Direct Volume Rendering with Shading via Three-Dimensional Textures

Allen Van Gelder   Kwansik Kim
Baskin Center for Computer Engineering and Computer Science
University of California
Santa Cruz, CA 95064 USA
E-mail: avg@cse.ucsc.edu, ksk@cse.ucsc.edu

UCSC–CRL–96–16

July 19, 1996

Abstract

A new and easy-to-implement method for direct volume rendering that uses 3D texture maps for acceleration, and incorporates directional lighting, is described. The implementation, called *Voltx*, produces high-quality images at nearly interactive speeds on workstations with hardware support for three-dimensional texture maps. Previously reported methods did not incorporate a light model, and did not address issues of multiple texture maps for large volumes. Our research shows that these extensions impact performance by about a factor of ten. Voltx supports orthographic, perspective, and stereo views. This paper describes the theory and implementation of this technique, and compares it to the shear-warp factorization approach.

A rectilinear data set is converted into a three-dimensional texture map containing color and opacity information. Quantized normal vectors and a lookup table provide efficiency. A new tesselation of the sphere is described, which serves as the basis for normal-vector quantization. A new gradient-based shading criterion is described, in which the gradient magnitude is interpreted in the context of the field-data value and the material classification parameters, and not in isolation. In the rendering phase, the texture map is applied to a stack of parallel planes, which effectively cut the texture into many slabs. The slabs are composited to form an image.

Keywords: Computer Graphics, Scientific Visualization, Texture Map, Direct Volume Rendering, Spherical Tesselation.

Note: This report appears in *Proceedings of ACM/IEEE Symposium on Volume Visualization*, San Francisco, Oct., 1996, except for the following modifications. The appendix of this report does not appear in the proceedings. References to the appendices are replaced by references to this report, in the proceedings. The proceedings includes 1/2 page of color images that are not included in the report, as distributed electronically. The color images are available separately from the authors.

1 Introduction

Rendering speed has always been a major problem in direct volume rendering, because all regions of the volume may contribute to the image, and because new orientations generally require considerable re-computation. A spectrum of methods offering different combinations of rendering speed versus image quality have been presented [DCH88, Sab88, UK88, Lev88, Kru90, Lev90a, MHC90, Wes90, LH91, Lev92]. Image-order algorithms like ray casting produce superior quality images, but have to traverse the data structure redundantly [Lev90b, DH92]. Some object-order methods use hardware-assisted Gouraud-shading capabilities to accelerate rendering, by calculating the projections of volume regions and treating them as polygons [ST90, LH91, WVG91, Wil92, VGW93, WVG94]. Shear-warp factorization methods have been developed for parallel volume rendering [CU92, SS92], and serial [LL94]. The method of this paper is compared with shear-warp in Section 4.

This paper presents a new method for direct volume rendering, called *Voltx*, that takes advantage of hardware-assisted, 3D texture mapping, and incorporates a light model. We have found Voltx to be significantly faster than the hardware-assisted Gouraud-shading method (called *coherent projection*), and comparable in speed to the shear-warp algorithm of Lacroute and Levoy (called *LL*) when comparable imaging options are selected. Whenever one voxel does not cover too many pixels, Voltx produces images comparable to ray-casting. Perspective projections, which are essential for comfortable stereographic viewing, do not degrade the performance. A stereo interface has been implemented.

Voltx is robust in the sense that it is able to produce images of good quality even though the transfer function has high frequencies or discontinuities. Also, rotations of the volume do not introduce artifacts, and perspective projections do not degrade the performance. All of these problems were reported for LL by Lacroute and Levoy [LL94]. Their paper suggests ways to get around some of the problems.

Some limitations of Voltx are the restriction to rectilinear volumes, the need to redefine 3D texture maps whenever the volume is viewed from a new rotation, and the fact that color, not data, is loaded into texture memory. The first limitation may be inherent in the method. The latter limitations apply only when directional lighting is chosen, but such lighting is usually needed to obtain informative images. They might be addressed as described in Section 3.3.

Several other papers describe the use of 3D texture-mapping hardware for direct volume rendering. Akeley briefly mentioned the possibility for the SGI Reality Engine [Ake93]. Cullip and Neumann sketch two approaches, which they call object space (somewhat related to shear-warp) and image space, and apply them to CT data [CN93]. Guan and Lipes discuss hardware issues [GL94]. Cabral, Cam and Foran describe how to use texture-mapping hardware to accelerate numerical Radon transforms [CCF94]. All of the methods described in these papers require substantial programming to “hand compute” transformations, clipping, and the like, details of which are omitted. Wilson *et al.* reported on an earlier version of Voltx [WVGW94]. To our knowledge no previously reported 3D texture-map method addresses either shading with a light model or the necessity of multiple texture maps for large volumes. Our research shows that these extensions impact performance by about a factor of ten.

After a quick overview in Section 2, this paper describes how most of the technical programming used by previously reported 3D texture-map methods can be eliminated by use of graphics library procedures to perform texture space transformations and set clipping planes (Section 3.2). A new method to judge which voxels are on a material boundary within the volume, for shading purposes, is described in Section 3.1.2. We also present a software look-up table technique with quantized gradients (actually, normal vectors), which makes our rendering technique nearly interactive, while providing excellent image quality (Section 3.1.3). A new tesselation of the sphere is described, which serves as the basis for normal-vector quantization (Section 3.1.4). Section 4 compares Voltx to shear-warp factorization. Experimental results are reported in Section 5. Conclusions and future directions are discussed in Section 6. Additional details are available in the appendices.

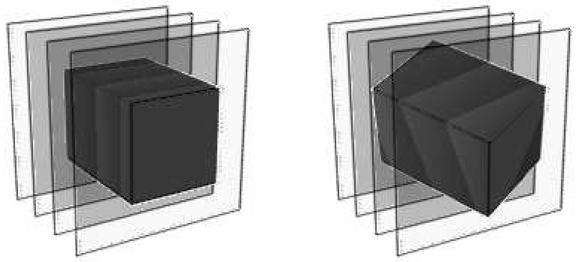


Figure 1: Slices through the volume in original orientation (left), and rotated (right). The viewer's line of sight is orthogonal to the slices. In both orientations, texture coordinates are assigned to corners of the square slices in such a way that they interpolate into the range $[0, 1.0]$ exactly when they are within the volume.

2 Overview of 3D Texture-Map Technique

In an initial step, the quantized gradient index and material classification of each voxel in the volume are computed. A voxel may be classified as either reflecting or ambient, depending on the client-supplied gradient-magnitude threshold. (If this threshold is modified, the voxels must be reclassified.) The result of this step is an index for each voxel into the lookup table described next.

