-image* order keeps the ray-per-pixel feature of full image order rendering, thus preserving high image quality with or without lighting. The new factorization improves memory coherence and reduces interpolation cost to four multiplications per component per sample. However, it is more difficult in s*hear-image order* to create accurate MPR's at arbitrary angles to the volume. This is because sample slices are not, in general, parallel to the MPR angle, so a cut plane through the volume must be used, along with a falloff filter at the edges. Shear-image order also requires a view-dependent alpha correction, if we choose to keep a constant sampling distance in permuted z direction.

The fundamental difference between shear-warp and *shear-image order* is that shear-image order allows rotation and shear of the sampling plane onto which rays are projected. This avoids the post warp inherent in shear-warp, thereby enhancing the image quality, especially for images with illumination and fine details. Figure 7 on the color page shows a comparison of image quality between shear-warp (rendered by VolumePro 500) and the shear-image method (rendered by VolumePro 1000). The lung fibers are substantially clearer and more detailed with the *shear-image order*. Another difference from shear-warp is the ability to embed polygon objects. The samples of the base plane of *shear-warp* are not in one-to-one correspondence with the pixels of the image plane, so it is difficult to embed polygons drawn in the context of the image.[2] The cost of *shear-image order* is a more complex voxel stepper and four multiplication operations per sample interpolation instead of three.

2D texture methods are similar to shear-image methods, but there are two key differences. First, 2D texture methods require three separate copies of the volume, one for each dimension, to allow for viewing at all view angles. The VolumePro implementation of shear-image order does not, because it can apply the Permutation transformation on the fly, thereby by reading slices in any dimension from the same 3D array of data. Second, shear-image performs true tri-linear interpolation of the volume data. That is, every sample value and gradient is obtained from its eight nearest neighbors. This is a direct result of interpolating slices in the permuted z-dimension first, and then interpolating in permuted

```
//  Co-routine voxelStepper.
Pre-load cache with voxel slices $\lfloor originZ_v - k \rfloor$, $\lfloor originZ_v - k + 1 \rfloor$,
        …, $\lceil originZ_v + k - 1 \rceil$, and estimate gradients
for (int $z_s = 0$, real $z_v = originZ_v$; $z_s < lastSlice$; $z_s$++,
        $z_v$ += $dz_v z_s$)  {
    if not voxel slice $\lceil z_v + k \rceil$ in cache then
        {read slice $\lceil z_v + k \rceil$ and estimate gradients}
    $Bz_s$ = next z-interpolated sample/gradient buffer

    for ($y_v = minY_v$; $y_v < maxY_v$; $y_v$++)
        for ($x_v = minX_v$; $x_v < maxX_v$; $x_v$++) {
            Derive z-interpolated sample and gradient from
                voxel positions ($x_v$, $y_v$, $\lfloor z_v - k \rfloor$), …
                ($x_v$, $y_v$, $\lceil z_v + k \rceil$)
            Write sample and gradient to $Bz_s[x_v, y_v]$
        }
    Send $Bz_s$ to sampleStepper
}   //  end of voxelStepper

//  Co-routine sampleStepper
for (int $z_s = 0$, real $x_v = originX_v$, real $y_v = originY_v$, real $D_v$ =
        $depth0$; $z_s < lastSlice$; $z_s$++, $x_v$+=$dx_v z_s$, $y_v$+=$dy_v z_s$,
        $D_v$+=$dDz_s$) {
    Get $Bz_s$ from voxelStepper

    for (int $y_s = 0$, real $x_{vv} = x_v$, real $y_{vv} = y_v$, real $D_{vv} = D_v$;
            $y_s < lastRow$; $y_s$++, $x_{vv}$+=$dx_v y_s$, $y_{vv}$+=$dy_v y_s$,
            $D_{vv}$+=$dDy_s$)
        for (int $x_s = 0$, real $x_{vvv} = x_{vv}$, real $y_{vvv} = y_{vv}$, real
                $D_{vvv} = D_{vv}$; $x_s < lastColumn$; $x_s$++,
                $x_{vvv}$+=$dx_v x_s$, $y_{vvv}$+=$dy_v x_s$, $D_{vvv}$+=$dDx_s$) {
            if ($x_s$, $y_s$, $z_s$), $D_{vvv}$ is visible then {
                Derive value, gradient from $Bz_s[x_{vvv}, y_{vvv}]$
                Forward derived sample and gradient for
                    further processing and illumination
        }
}   //  end of sampleStepper
```

