Direct Volume Rendering via 3D Textures

Orion Wilson, Allen Van Gelder, Jane Wilhelms
Computer and Information Sciences
University of California, Santa Cruz
UCSC-CRL-94-19
Baskin Center for Computer Engineering and Information Sciences
University of California, Santa Cruz 95064
moria@cs.ucsc.edu avg@cs.ucsc.edu wilhelms@cs.ucsc.edu

June 29, 1994

Abstract

The advent of very fast texture mapping hardware in modern graphics workstations has warranted research into rendering techniques that use texture mapping to full advantage. We have developed a new and easy to implement method for direct volume rendering that produces high-quality images at speeds approaching two orders of magnitude faster than existing techniques, on workstations with hardware support for three-dimensional texture maps. A rectilinear data set is converted into a three-dimensional texture map containing color and opacity information. In the rendering phase, the texture map is then applied to a stack of parallel planes, which effectively cut the texture into many slices. The slices are composited to form an image of the original data set. This paper describes the theory and implementation of this technique.

Keywords: Computer Graphics, Scientific Visualization, 3D Texture Mapping, Direct Volume Rendering.

1 Overview

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, Lev90, MHC90, Wes90, LH91, Lev92]. Some methods use hardware-assisted Gouraud-shading capabilities to speed rendering, by calculating the projections of volume regions and treating them as polygons [ST90, LH91, WVG91, Wil92, VGW93].

In this paper, we present a new method that takes advantage of a more sophisticated capability of some new graphics computers: hardware-assisted 3D texture mapping. We have found this method to be significantly faster than the hardware-assisted Gouraud-shading method (which we call *coherent projection*), easier to implement, and producing images of comparable or better quality. We have found its main limitations to be restriction to rectilinear volumes, and the small size of 3D texture maps allowed by present hardware. The former limitation may be inherent in the method, but the latter can be dealt with as described below. Akeley mentions the possibility of using a 3D texture map for volume rendering in his report on the design of the Silicon Graphics Reality Engine [Ake93]. Independently from the work presented here, three other papers have been written describing use of 3D texture mapping hardware for direct volume rendering. Cullip and Neumann sketch two approaches, which they call object space 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 require substantial programming to "hand compute" transformations, clipping, and the like, details of which are omitted. This paper describes how most of this programming can be eliminated by use of graphics library procedures to perform texture space transformations and set clipping planes.

In our new method, we first convert the data volume to a 3D texture map, using a one-time pass through a transfer function that maps scalar data values to appropriate color (red, green, and blue) and opacity. Color and opacity are modified to account for depth integration (taking into account the desired number of planar slices) and stored in a 3D texture map. This texture map is then applied to many parallel planes, which make slices through the texture. Each plane is rendered as a square, so texture coordinates need be specified only at its four corners.

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 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.

The squares are parallel to the projection plane in screen space, while the 3D texture map can be oriented as the user 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 a 3D texture map, the Reality Engine's specialized texture hardware can perform the slice rendering and compositing very quickly. For example, as shown in Table 1 rendering is 14 to 140 times faster than coherent projection, which is itself considered a fast method [WVG91]. The conversion into the texture map need only be done once, and then the volume can be rendered arbitrarily scaled, translated, and rotated.

Resulting images have definition and clarity comparable to other rendering methods. 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.

2 Detailed Description of Method

The two major steps in this method are: first, create the texture map; and second, render the slices.

2.1 Creating the Texture Map

We interpret each plane to be rendered as being at the center of a slice (with thickness) through the data, like a slice of bread. This means that each plane must contribute the color intensity and opacity due to one such slice. The thickness of every slice, Δ, is just the total distance covered by the stack of planes divided by the number of planes. Knowing Δ, the color emission per unit distance E, and opacity per unit distance A_1 assigned to the data at each point by the *transfer function*, we compute the color C and opacity A that must go into the texture map. For a detailed explanation of this computation, see [WVG91, VGW93], but briefly the formulas are:

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

where E and C have three components, *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 slice. The resulting color intensities 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 for storage in the texture map.

After each voxel in the data has been assigned a color C and an opacity A, an eight- or twelve-bit texture map entry, called a *texel*, is generated and stored. This process needs to be repeated whenever the transfer function is changed or the number of slices is changed, but does not need to be repeated for different viewing transformations such as rotation or scaling.

2.1.1 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 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.