Whenever a new orientation of the volume or new light position is specified by the client, a software lookup table is created that assigns, for each combination of quantized gradient [1] and reflecting material, the appropriate color and opacity (RGBA). Color and opacity are calculated to account for depth integration. Color calculations for ambient material are also performed, but they are independent of the gradient, and consume negligible time. Thus the size of the lookup table is effectively the product of the number of quantized gradients and the number of reflecting materials, and is independent of the size of the volume. References to "color" generally mean color and opacity in the ensuing discussion.

With the pre-assigned look-up table index of each voxel, 3D texture maps are filled with pre-computed color values, one "texel" per volume voxel. While this work is proportional to the size of the volume, it involves only few integer operations from one array to another, per voxel. The preceding steps are easily parallelizable in a MIMD shared-memory environment. However, transferring the texture maps into texture memory, which uses the graphics pipeline, is *not* parallelizable. Limits on texture memory force large volumes to be partitioned into multiple texture maps.

These texture maps are then processed in the back-to-front visibility order of the partitions of the volume that they represent. Each one is applied to many parallel planes, which make slices through the texture (see Figure 1). Each plane, orthogonal to screen-z, is rendered as a square, so texture coordinates need be specified only at its four corners. Whenever the lighting model changes or the client's viewing direction changes, the lookup table entries are updated and the texture map is redefined.

In 3D texture mapping, each polygon vertex is given a point in the texture space, and the graphics system maps values from the texture map onto the polygon surface by interpolating texture coordinates. This is very similar to Gouraud shading except that texture coordinates are being interpolated instead of colors. The crucial difference is that *texture coordinates* outside of the range $[0, 1.0]$ are still interpolated, whereas such *colors* would be clamped. The corners of the squares have out-of-range texture coordinates, but interpolation creates in-range values precisely when the pixel is within the volume.

[1] We shall refer to "quantized gradients" although only the direction is quantized, and not the magnitude.

The squares are parallel to the projection plane in screen space, while the 3D texture map can be oriented as the client desires. More squares at thinner spacing, up to a point, give better image quality, while fewer give greater speed.

The major advantage of this method is that after the original data is converted into 3D texture maps, the Reality Engine's specialized texture hardware can perform the rendering and compositing of squares very quickly. While few graphics workstations offer 3D texture mapping in hardware at present, we believe it will become more common, providing a quick and simple direct volume rendering method for rectilinear data.

3 Detailed Description of Voltx

The two major steps in Voltx are: first, create the texture map (Section 3.1); and second, render the slices (Section 3.2). Frequencies of these steps is discussed in Section 3.3.

3.1 Creating the Texture Map

The texture-map calculations have several parts, which are triggered by different events. Voxel classification occurs when the transfer function or certain gradient parameters change; a lookup table is recalculated when the volume rotation changes, or the light position changes; the texture map is filled whenever any of the above events occur. This description assumes directional lighting is in effect. For ambient light only, faster alternatives exist (Section 3.3).

Each texture-map entry, or "texel", corresponds to one voxel. Its color is the sum of ambient and reflecting components. The "ambient" component is based on the traditional "luminous gas" model used in direct volume rendering. The reflecting component is based on a surface responding to directional light, and only applies at parts of the volume judged to represent the boundary surface between different materials. Details are spelled out below.

3.1.1 Ambient Light Component

We interpret each plane to be rendered as being at the center of a slab (slice with thickness) through the data. This means that each plane must contribute the color intensity and opacity due to one such slab. The thickness of every slab, Δ, is just the total distance covered by the stack of planes divided by the number of planes. Let Δ be the slab thickness. Let $E(p)$ denote the color emission per unit distance, for each primary color p, and let A_1 denote opacity per unit distance. These values are computed at each voxel of the volume based on the client-supplied *classification function* and *transfer function*. The color $C(p)$ and opacity A that must go into the texture map are computed by now-standard methods [WVG91, VGW93]. Briefly, the formulas are:

$$\begin{aligned} \alpha &= \ln\left(\frac{1}{1-A_1}\right) \\ C(p) &= E(p)\left(\frac{1-e^{-\alpha\Delta}}{\alpha\Delta}\right) \\ A &= 1-e^{-\alpha\Delta} \end{aligned}$$

where p ranges over *Red*, *Green* and *Blue*. Note that C is well-defined as α approaches 0, by taking a power series expansion.

This equation expresses the integration of color and opacity through the thickness of the slab, without considering the shading model, and is applied to both ambient and reflecting materials.

For perspective viewing, the effective slab thickness increases slightly with the off-axis angle. Our implementation ignores this effect, which is at most 4% on the periphery of the image, for normal viewing geometry with a 600-pixel window width (about 15 cm.) and viewing distance 70 cm. The error can increase to 14% for a 1000-pixel window width. One way to compensate for this variation is to render concentric spherical shells, rather than planes. The spherical tesselation technique of Section 3.1.4 is suitable for generating the triangles.

3.1.2 Reflecting Surface Classification

A voxel is considered to be reflecting material when it is judged to be "near" the boundary of a volume segment composed of that material, based on the range of field values for that material, and the gradient of the field function.

Example 3.1: Consider a CT data set with unit spacing between voxels, in which *bone* is considered to be 110 and greater densities. Suppose a voxel's data is 114, and the gradient magnitude is 10. We estimate that the bone boundary is 0.4 units from the location of this voxel, traveling in the direction of the negative gradient. If the voxel spacing is 1 unit, we judge the voxel to be on the boundary of *bone*, and classify it as reflecting material.

However, if the voxel's data is 130, even though the gradient magnitude is still 10, we judge that that neighboring voxels are also *bone*. Thus, this voxel is *not* considered to be on the surface, and we classify it as ambient material. $\square$

Given a voxel at location $\bar{q_0}$, with data value d, and gradient vector γ, an approximate model of the field function in its neighborhood is

$$\tilde{f}(\bar{q}) = d + \gamma \cdot (\bar{q} - \bar{q_0})$$

Let b be the lower limit of the range of data that is classified as the same material as d. (In the example, b was 110.) Let $\bar{\Delta}$ be the inter-voxel spacing. Based on this linear approximation, one of the 26 neighboring voxels has data below b if and only if $(d - b) < s$, where

$$s = \bar{\Delta}_x |\gamma_x| + \bar{\Delta}_y |\gamma_y| + \bar{\Delta}_z |\gamma_z|$$

Let us call the quantity s the *cell-diagonal data shift*. A similar test applies for the upper boundary a, using $(a-d) < s$.