Figure 6: The *shear-image order* algorithm

[2] It would be possible to drawn the polygons in the graphics context of the base plane, then warp them along with the image of the volume. However, this introduces severe aliasing or requires substantial over-sampling.

x and y-dimensions within each slice. The basic 2D texture method does these in the opposite order, than is by interpolating first in the x and y-dimensions within a single 2D texture map, then interpolating between texture maps using a multi-texture technique [8]. This is not equivalent to tri-linear interpolation unless rays happen to be perpendicular to the slices of the volume. In other cases, the eight voxels that contribute to a sample would not, in general, be part of a 2×2×2 cube of the volume. Instead, they would be part of two 2×2 squares that could be offset from each other in either or both of the x and y-dimensions. This introduces artifacts that are not present with shear-image order, particularly when used to compute normal vectors for illumination.

On-the-fly gradient estimation to obtain normal vectors is difficult in commodity graphics hardware, not for any conceptual reason but because of the number of texture operations needed for each gradient generates an enormous load on the texture subsystem. The alternative is to pre-compute gradients when a volume is loaded. This reduces the number of texture operations required during rendering, but it quadruples the texture memory storage and bandwidth requirements.

VolumePro 1000 includes other operations not shown in the shear-image algorithm. For example, transfer functions may be applied either before or after interpolation. Illumination functions are also applied to highlight surfaces and create a realistic 3D appearance. Filters may also be applied based on a number of criteria to selectively enhance or suppress certain samples. For example, Figure 8 in the color page shows how surfaces can be highlighted by filtering out samples with small gradient magnitudes. This image is from a CT scan of a foot, and it highlights the skins and bones at the same time.

4 EMBEDDING POLYGONS

Volume visualization applications often need to render volume and polygon data together. For example, a surgical planning application might model a prosthetic device in a CAD environment, render it using conventional polygon graphics, and then embed the device into a volume rendered image of the patient's body. Similarly, a geophysical application might render seismic data as volume data but oil wells as polygons.

Figure 9 illustrates an example of a simple polygon object passing through the cranial cavity of a human head as rendered from a CT scan of a living person. The soft tissue of the brain has been rendered transparent to expose the bone and blood vessels. It can be seen that the polygon object lies in front of some parts of the volume (e.g., blood vessels and bone) and behind other parts.

	Shear Warp	**Full image/ 3D texture**	**2D texture**	**Shear image**
Memory Continuity	Yes	No	Yes	Yes
Interpolation	3 multiplications, true trilinear	7 multiplications, true trilinear	3 multiplications, bilinear or pseudo-trilinear	4 multiplications, true trilinear
Image quality	Post-warp from base plane	Pixel-per-sample	Pixel to texture slices	Pixel-per-sample
Embedded Geoemtry	No alignment between base plane and image plane	Direct	Direct	Depth adjustment per sample directly to pixel

Table 1: Comparison of different volume rendering methods

Various techniques have been used in the past to combine volume and polygon data into the same image. In methods in which volumes are converted to polygons, it is a simple matter to sort all of the polygons and render them using conventional polygon graphics. Another technique is to *voxelize* the polygon objects — i.e., convert them to voxels, then write them into the volume data set.

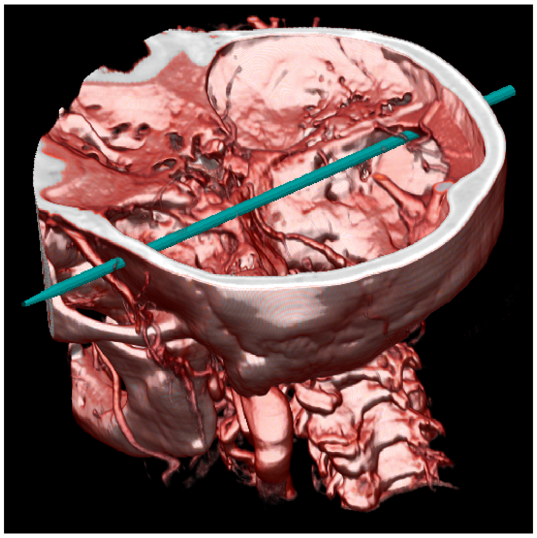
Figure 9: A polygon object embedded in a volume