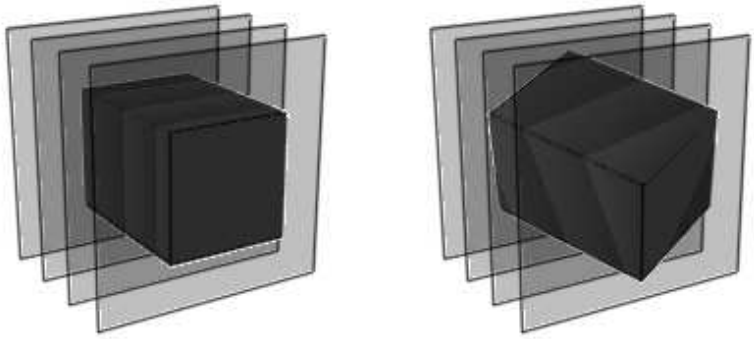

Figure 1: Slices through the volume in original orientation (left), and rotated (right). The viewer's line of sight is orthogonal to the slices. In both orientations, texture coordinates are assigned to corners of the square slices in such a way that they interpolate into the range $[0, 1.0]$ exactly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

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.

2.2 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 slices through the texture, like a sliced loaf of bread (see Figure 1). The squares are drawn from back to front. The orientation of the squares must remain fixed in screen space, parallel to the projection plane, with their normals towards the viewer. Otherwise, if the squares were viewed obliquely, their separation along the line of sight would not be Δ, and the color and opacity obtained from the texture map would not be correct. Because color and opacity are nonlinear functions of Δ, it would be impossible to adjust for this discrepancy.

2.2.1 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 $n_z \Delta z$. In its initial orientation, we want the texture coordinates of the point $(-n_x \Delta x/2, -n_y \Delta y/2, -n_z \Delta z/2)$ to be $(0, 0, 0)$. Similarly we want the texture coordinates of $(n_x \Delta x/2, -n_y \Delta y/2, -n_z \Delta z/2)$ to be $(1, 0, 0)$, and so on for other corners of the volume. Since the corners of the bounding cube are $(\pm d/2, \pm d/2, \pm d/2)$, it follows that we want to assign texture coordinates to *these* corners as follows:

- The s texture coordinate for lower-in-x corners is given by

$$s(-d/2) = \tfrac{1}{2}\left(1 - \frac{d}{n_x \Delta x}\right)$$

- The s texture coordinate for upper-in-x corners is given by

$$s(d/2) = \tfrac{1}{2}\left(1 + \frac{d}{n_x \Delta x}\right)$$

  (See Figure 2 for an illustration of the geometry.)

- The same pattern is followed for the t texture coordinate, using $n_y \Delta y$, and distinguishing between lower-in-y and upper-in-y.

- The same pattern is followed for the r texture coordinate, using $n_z \Delta z$, and distinguishing between lower-in-z and upper-in-z.

If we could choose a texture map of resolution (n_x, n_y, n_z) we would be done, but the existing system essentially requires the texture map resolutions to be powers of two.

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

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

We now require the upper-in-x face of the volume to have texture coordinate $s = n_x/N_x$ (instead of 1). The upper-in-y and upper-in-z faces are similarly modified for t and r. The required equation for $s(x)$ becomes

$$s(x) = \frac{x + \frac{1}{2} n_x \Delta x}{N_x \Delta x}$$

in that this function evaluates to 0 and n_x/N_x, respectively, at the corners of the embedded volume, which are located at $x = \pm\frac{1}{2} n_x \Delta x$. Equations for t and r are similar, replacing x by y and z, respectively.

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 $(n_x/2N_x,\ n_y/2N_y,\ n_z/2N_z)$. Then

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

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.

Example 2.1: 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

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

□

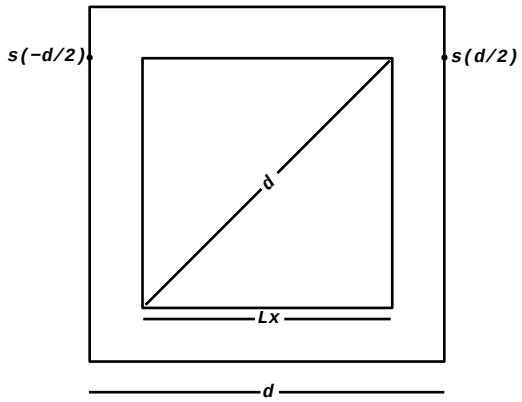

Figure 2: Geometry of Texture Coordinates. This illustrates an orthographic projection down on a 3-D volume of size (L_x, L_y, L_z) and the geometry squares which contain the texture-mapped image of the volume. Sides L_y and L_z are not labeled, and the long diagonal through the volume is length $d = (L_x^2 + L_y^2 + L_z^2)^{\frac{1}{2}}$.