When s is near the value of $(d - b)$ or $(a - d)$, we can assign a "probability" of being on the boundary, instead of making a binary decision, to allow for noise in the gradient and nonlinearity of the field function. In the implementation, when $s \leq 0.5(d - b)$, the "probability" is set to 0, and it increases linearly to become 1 when $s = 1.5(d - b)$. The client can fine tune the classification by means of a scaling factor to be applied to this probability function, causing either more or fewer voxels to be judged to be on a surface with some nonzero "probability". The final shading for the voxel combines the ambient and reflecting colors and opacities according to the "probability".

Thus the gradient magnitude is interpreted in the context of the data value and the classification parameters, and not in isolation, departing from earlier work [Lev90a, LL94].

3.1.3 Reflected Light Component

As mentioned, for each combination of quantized gradient and material, the material's surface response to directional lights is calculated, added to the ambient component, and stored in a lookup table. Our implementation uses the "half-way vector" specularity model [TM67, CT82], as well as Lambertian diffuse shading. Gradient quantization is discussed in Section 3.1.4.

The resulting color intensities, clamped if necessary, are real numbers in the range $[0, 1.0]$ where 0 is black and 1.0 is full color. These floating point values must be converted to integers in an appropriate range, and packed into a 32-bit or 64-bit texture-map entry, for storage immediately in the lookup table, and eventually in the texture map.

Let the client have specified:

- m as the number of materials;
- k as the number of refinement levels for quantized gradients, as described in Section 3.1.4; then
- $g = 30 * 4^k + 2$ is the number of quantized gradients.

The lookup table will have $(m + 1) \times g$ entries which are possible combinations of materials and gradients. Material 0 is the null material; however its places in the lookup table are used to store colors for ambient materials, which need not be combined with a gradient index.

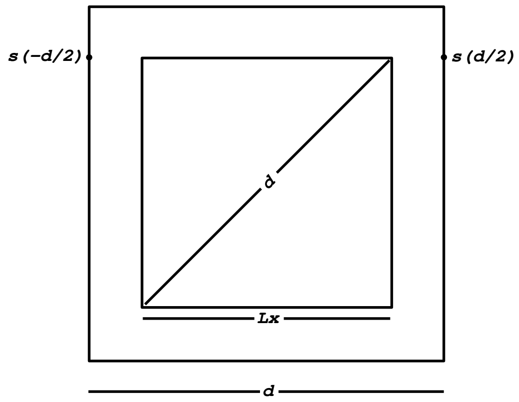


Figure 2: Geometry of Texture Coordinates. This illustrates an orthographic projection of a 3-D volume of size (L_x, L_y, L_z) and the bounding cube of side d. Sides L_y and L_z are not labeled. The long diagonal through the volume is length $d = (L_x^2 + L_y^2 + L_z^2)^{\frac{1}{2}}$.

In the setup procedure, each voxel's material index M is classified and its field gradient is calculated. If the voxel has reflection properties, according to the criteria of Section 3.1.2, then the quantized gradient index G is determined by finding the closest available quantized gradient. The voxel's final lookup index is $(Mg + G)$. If the voxel has no reflection properties, its index is simply M.

The lookup table needs to be recalculated whenever any factor involving light changes, including a rotation of the volume or a movement of the light source. Previously reported methods for volume rendering with 3D texture maps did not allow for directional lighting effects, and therefore could avoid recalculation after rotation of the volume [CN93, CCF94, WVGW94].

3.1.4 Gradient Quantization

We quantize the gradient direction space into a finite number of unit gradients. The goal is to distribute a set of points on a unit sphere as evenly as possible, and take the vectors from origin to the points as the quantized gradients. Our method tesselates the sphere into triangles, and uses the triangles' vertices as the quantized gradient directions. Numerous tesselations of the sphere are known. Two are based on equilateral spherical triangles as the starting point: the regular octahedron, with 8 triangles and the regular icosahedron, with 20 triangles. We have developed a new method by considering both an icosahedron and its dual figure, the dodecahedron, with 12 pentagons. See Figure 7, top, which shows the vertices of these dual polyhedra. Note that the icosahedron has 12 vertices (black) and the dodecahedron has 20 vertices (gray). This combination provides 60 congruent, but not quite equilateral, triangles as the starting point.

A typical method to refine the initial quantization is with recursive subdivision on the starting triangles. By connecting midpoints of sides, each triangle is partitioned into four new triangles. However, on the sphere, unlike the plane, the new triangles are not congruent. For example, a quick and easy way is to subdivide the equilateral triangles from the regular octahedron. Unfortunately, this method will not produce very evenly distributed gradients, because certain triangles will continue to have a 90° angle, while others approach the more desirable 60° angles. Starting from the icosahedron's 20 triangles, the discrepancy is less, varying from 72○ to 60°.

Our new method starts from the 32 vertices of the icosahedron and dodecahedron, combined, which produce 60 triangles. These triangles, being more nearly planar than the starting spherical triangles associated with the regular polyhedra, undergo less distortion during subdivision. The result of two levels of refinement is shown on the bottom of Figure 7, where refinement vertices are shown as smaller dots.

Images in this paper are based on four levels of refinement, yielding 7682 vertices. The general rule for the number of vertices after k levels of refinement is $g = 30 * 4^k + 2$.

3.2 Rendering Slices

Once the 3D texture is created, we render the volume by applying the texture to parallel planes (represented as squares in screen-space) and, thus, build up a stack of slabs through the texture, each slab being the region between two adjacent slices (see Figure 1). The squares are drawn from back to front.

3.2.1 Texture Coordinates for Original Orientation

Consider a world space (x, y, z) coordinate frame in which the center of the volume is the origin. Texture coordinates (s, t, r) will become proxies for spatial (x, y, z). Essentially, we construct a bounding cube centered on the origin that is large enough to contain any rotation of the volume. The side of the bounding cube needs to be the length of the diagonal through the volume, which we shall denote by d. The squares to be rendered (see Figure 1) comprise a series of slices through the bounding cube, parallel to the cube's xy faces.