Shear-image order makes it easy to use fast commodity graphics chips to render polygons and embed them into volumes. The polygons are rendered in the graphics environment using the same Model, View, Projection, and Viewport transformations shown in Figure 5. In particular, each polygon is scan-converted to the viewport so that the centers of its pixels are accurately and precisely aligned with the rays. When all of the polygons have been rendered, the depth and color buffers are captured and are used to control the following process:

- In the first pass, rays are initialized to the foreground color and then cast through the volume starting at the foreground and ending at the captured depth buffer. This renders the portion of the volume in front of the depth buffer.
- The previously captured color buffer is then blended behind the image plane resulting from the first pass.
- In the second pass, rays are initialized with the result of the blend operation, then cast starting at the depth buffer and terminating at the background. This renders the portion of the volume behind the depth buffer.

The result is an image of the polygonal objects embedded within the volume. These objects appear to the viewer to be in exactly the right places relative to the volume, independently of whether the polygons are opaque or translucent. Note also that in the case of opaque polygons, only the first rendering pass is necessary.

Using two depth buffers, the process can be generalized to arbitrary translucent geometry and to images of other objects, provided that they can be expressed as an ordered sequence of layers so that no two layers mutually obscure each other. This is illustrated in Figure 10. The rectangle is a cross section of a volume data set in the x_v- and z_v-dimensions, and the y_v-dimension is perpendicular to the page. Similarly, the heavy line is an edge view of the image plane showing the x_i-dimension and showing its pixels as × characters. The reader should imagine that the y_i-

dimension is perpendicular to the page. The curves depict edge views of three translucent objects render using polygon graphics. Rays are represented as arrows.

Note that from the *DW* transformation of Equation 3, the depth value of each sample point (x_s, y_s, z_s) is

$$D(x_s, y_s, z_s) = dDx_s{*}x_s + dDy_s{*}y_s + dDz_s{*}z_s + depth0. \quad (8)$$

The algorithm of Figure 6 keeps track of this for each sample in the *sampleStepper*. This value can be used to compare with the corresponding entry (x_i, y_i) of a depth buffer. Based on the depth comparison, the sample can be included or excluded.

Let the two depth buffers be labeled *D0* and *D1*. In the first pass of Figure 10, *D0* is initialized to the depth of the foreground, while *D1* is set to the depth buffer of the first layer of polygons. The volume rendering parameters are set to select only samples with depth values in the range [*D0*, *D1*) — i.e., samples (x_s, y_s, z_s) that satisfy

$$D0[x_s, y_s] \le D(x_s, y_s, z_s) < D1[x_s, y_s] \,. \quad (9)$$

At the end of the pass, the first color buffer is blended behind the image produced by the volume-rendering pass.

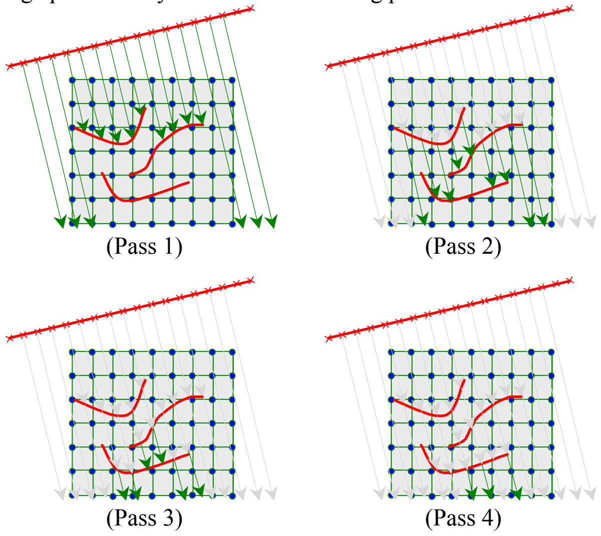

Figure 10: Embedding Polygon Objects

At the beginning of the second pass, the contents of *D0* are replaced by the contents of *D1*, and the depth buffer of the second layer is loaded into *D1*. The rays are initialized to the blended image from the previous pass, and the volume is rendered again, selecting only the samples in the range [*D0*, *D1*). Then the second color buffer is blended behind the result. This process is repeated for each layer. In Figure 10, four passes are seen, with the portions of the rays of the current pass highlighted and the rays of previous passes shown in gray.

By this means, each polygon object is inserted pixel-by-pixel between the samples along the rays. Obviously, the process must be repeated for each change in view direction, model transformation, and other parameter. The method can also be extended to embed images of non-polygon objects, both opaque and translucent, provided that they can be expressed as a sequence of color and depth buffers.