2.2.2 Arbitrary Viewing Angles

From the description in Section 2.2.1, 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 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 1, as follows:

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

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 $T\,S\,R^{-1}$ 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 1, 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 $\Delta's$ in each dimension, the equations for s, t, and r above result in a non-uniform scaling of the texture coordinates; they are stretched in one dimension more than another. In this situation, performing rotation and scaling in the wrong order leads to shearing distortion. When transforming the texture coordinates, the correct order of operations is rotation, then scaling. This seems counter-intuitive to the way object

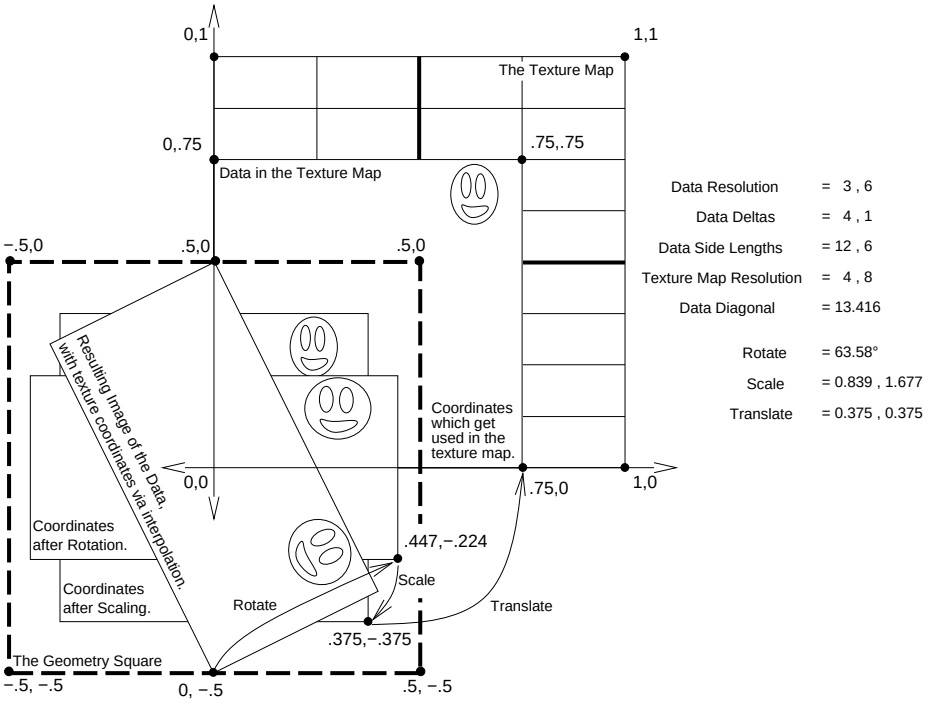

Figure 3: 2-D Example of Texture Coordinate transformations. Coordinates are specified at the corners of the Geometry Square, then by a series of matrix transformations are mapped to the texture map.

---

transformations are usually done, but the trick is that we are transforming the points which sample the texture map, not the texture map itself.

Example 2.2: 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. (See Figure 3.)

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:

$$\begin{aligned} \mathit{Rotate} &= (-\Theta_z, \mathbf{'z'}) = (63.58°). \\ \mathit{Scale} &= \left(\frac{d}{L_x}, \frac{d}{L_y}\right) = \left(\frac{13.42}{4}, \frac{13.42}{8}\right) = (.839,\ 1.677). \\ \mathit{Translate} &= \left(\frac{n_x}{2N_x}, \frac{n_y}{2N_y}\right) = \left(\frac{3}{8}, \frac{3}{8}\right) = (.375,\ .375). \end{aligned}$$

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 2.