Now suppose the volume has resolution (n_x, n_y, n_z) and spacing $(\Delta x, \Delta y, \Delta z)$. We view this as a set of voxels, so the volume has sides of lengths $n_x\Delta x$, $n_y\Delta y$, and $n_z\Delta z$. The existing hardware requires the texture-map resolutions to be powers of two. Therefore, let N_x, N_y and N_z be the least powers of two that are at least as great, respectively, as n_x, n_y and n_z. We also define

$$L_x = N_x\Delta x,\ L_y = N_y\Delta y,\ L_z = N_z\Delta z.$$

In its initial orientation, we want the texture coordinates of the point $(-\frac{1}{2}n_x\Delta x, -\frac{1}{2}n_y\Delta y, -\frac{1}{2}n_z\Delta z)$ to be $(0, 0, 0)$. Similarly we want the texture coordinates of $(\frac{1}{2}n_x\Delta x, -\frac{1}{2}n_y\Delta y, -\frac{1}{2}n_z\Delta z)$ to be $(n_x/N_x,\ 0,\ 0)$, and so on for other corners of the volume. The constraints are satisfied by the functions

$$\begin{aligned} s(x) &= (x + \tfrac{1}{2}n_x\Delta x)/(N_x\Delta x) \\ t(y) &= (y + \tfrac{1}{2}n_y\Delta y)/(N_y\Delta y) \\ r(z) &= (z + \tfrac{1}{2}n_z\Delta z)/(N_z\Delta z) \end{aligned} \tag{1}$$

Since the corners of the bounding cube are $(\pm\frac{1}{2}d, \pm\frac{1}{2}d, \pm\frac{1}{2}d)$, it follows that we want to assign texture coordinates to *these* corners by evaluating s, t, and r in Eq. 1 at $\pm\frac{1}{2}d$.

We can represent the required 3D transformation from (x, y, z) to (s, t, r) as a combination of scales and translation. Let matrix D denote the uniform scale by d, let matrix S denote the nonuniform scale by $(d/L_x,\ d/L_y,\ d/L_z)$ (the reason for two scales will become evident later), and let matrix T denote the translation by $(\frac{1}{2}n_x/N_x,\ \frac{1}{2}n_y/N_y,\ \frac{1}{2}n_z/N_z)$. Then

$$(s, t, r) = (x, y, z)\, D^{-1}\, S\, T \tag{2}$$

The texture map has resolution (N_x, N_y, N_z).

Once the texture coordinates for the corners of the bounding cube have been found, those for the squares that slice up the cube are found easily by interpolation in z; only the r coordinates are affected.

3.2.2 Arbitrary Viewing Angles

From the description in Section 3.2.1, the volume can be correctly rendered if it is being viewed "from head-on", in its original orientation. (Observe that n_z ordinary 2D textures could be used to do this.) However, to view the volume from an arbitrary angle requires a 3D texture map, to correctly calculate the intersection of the rendered squares with the 3D volume. To render the volume from a rotated viewpoint, we keep the bounding cube stationary in screen space,

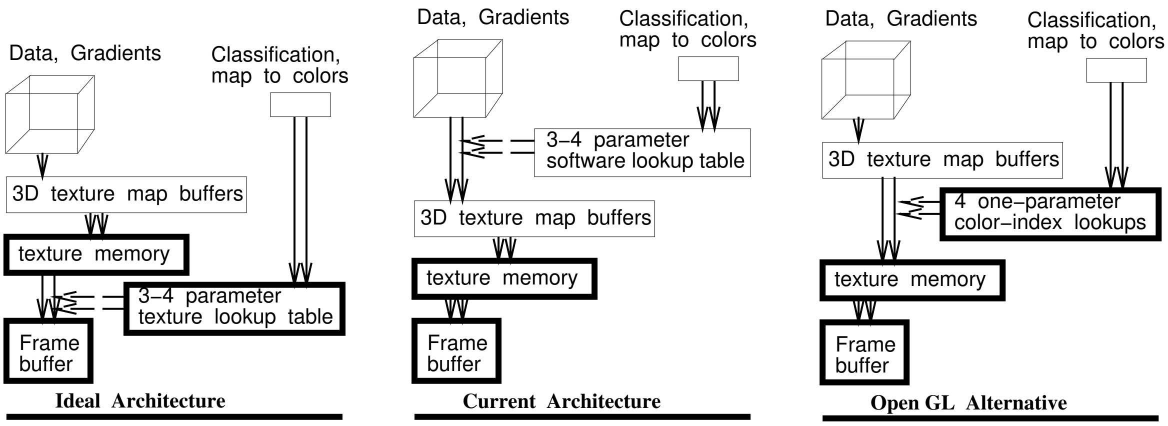


Figure 3: Alternatives for use of 3D texture maps. Heavy boxes denote special hardware; light boxes are software tables. Double arrows indicate operations needed for each new viewpoint. Single arrows denote occasional operations. Trilinear interpolation applies to contents of texture memory. Ideal architecture would apply 3-4 parameter texture lookup table to the interpolated values. Reality engine offers 4 one-parameter texture lookup tables, which suffice only for rendering without directional lighting.

and instead rotate the texture-space coordinates of the corners of the squares that slice up the bounding cube (see right half of Figure 1).

This can be done in program code using standard matrix multiplication techniques. Alternatively one can use the texture matrix, which is part of the graphics system. This matrix is just like a viewing matrix: it transforms texture coordinates before they actually are used. In this way the programmer gives the same texture coordinates at geometry vertices no matter what the orientation and the texture matrix does the rotation. In either method, the CPU overhead is small.

The required texture-space transformation is obtained simply by inserting the inverse of the client's rotation matrix R into the transformation of Equation 2, as follows:

$$\begin{aligned} (s,t,r) &= (x,y,z)\, R^{-1}\, D^{-1}\, S\, T \\ &= (x,y,z)\, D^{-1}\, R^{-1}\, S\, T \end{aligned} \tag{3}$$

using the fact that uniform scaling commutes with rotation. Evaluating (s,t,r) at the corners of the bounding cube (i.e., $(\pm\frac{1}{2}d,\ \pm\frac{1}{2}d,\ \pm\frac{1}{2}d)$) can be accomplished by applying the *texture-space* transformation $R^{-1}\, S\, T$ to $(\pm\frac{1}{2},\ \pm\frac{1}{2},\ \pm\frac{1}{2})$. The graphics library calls, while in texture mode, are therefore the given translate T, nonuniform scale S, as defined before Equation 2, following by negated, reversed order, rotation calls as specified by the client.

The mapping from world space to texture space is linear, therefore any rotation of it is also linear. Trilinear interpolation (done in hardware, in our case) of *linear* functions commutes with rotation. Therefore the image of the volume is not deformed. Care must be taken however with regards to the order in which scaling and rotation operations are performed on the texture coordinates. If a volume does not have the same Δ's in each dimension, matrix S specifies a non-uniform scale, which does not commute with rotation. The wrong order leads to shearing distortion. When transforming the texture coordinates, the correct order of operations is rotation, then scaling. More details and examples appear in Appendix A.