VolumePro 1000 supports dual depth buffers and a flexible set of depth tests as part of the rendering engine. It also allows depth buffers to be updated dynamically based on various conditions. This is useful, for example, in creating a “depth mask” for the visible surface or picking voxels of a volume. The basic volume-rendering algorithm of VolumePro 1000 is optimized to skip efficiently over slices and groups of slices that would fail the depth tests.

5 ANISOTROPIC AND SHEARED DATA

Anisotropic data sets — in which voxels are spaced differently in each dimension — are the rule rather than the exception in medical and geophysical imaging. In seismic applications, for example, distances on the surface of the earth are measured in miles or kilometers, but the vertical direction into the earth is often measured in the time it takes to detect an echo. In CT (computed tomography) scans, the spacing of slices in the longitudinal axis of the patient is determined by the (adjustable) speed of the table, while the spacing within a slice is determined by the geometry of the scanner. Also common are sheared data sets, in which the axes are not at right angles to each other. For example, the gantry of a CT scanner may be tilted with respect to the axis of the patient.

The anisotropy and shear of a volume are described by its Correction transformation of Figure 5 and therefore are taken into account in the derivation of *R* in Section 3.2. These will position sample points along rays according to the rendering parameters, regardless of view direction, anisotropy and shear, thereby producing a correct view of the volume. Most of the images in this paper are rendered from anisotropic data sets.

By illuminating each sample point, a volume rendering system can expose surfaces within the volume and give them a realistic 3D appearance. These lighting calculations depend upon the gradient at each sample point. It is easy to estimate gradients from voxel values in a rectilinear volume using central differences or other separable convolution kernel. High quality kernels are described in [7]. Separable kernels have the advantage that each of the three gradient components can be obtained independently.

The problem with anisotropic and sheared volume data is that separable kernels produce gradients with the wrong direction, wrong magnitude, or both. To obtain accurate gradients directly from the volume data requires a full three-dimensional convolution, which is computationally prohibitive in a hardware accelerator. Therefore, practical systems continue to use separable kernels but correct gradients before using them in lighting calculations.

To derive the gradient correction, we consider an anisotropic volume in both world space and voxel space. Figure 11 is a stylized representation in two dimensions. The top left shows an object in world space with its gradient, the light direction, and the eye vector. The top right shows the same object in voxel space; the volume appears distorted on account of its anisotropic sampling, and the gradient points in the wrong direction. Therefore, diffuse and specular lighting calculations would be incorrect if done in this space.

Applications typically specify lights in world space. Therefore, it would be sufficient to correct the gradient to world space and use the world space versions of the light direction and eye vector. However, even this simple correction would require a 3×3 matrix and nine multiplications per gradient. This was considered to be an expensive use of space in the semiconductor implementation of VolumePro 1000, especially because multiple gradients are processed in parallel.

An alternative is to decompose the *Model*×*Correction* transformation into a shear-scale transformation and a rotation transformation and to define a new intermediate coordinate system called *Lighting space*. This decomposition is shown at the bottom of Figure 11. The shear-scale transformation *LS* can be to convert the gradient to lighting space, while the controlling software can transform its lighting calculations from world space to lighting space by the rotation $(LR)^{-1}$. This rotation preserves dot products between vectors. From Figure 11, it follows that

$$LR^{-1} \times M \times C = LS. \tag{10}$$

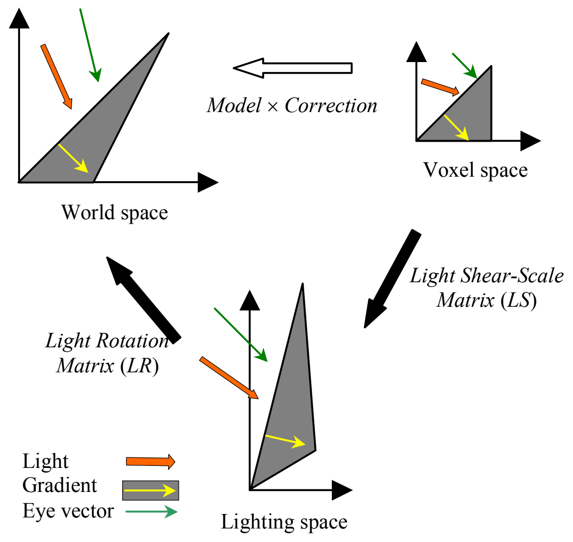

Figure 11: 2D Illustration of Lighting Space

Lighting space is a rotation of world space but a shear-scale of voxel space. It is not the same as object space.

Mapping gradients of voxel space into surface normals in lighting space requires the multiplication by $(LS^{-1})^T$, that is, the inverse transpose of the shear-scale lighting matrix. Therefore, VolumePro 1000 includes a gradient correction matrix in hardware of the form

$$G = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ 0 & g_{22} & g_{23} \\ 0 & 0 & g_{33} \end{bmatrix} = \left(LS^{-1}\right)^T$$

to convert gradients into lighting space. This correction requires *six* multiplications per gradient in the general case but only *three* multiplications per gradient in non-sheared, anisotropic cases. It should be noted that when lights rotate with the volume object, the *LS* matrix does not change from frame to frame. This saves the time needed to recompute light maps for each angle of rotation.

Figure 12 on the color page is a dramatic illustration of the importance of illumination in volume rendering. This shows the blood vessels in a human brain from an anisotropic CT scan of a living person.

6 IMPLEMENTATION

VolumePro™ 1000 is a second-generation volume rendering system that implements the *shear-image* order method. It comprises an ASIC (application-specific integrated circuit), a board that can be plugged into the PCI bus of a personal computer or workstation, and a library of controlling software. The board includes the ASIC and up to 2 gigabytes of high performance memory. The VolumePro 1000 is a generalization in many dimensions of the VolumePro 500.

The VolumePro 1000 ASIC is the rendering engine implemented as a semiconductor. It includes a Sequencer, four processing pipelines, a memory controller, a PCI bus interface, on-chip caches of various lookup tables, and on-chip buffers for voxels, *z*-interpolated samples, pixels, and depth values. The ASIC implements the *shear-image* order algorithm of Figure 6, along with other volume rendering functions including gradient estimation and correction, classification, illumination, alpha correction, filtering based on gradient magnitude and other properties, and cut and crop planes. The Sequencer and processing pipelines operate at 250 MHz, so that the ASIC can render 10^9 samples per second.

Memory is organized so that 3D objects are stored as mini-blocks of *2×2×2* voxel values, and 2D objects are stored as *2×2* stamps of pixel values. This allows sequences of related data values to be read or written in burst mode, thus maximizing the available bandwidth of memory chips. The memory subsystem itself comprises eight channels of 16-bit Double Data Rate Synchronous DRAM (DDR-SDRAM) operating at 133-166 MHz. Eight 16-bit voxels can be fetched or four 32-bit pixels can be written per memory cycle. The net memory bandwidth is 4.2-5.3 gigabytes per second. Operational experience suggests that at least 50% more memory bandwidth should have been provided.

Each pipeline includes a gradient estimation module, a classifier for mapping voxel values into RGBα pixel values, an interpolator in two parts to execute the *shear-image* order algorithm, an illuminator, and a compositor. The classifier and interpolator are cross connected so that classification and interpolation can proceed in either order. There are functional as well as aesthetic reasons where one order might be more appropriate than the other in a particular application. Classification first mode is important especially for rendering volumes with mask fields (for example, fields indicate they are bones, vessels etc.).

Voxels may have up to four fields, programmable by the application as to size, position, and format. Each field is associated with its own lookup table for mapping field values to color and opacity values. These can be combined by a hierarchical set of arithmetic-logic units as described in [5]. The interpolator is linear in the *z*-dimension and bi-linear in the *x*- and *y*-dimensions, thereby requiring four multiplications per sample. There are seven interpolation channels, one for each of the voxel fields or color-opacity components plus one for each gradient component.

The illuminator is a reflectance map implementation of the Phong module of lighting. It provides emissive, diffuse, and specular lighting from an arbitrary number of light sources. It also provides a modulation function based on the magnitude of the gradient. The compositor provides alpha-blending, maximum and minimum intensity projection, sum and count, and other functions for combining sample values along rays. It includes an alpha-correction function to adjust opacity values for different spacing of samples. The compositor also implements an early ray termination test that stops processing of an individual ray when it reaches a threshold of opacity.