3.2.3 Planar Regions Outside the Volume

One issue is how to deal with rendering regions of the planes that lie outside the volume. Recall that the planes always remain parallel to the projection plane, and extend to the bounding cube, but the image of the volume within them rotates. Previous methods "hand-clipped" planes by intersecting them with the faces of the volume, producing a 3–6 sided polygon. The solution we chose uses clipping planes. We can position six programmable world-space clipping planes (available in Silicon Graphics workstations) at the six faces of the volume, to clip out any parts of the rendered squares that lie outside the volume. Clipping planes may also be controlled by the client to restrict the portion of the volume to be drawn.

3.2.4 Number of Planes

The default number of planes is chosen as $d/\Delta z$, so that when viewed straight on, each data point is sampled by one plane. For example, a 64^3 volume with uniform Δ's would get 110 planes. The rendered images are more accurate as more planes are used. With few planes, artifacts may be noticeable where a clipped edge of a rendered square is visible. We usually chose two to four times the default, as the cost of additional rendered planes is a very minor factor with directional lighting.

3.3 Hardware Texture Map Issues

Several possible organizations of a volume rendering system based on 3D texture maps in hardware are shown in Figure 3. Ideal texture-mapping hardware would be able to interpolate data and gradient, *then* map to color ("color" includes opacity in this discussion). This produces the greatest accuracy in the presence of nonlinear mappings from data to color [WVG94]. For such interpolation to make sense, the texture memory must be a continuous function of the data and gradient values.[2]

For rendering *without* directional lighting, the gradient is irrelevant, and a one-parameter texture lookup table, as offered in the SGI Reality Engine, suffices. In this case, Voltx realizes the "ideal" architecture. But such images have limited usefulness. One possible application is simulation of x-ray views (Figure 6).

In the more common case where directional lighting is desired, Voltx realizes the architecture in the middle of Figure 3. In this case, and in the Open GL alternative, the texture memory contains color, computed independently at each voxel of the volume. Buffers contain discontinuously encoded representations of the gradient, as described in Section 3.1.3. (In the figure, the "Data, Gradient" are so encoded in the "current" architecture; this one-time operation is not shown. The buffers in the middle section are maintained by the SGI GL library, and contain color. The buffer shown in the "Open GL alternative" is encoded and maintained by the application program.) Thus the hardware interpolates color, rather than data. This leads to inaccuracies in the presence of nonlinear mappings from data to color [WVG94]. The inaccuracies are more pronounced in close-up views. The 3–4 parameter texture lookup table needed to realize the "ideal" architecture seems unlikely to be economically justified for commercial systems in the foreseeable future.

4 Comparison to Shear-Warp Factorization

The Voltx method has certain similarities to the shear-warp technique, in that the volume is represented as a series of parallel planes for rendering purposes. In the shear-warp technique, those planes correspond to slices of the original volume. The *principal* slices are used, which are those most nearly orthogonal to the sight line. In general, they are oblique to the line of sight. The *principal face* is the principal slice closest to the viewer. In the Voltx method the planes are parallel to the projection plane, and are, in general, oblique to the volume.

The shear-warp method can be viewed as a form of ray-casting in which the rays are sparse, and arranged in a parallelogram pattern that corresponds to the vertices in the principal face of the volume (Figure 4, left), rather than the usual square pattern of pixels. Values computed on these obliquely arranged rays in the "shear" phase are

[2] An attempt to encode data and gradient discontinuously into one parameter, and sample the texture memory by the "nearest neighbor" method failed on the SGI Reality Engine. Following our problem report, the new SGI documentation states that the "nearest neighbor" method is only approximate. Due to inaccuracy of a filter, values that do not correspond to any neighbor may be delivered. This makes it impossible to use a discontinuous texture lookup table to perform arbitrary decodings of the texture map.

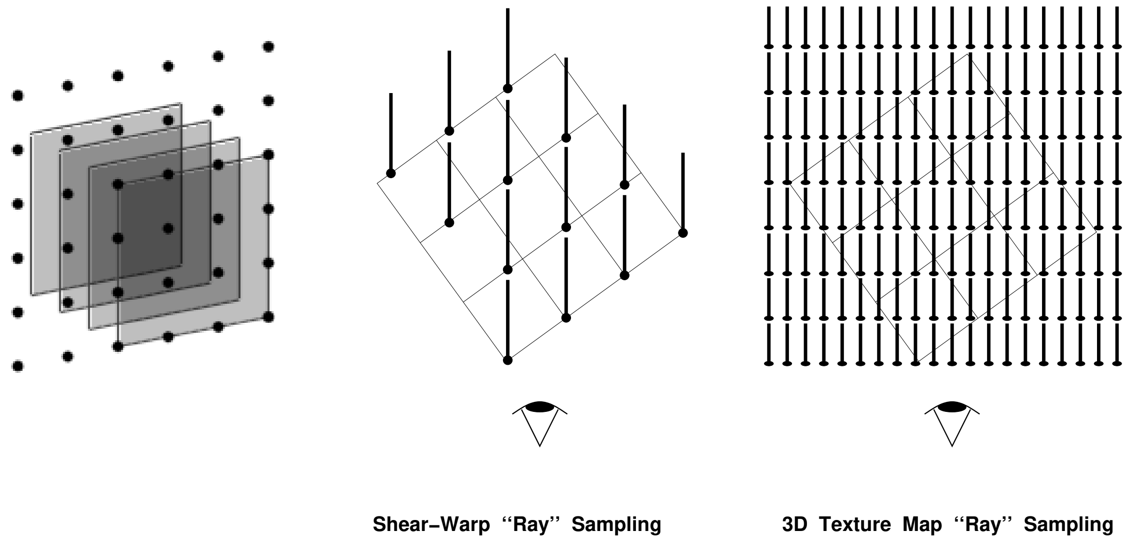


Figure 4: Shear-warp and volume texture methods are equivalent to certain ray-casting policies, as discussed in Section 4. The volume is $4 \times 4 \times 4$. Screen-z is out of the page on the left, where the volume is represented by its principal slices. Screen-z is down in the center and right, where the principal slices slant up to the right.

---

interpolated at screen pixels in the “warp” phase. Figure 4, center, shows how shear-warp “rays” sample the volume. The dots represent points sampled, and the lines going up from the dots indicate where the volume is assumed to have the sampled value for color/opacity integration purposes. Figure 4, right, shows the comparable situation for Voltx, assuming 20 pixels and 8 rendered planes. In effect, a ray is shot from each pixel, and the sample is computed by trilinear interpolation.

Lacroute and Levoy have reported on a fast shear-warp implementation [LL94], a limited version of which has been distributed. It will be called *LL* in this discussion. It is apparent from their paper and their code that speed was a major design objective of LL, and they have succeeded admirably in attaining this objective. We have made comparison tests of the same volume data on LL and Voltx. The general rule is that LL is somewhat faster, while Voltx is less prone to artifacts. Some of the issues are discussed below.