VolumePro 1000 tries very hard to skip over samples that are not visible. The Sequencer keeps track of portions of the volume or image that are cut, cropped, clipped, or that fail depth tests, and it jumps over them when it is useful to do so. This kind of space leaping is called *geometry-based* space leaping because it depends only upon the position of a sample, not its value. A second kind of space leaping, called *content-based* space leaping because it jumps over samples that are invisible by virtue of opacity assignment or filtering, was not included in VolumePro 1000.

The actual performance of VolumePro 1000 is proportional to the number of rays in the image plane and how quickly they terminate. By contrast, VolumePro 500 performance is always proportional to the number of voxels in the volume data set.

A dominant consideration in a semiconductor implementation of a volume-rendering algorithm is the amount of on-chip memory needed. Both the *shear-warp* algorithm and *shear-image* order require one or more full slices of voxel values to be cached on the chip. This is a severe constraint and one of the most important factors in the ASIC design. Both generations of VolumePro solve this problem by partitioning the volume data set into *sections* and by rendering the volume one section at a time. The amount of on-chip memory is thus proportional to the number of voxels and/or samples in a section.

VolumePro 1000 implements early ray termination on a section-by-section basis. While it is rendering a section, it maintains a bit map indicating which rays have reached a (programmable) threshold of opacity. When the bit map is full, the section is terminated and the next section is begun.

Figure 13 of the color page shows zoomed views of the carotid arteries rendered by VolumePro 1000. High image quality with lighting shows the details of the image. Although a printed article cannot show it, all of the images of this paper (except Figure 7, right) were generated at interactive speeds.[3]

7 CONCLUSIONS & FUTURE WORK

This paper describes the *shear-image* order ray-casting algorithm along with its advantages in embedding polygon geometry and direct rendering anisotropic and sheared volume. The paper also introduces the implementation of this method in VolumePro 1000, which provides high image quality, high speed and various advanced features. VolumePro 1000 supports the real-time volume visualization of large volume data sets for a variety of interactive applications. This paper only addresses parallel projections.

Two challenges in the implementation of a future generation of VolumePro are content-based space leaping and ray-per-pixel rendering in perspective projection. Content-based space leaping can be implemented with a bit map indicating the spatial areas of the volume object that are transparent and those that are not. The challenge is providing enough on-chip memory for the bit map and in accessing it fast enough to gain a performance benefit.

Efficient perspective rendering is needed maintain a sense of the viewer's position within the volume, for example in interactive fly through of the colon. Both 2D texture methods and shear-warp can be made to support perspective volume rendering, and shear-warp methods can be efficient [2]. However, as in the parallel projection, the need for a post warp step in perspective versions of shear-warp degrades the image quality. VolumePro 1000 currently supports perspective using a variation of shear-warp.

The shear-image algorithm can be adapted to support ray-per-pixel perspective volume rendering. However, the image must be partitioned to avoid rays that diverge so they become too nearly parallel to slices of samples. The different partitions require different permutation matrices, and the slices of samples are parallel to different faces of the same volume. As a result, the volume must be rendered in multiple passes, one per partition. In addition, sample spacing varies from ray to ray in perspective projection. For high quality images, opacity values need to be adjusted on a ray-by-ray basis. (A single opacity adjustment per partition rays is used as an approximation in VolumePro 1000.)

Acknowledgements

This work began when all four authors were members of Mitsubishi Electric Research Laboratories (MERL) of Cambridge, Massachusetts. Authors Wu, Bhatia, and Lauer continued as members of TeraRecon, Inc., after TeraRecon acquired the VolumePro business from Mitsubishi Electric in 2001.

Jan Hardenbergh is a principal contributor to the architecture of VolumePro and acted as our proofreader, advisor, conscience, and chief image wrangler in the preparation of this paper. Through her persistence as a user, Lisa S. Avila of Kitware forced us to get the definition of Lighting Space right.

We would also like to thank James Foley, Sarah Frisken (Gibson), Hanspeter Pfister, Vikram Simha, Andy Vesper, and T. C. Zhao for their insight into the issues of real-time volume rendering. Thanks are also due to Andy Adams, Bill Booth, Steve Burgess, Kenneth Correll, James Knittel, Jeff Lussier, Bill Peet, and Jay Wilkinson for making the VolumePro 1000 semiconductor and hardware a reality.

[3] The shear-image rendering of the lung was generated on a bit accurate simulator of VolumePro 1000 prior to fabrication of the ASIC. It required about 75 hours.

References

[1] Cabral, Brian; Cam, Nancy; Foran, Jim, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," *Proceedings of the 1994 symposium on Volume visualization*, Tyson's Corner, Virginia, USA**,** 1994.

[2] Brady, Jung, Nguyen H, Nguyen T. Two phase Perspective Ray Casting for Interactive Volume Navigation. *Visualization 1997.*

[3] Engel, Klaus; Kraus, Martin; Ertl, Thomas; "High-Quality Pre-Integrated Volume Rendering using Hardware-accelerated Pixel Shading," *Eurographics/SIGGraph Graphics Hardware Workshop*, 2001.

[4] Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F., *Computer Graphics, Principles and Practice*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1987.

[5] Gasparakis, C. Multi-resolution Multi-field Ray Tracing: A Mathematical Overview, *Proceedings of the Conference on Visualization 99*, ACM SIGGRAPH, San Francisco, October 1999.

[6] Lacroute, Philipe, and Levoy, Marc. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Proceedings of SIGGRAPH 94* (Orlando, Florida), Computer Graphics Proceedings, Annual Conference Series, pages 409–412. ACM SIGGRAPH, ACM Press, July 1994.

[7] Lichtenbelt, Barthold, Crane, Randy, Naqvi, Shaz, *Introduction to Volume Rendering*, Prentice-Hall PTR, Upper Saddle River, New Jersey, 1998.

[8] Möller, Torsten, Mueller, Klaus, Kurzion, Yair, Machiraju, Rahgu, and Yagel, Roni. Design of Accurate and Smooth Filters For Function and Derivative Reconstruction. *Proceedings of 1998 Symposium on Volume Visualization*, ACM SIGGRAPH, pp. 143-151, October 1998.

[9] Pfister, Hanspeter, Hardenbergh, Jan, Knittel, James, Lauer, Hugh, and Seiler, Larry. The VolumePro Real-time Ray-casting System. *Proceedings of SIGGRAPH 99* (Los Angeles, California), pages 251-260, August 1999.

[10] Rezk-Salama, C.; Engle, Klaus; Bauer, M.; Greiner, G.; Ertl, Thomas; "Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization," *Proceedings of 2000 Siggraph/Eurographics Workshop on Graphics Hardware*, Interlaken, Switzerland, 2000.

[11] Levoy, Marc. Display of surfaces from volume data*. IEEE CG&A 8(3), pp. 29-37.*

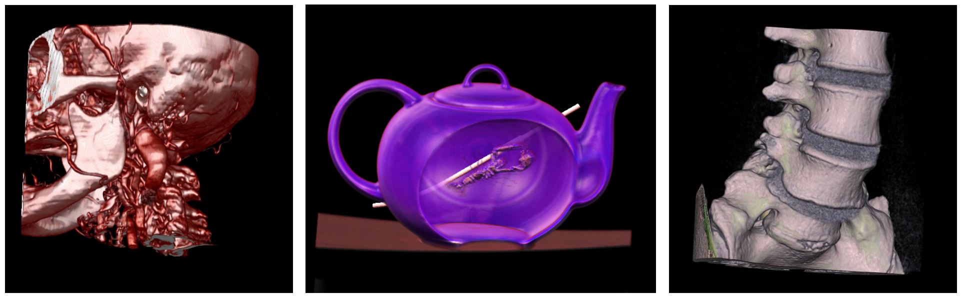
Figure 1: Shear-Image Rendering Gallery: Various volumes with lighting effects or embedded geometry

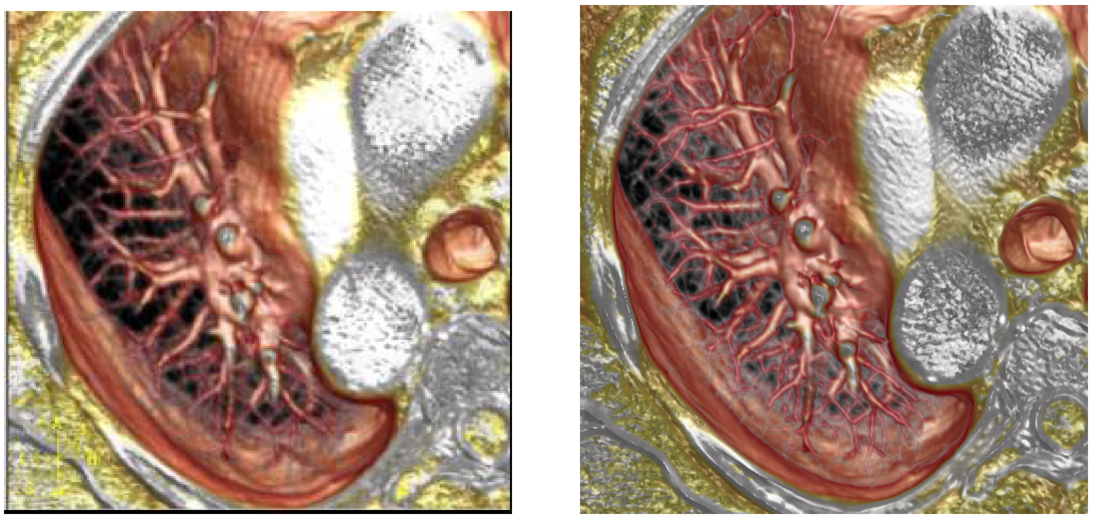
Figure 7: Comparison of shear-warp (left), and shear-image order (right)