Both methods use planes through the volume, and increasing the number of planes improves the resolution. In Voltx the number of planes is a minor cost consideration and can be adjusted at run time, as it is independent of the volume. In LL, the number of planes is fixed by the resolution of the volume. To increase it requires resampling the volume in a preprocessing step, and the increased resolution has a substantial impact on performance. This appears to be an implementation choice, not an inherent property of shear-warp.

Both methods currently compute color (“color” includes opacity in this discussion) independently at each voxel of the volume, then interpolate color from those points. This leads to inaccuracies in the presence of nonlinear mappings from data to color, which are more pronounced in close-up views, as discussed in Section 3.3. The shear-warp technique, being entirely in software, can interpolate data and gradient, then map to color.

Smoothing out the transfer function was also recommended as a way to improve the shear-warp image quality, but such a step must bring image *accuracy* into question. Actually, Voltx always uses discontinuous transfer functions, as the each voxel is always classified as a single material. This can cause aliasing artifacts, but they are not noticeably sensitive to the orientation of the volume. Lacroute and Levoy reported that artifacts increase for LL as the rotation approaches 45 degrees.

5 Results

Rendering times for data with various sizes are summarized in Figure 5. The rendering was done on a Silicon Graphics Reality Engine II with two megabytes of 3D texture-map hardware and one raster manager. Classification time includes material classification, and quantizing the voxel gradients, and is based on one processor. This task is needed only once per run, as a rule. Rendering time includes updating the lookup table and rendering with new rotation or new lighting model for material definition. Times are real elapsed seconds, rather than CPU seconds, because CPU times do not reflect delays due to the graphics pipeline, which is heavily used. For comparison, the last column shows times without directional lighting, which are about 10 times faster.

Figures 6 and 8 (color plate) present a sample rendering of data sets listed in Figure 5. We used 1000 planes to render all volumes except Hipip, (High Potential Iron Protein) for which we used 128. ("Planes" were discussed in Section 3.2.) All images were rendered at 600×600 pixels, with four 8-bit color channels.

We found that times can be reduced by a factor of 1.2 to 2.5 by using four processors. These disappointing speed-ups are due to the fact that the graphics pipeline must be used serially, and it comprises a large fraction of the elapsed time. The GL interface requires redundant copying and processing of of 3D texture maps. For example, on the *MR brain* data set, 8 seconds are consumed in defining the 32 texture maps with `texdef3d`, using one processor. Open GL offers a different interface (`glTexImage3DEXT` with `COLOR_INDEX` format, and mapping tables for each of R,G,B,A), but our test showed that it still took 8 seconds to process 32 texture maps.

However, perspective projections incur no observable time penalty, in contrast to shear-warp techniques. They are essential for comfortable stereographic viewing. Also a pair of stereo images often requires much less than twice the time of a single view, as one pass through the texture loading suffices.

6 Conclusions and Future Directions

Volume texturing is a fast and simple method for direct volume rendering of rectilinear volumes available to those with appropriate hardware. With a directional light model, images have very good quality. Our implementation uses either orthographic or perspective projection, and supports stereo viewing (Figure 6 and color plate Figure 8).

By turning over many tasks to the hardware and libraries, we have simplified the programming task, but have given up on many optimization opportunities. On the other hand, with directionally shaded 3D texture-map images, the best-quality options consume only a little more time than the cheapest, so you might as well use them.

Lacroute and Levoy describe several optimizations that greatly increase the performance of LL. Their best reported times depended heavily on: (1) skipping over low-opacity (.05) voxels and nearly opaque (.95) pixels, (2) monochrome imaging, and (3) the use of orthographic, rather than perspective, projection. These practices compromise image quality under some circumstances, but they are not inherent parts of shear-warp. The fact that LL software competes with hardware-dependent Voltx calls into question the value of hardware 3D texture maps, at least as offered in the Reality Engine, for this application. (The fact that hardware designs cannot be fixed or modified by end-users is another practical consideration.)

While the graphics library and hardware handle many implementation details that would otherwise require hand-coding, there is some possibility that a software implementation of the same transformations could be competitive through increased ability to use parallelism. This would permit improved accuracy by interpolating data and gradients, rather than color, as discussed in Section 4. This is one topic for future investigation.

A related question that deserves investigation is whether the shear-warp technique can be improved in accuracy by interpolating data and gradients, rather than color, without incurring too great a loss of time.

Acknowledgements Funds for the support of this study have been allocated by a cooperative agreement with NASA-Ames Research Center, Moffett Field, California, under Interchange No. NAG2-991, and by the National Science Foundation, Grant Number ASC-9102497, and Grant Number CCR-9503829.

Data Set	Size (voxels)	Texture Maps Needed	Wall-Clock Seconds					
			Class-ific'n	Render, 1 proc		Render, 4 procs		1 proc
				1 view	stereo	1 view	stereo	unshaded
Hipip	$64\times64\times64$	1	1	1.0	1.4	.8	1.2	.16
MR brain	$256\times256\times109$	32	61	13.3	17.4	5.0	9.5	.70
Engine	$256\times256\times110$	32	62	13.5	16.9	4.7	8.8	1.10
CT head	$256\times256\times113$	32	77	13.5	17.0	5.3	9.5	1.33

Figure 5: Rendering and classification times for Voltx. Times are for a Reality Engine II, with one raster manager, using one or four 150-MHz processors. Rendered images are 600x600 pixels. See Section 5 for discussion.

References

[Ake93] Kurt Akeley. RealityEngine graphics. *Computer Graphics (ACM Siggraph Proceedings)*, 27:109–116, 1993.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *ACM Symposium on Volume Visualization*, pp. 91–98, Washington, 1994.

[CN93] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill, N. C., 1993.

[CT82] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, 1982.

[CU92] G. G. Cameron and P. E. Undrill. Rendering volumetric medical image data on a SIMD-architecture computer. In *Third Eurographics Workshop on Rendering*, Bristol, UK, MAy 1992.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):65–74, 1988.

[DH92] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. In *ACM Workshop on Volume Visualization*, pp. 91–98, Boston, 1992.

[GL94] S. Guan and R. G. Lipes. Innovative volume rendering using 3D texture mapping. In *Image Capture, Formatting and Display*. SPIE 2164, 1994.

[Kru90] Wolfgang Krueger. Volume rendering and data feature enhancement. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):21 – 26, 1990.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, March 1988.

[Lev90a] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[Lev90b] Marc Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications*, 10(2):33–40, March 1990.

[Lev92] Marc Levoy. Volume rendering using the fourier projection-slice theorem. In *Proceedings of Graphics Interface '92*, Vancouver, B.C., 1992. Also Stanford University Technical Report CSL-TR-92-521.

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):285–288, 1991.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics (ACM Siggraph Proceedings)*, pp. 451–458, 1994.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *ACM Workshop on Volume Visualization*, 24(5):27–33, 1990.

[Sab88] Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, 22(4):51–58, 1988.

[SS92] Peter Schroeder and Gordon Stoll. Data parallel volume rendering as line drawing. In *ACM Workshop on Volume Visualization*, pp. 25–32, Boston, 1992.

[ST90] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, December 1990.

[TM67] Kenneth E. Torrance and Sparrow E. M. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, 1967.

[UK88] Craig Upson and Michael Keeler. The v-buffer: Visible volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):59–64, 1988.

[VGW93] Allen Van Gelder and Jane Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. In *IEEE Visualization '93*, 1993. (extended abstract) Also, Univ. of California technical report UCSC-CRL-93-02.

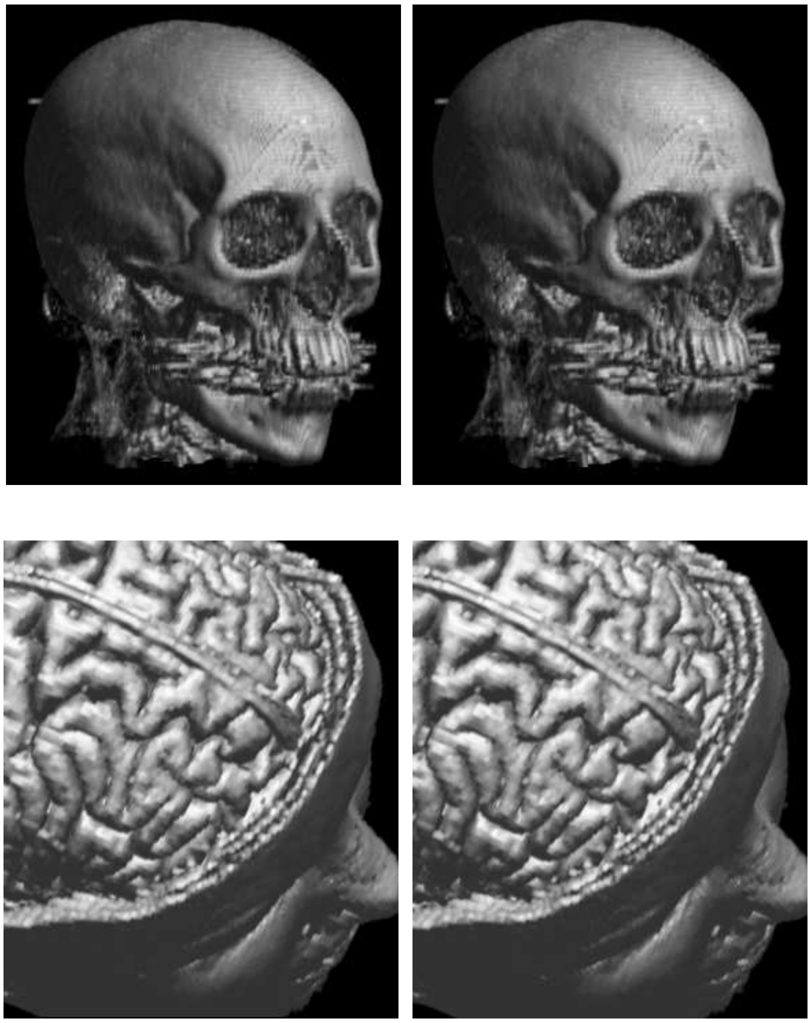
[Wes90] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 24(4):367–76, 1990.

[Wil92] Peter Williams. Interactive splatting of nonrectilinear volumes. In *IEEE Visualization '92*, pp. 37–44, 1992.

[WVG91] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):275–284, 1991.

[WVG94] Jane Wilhelms and Allen Van Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *ACM Symposium on Volume Visualization*, Washington, 1994.

[WVGW94] Orion Wilson, Allen Van Gelder, and Jane Wilhelms. Direct volume rendering via 3D textures. Technical Report UCSC-CRL-94-19, Univ. of California, Santa Cruz, 1994.



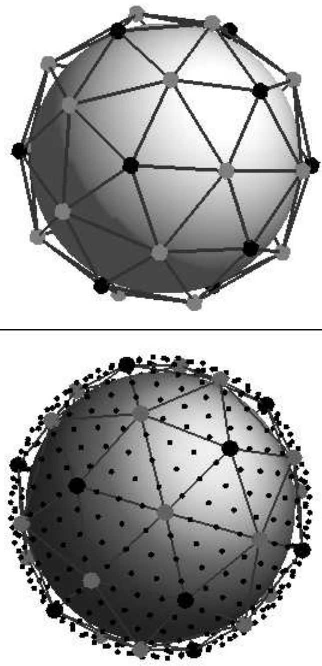


Figure 6: Autostereograms of CT head and MRI head (left and center), to be viewed without instruments at a distance of 1–2 feet. Try to look at a point about three times the distance of the page, 3–6 feet away. Raise page into line of sight, seeing 3 images. Middle image should fuse, giving stereo effect. Blinking the eyes might help. Alternatively, place a card or piece of stiff paper between images, and extending towards the eyes; page should still be about one foot away. Right upper image is x-ray-style image of CT head. Volume has opacity only, and light is emitted from back plane. CT head is classified as bone on a ramp from 0% at 90 to 100% at 130 Hounsfield units. MRI uses binary classification at threshold 550. For a new viewpoint, stereo images required 18 seconds per pair; x-ray image required 1 second. All images required 32 texture maps.

Figure 7: Vertices of regular icosahedron (black) and its dual dodecahedron (gray) are triangulated into 60 triangles (right, center), which are recursively subdivided (right, lower) to give points on the unit sphere that serve as quantized normal vectors.

Color plates omitted; available separately from authors.

Figure 8: Autostereogram of Hipip (left, center, see Fig. 6 for viewing hints), and image of Engine (right).