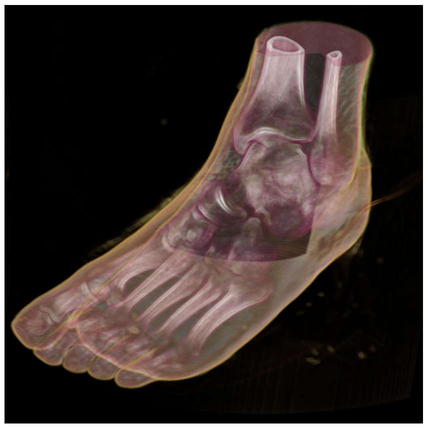
Figure 8: Gradient magnitude modulation

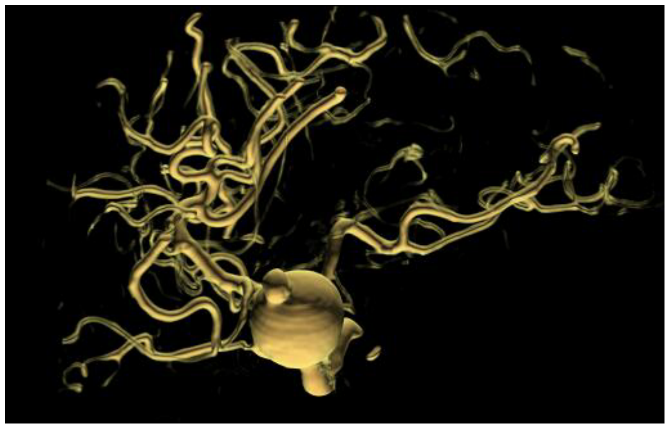
Figure 12: An image of a cerebral aneurysm

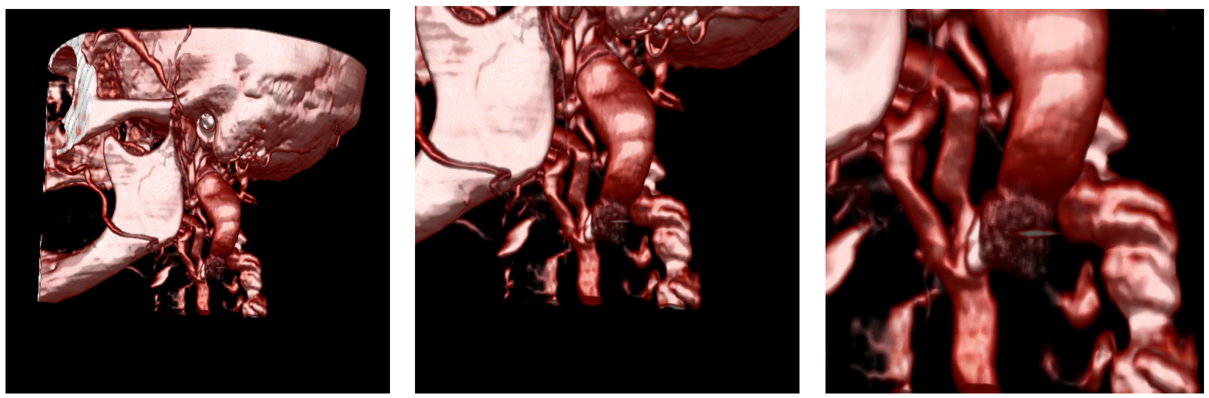
Figure 13: Three zoomed views of the carotid arteries of the neck, showing the image quality of shear-image order