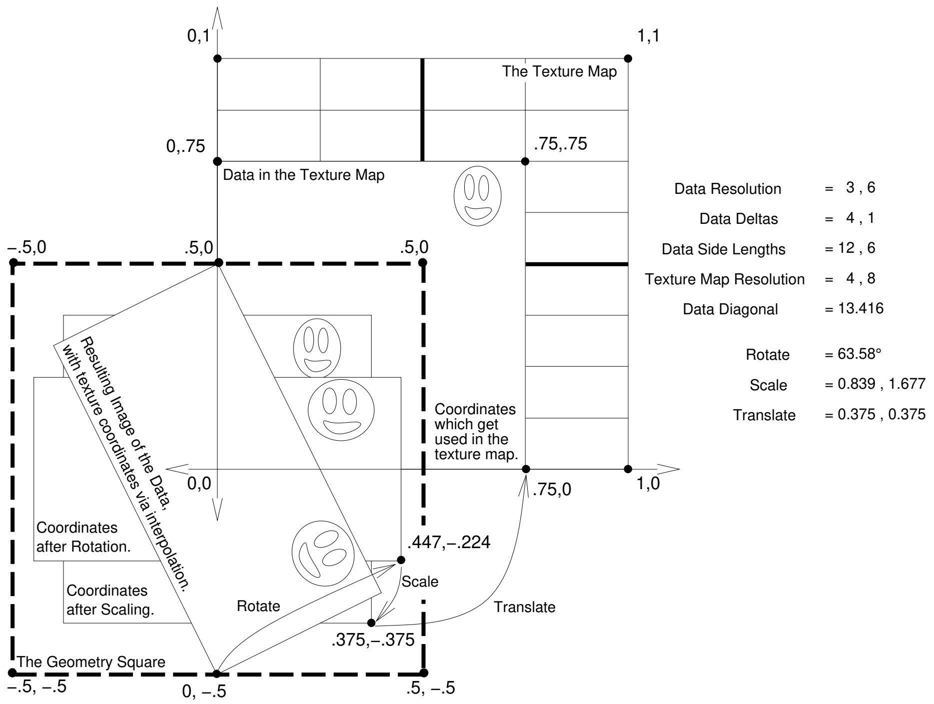


Figure 9: 2-D Example of Texture Coordinate transformations. Coordinates are specified at the corners of the Geometry Square, then by a series of matrix transformations are mapped to the texture map.

Appendix A Texture Coordinate Examples

We thank Orion Wilson for helping to work out these examples and for drawing the accompanying picture.

Example A.1: Assume the volume has resolution $(3, 4, 4)$ and spacing $(\Delta x, \Delta y, \Delta z) = (1, 3, 1)$. The diagonal is $d = 13$. Then $(N_x, N_y, N_z) = (4, 4, 4)$. We get

$$\begin{aligned} s(-d/2) &= -1.25 \\ s(d/2) &= 2.00 \\ t(-d/2) &= -0.041667 \\ t(d/2) &= 1.041667 \\ r(-d/2) &= -1.125 \\ r(d/2) &= 2.125 \end{aligned}$$

□

Example A.2: This 2-D example works thru a texture-matrix calculation to show that the pre-matrix texture coordinates for a given corner of the data rectangle get transformed to the correct final texture coordinates.

Assume the data has resolution $(n_x, n_y) = (3, 6)$ and spacing $(\Delta x, \Delta y) = (4, 1)$. Then the diagonal is $d = 13.42$, and the texture map resolution $(N_x, N_y, N_z) = (4, 8)$. Assume the viewer has rotated the volume $\Theta_z = -63.58°$. (In case of multiple rotations by the viewer, their order must be reversed here). The parameters of the texture matrices are:

$$Rotate \quad = \quad (-\Theta_z, \texttt{'z'}) = (63.58°).$$

$$
\begin{aligned}
Scale &= \left(\frac{d}{L_x}, \frac{d}{L_y}\right) = \left(\frac{13.42}{4}, \frac{13.42}{8}\right) = (.839,\ 1.677). \\
Translate &= \left(\frac{n_x}{2N_x}, \frac{n_y}{2N_y}\right) = \left(\frac{3}{8}, \frac{3}{8}\right) = (.375,\ .375).
\end{aligned}
$$

We choose the pre-matrix texture coordinates at the corners of the planes to be $(-.5, -.5), \ldots, (.5, .5)$, as explained in connection with Equation 3.

For example, the interpolated "raw" texture coordinates of the lower right corner of the rotated data set are $(0, -.5)$. These are transformed in hardware by the texture matrices, as follows:

$$
\begin{aligned}
Original\ Coordinates &= (0, -.5) \\
After\ Rotation &= (.447, -.224) \\
After\ Scaling &= (.375,\ -.375) \\
After\ Translation &= (.75,\ 0)
\end{aligned}
$$

which agrees with the corresponding texel in the texture map, whose coordinates are:

$$\left(\frac{3}{4}, \frac{0}{8}\right) = (.75,\ 0).$$

□

Appendix B Hardware Texture Map Details

This section discusses some detailed choices to be made in the use of the hardware texture map feature.

Appendix B.1 Interpolation Options

The Reality Engine can be set to sample the texture map either using a nearest neighbor method (so-called *point* interpolation)[3] or a trilinear interpolation. The trilinear method is about 50% slower than point interpolation, but gives smoother-appearing images. Because the actual rendering time is dwarfed by the time needed to recalculate the texture map after a rotation, we normally choose the superior image.

Appendix B.2 Rounding Errors

In creating the texture map, it is possible to encounter significant rounding error problems when using an eight-bit rather than a twelve-bit texture map. However, rendering with an eight-bit texture map is about twice as fast as with a twelve-bit map, the memory requirement is half as much, and the problem only occurs under certain conditions. Consequently, it is sometimes desirable to use an eight-bit map.

For example, if the color contribution to a plane is small, rounding error can make it appear zero. This can be seen as blank areas in the final image when the number of slices is large enough so that the color contributions of many single pixels in each plane are rounded to zero. If the texture map has eight bits per color channel, then rounding errors are up to one half of $1/256$. In this case, rounding-error may become detrimental if several hundred slices are used. The problem is exacerbated if the data set in question is largely homogeneous, because in a region of constant value, the pixels are all rounded in the same direction and the error accumulates. In non-homogeneous data, rounding-errors tend to cancel each other out.

The problem becomes negligible (for hardware texture maps of a size available now) when the texture map is given more bits per channel, simply reducing the magnitude of the errors. Twelve-bit textures (supported in the Reality

[3] Actually, the "nearest neighbor" method is only approximate. Due to inaccuracy of a filter, values that do not correspond to any neighbor may be delivered. This makes it impossible to use a discontinuous "texture lookup table" to perform arbitrary decodings of the texture map.

Engine) let rounding-error be only one half of $1/4096$. This is quite sufficient to make an acceptable texture map for distributing the data even over several thousand slices. An accurate final image rarely requires more slices than 3.5 $(2 \times \sqrt{3})$ times the maximum resolution in the volume.

Besides rounding-errors in the storage of the texture, we were also concerned that errors would accumulate during the blending and compositing stage of rendering, but this has not been noticed.