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:

$$\mathit{Original\ Coordinates} = (0, -.5)$$

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

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

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

□

2.2.3 Planar Regions outside the Data Volume

One complexity 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, but the image of the volume within them rotates. The solution we chose uses clipping planes. Because the texture-rendered volume lines up with world-coordinates, we can position six programmable world-space clipping planes (available, in the Silicon Graphics line, on VGX graphics systems and up) to geometrically clip out any parts of the squares that lie outside the volume. In our case, this involves a small "gotcha". To orient the geometry planes parallel to the projection plane, we perform the inverse of the current world rotation before drawing them. Since the clip planes use whichever transformation matrix was active when they were defined, we need to define the clip planes before this inverse rotation. This way, the clip planes are defined in world space, line up with the rendered image of the volume, and properly clip away parts of the geometry planes which lie outside the volume. Clipping planes have two related advantages. First, rendering speed increases by about 5.6 times because texture coordinates are not calculated for invalid pixels. The expected speedup from this technique is $3\sqrt{3} = 5.20$.

The speedup varies because data sets of different sizes take advantage of clipping more or less than others. For example, a thin, column-shaped volume will clip more than a cube-shaped one. The second advantage is that the region enclosed by the clipping planes may be shrunk so that only a desired subregion of the volume is rendered. Another solution is to clip the planes to the volume yourself and then send the resulting polygons to the graphics system for rendering. Since the clipping planes are so efficient and easy to implement, we chose to use them.

2.2.4 Picking Number of Planes

The default number of planes is chosen so that when viewed straight on, each data point is sampled by one plane. Since the world-space distance covered from the first to last plane equals d, the length of the diagonal through the volume, the default number of planes is thus $d/\Delta z$. For example a 64^3 volume with $\Delta z = 1$ would get 110 planes. The rendered images look better as more planes are used. If only a few planes are used relative to the resolution of the data set, some of the sample points may not contribute to the final image at all.

Also, with few planes, artifacts are noticeable wherever a clipped edge within a rendered square is visible. (Clipping planes are rotated from world space to screen space by the same rotation as was applied to texture coordinates.) This is because the transition from inside the clipping region where color is being contributed to the image to outside the plane is abrupt, and the edge shows up. These problems are avoided by increasing the number of slices.

Data	Method	Slices	Bits	Interpolation	Setup	Rendering	cp/voltx	Total
Hipip 64^3	Voltx	220	12	Trilinear	5.0 sec	1.11 sec	20	6.6 sec
	Voltx	220	12	Point	5.0 sec	0.71 sec	31	5.7 sec
	Voltx	220	8	Point	4.5 sec	0.38 sec	58	4.9 sec
	Voltx	110	8	Point	4.5 sec	0.20 sec	112	4.7 sec
	Voltx	55	8	Point	4.5 sec	0.12 sec	192	4.6 sec
	CP	*na*				22.10 sec		22.1 sec

Table 1: Times for volume rendering options. (Times are elapsed seconds on a 150 MHz Reality Engine 2, Rendered images are 600x600 pixels, double-buffered.)

For autorotation, something in the neighborhood of 50 planes allows a fairly smooth realtime animation of textures sized 64^3, with images covering about 500×500 pixels. We obtained rendering speeds of about five frames per second. Having more planes, of course, slows down rendering.

2.2.5 Hardware Texture Map Issues

The Reality Engine can be set to sample the texture map either using a nearest neighbor method (so-called *point* interpolation) or a trilinear interpolation. The trilinear method is about 50% slower than point interpolation, but gives smoother-appearing images.

A final hardware consideration is that the texture memory is limited. Our machine allows a maximum size 3D texture map of two megabytes. This permits a 64^3 texture map with four channels and 12-bit texels. Larger volumes must be rendered by subdividing them into appropriately sized subvolumes and using this method on each of them in back-to-front order. The speed overhead incurred is about one tenth of a second per texture map to copy the texture from regular memory into texture memory.

3 Experimental Results

We compared 3D texture-mapped direct volume rendering using our program *voltx* under different user-settings, and also compared to coherent projection direct volume rendering [WVG91], a relatively fast direct volume rendering method for rectilinear volumes. The user-settings include number of slices (which can alternatively be set by specifying desired distance between planes), 8-bit or 12-bit texture maps, and point or trilinear interpolation.

Table 1 shows our results on the 64^3 Hipip data set[1]. As can be seen from the table, trilinear interpolation is about 50% slower than point interpolation, 8-bit textures are about twice as fast as 12-bit textures, and decreasing the number of slices by half about doubles the rendering speed. However, compared to coherent projection, any of the volume textured methods examined are admirably fast.

An equation approximating the time to render n planes is

$$seconds = n * .001455 * (1 + 1.06 * TwelveBit) * (1 + .63 * Trilinear)$$

where $TwelveBit$ and $Trilinear$ are booleans indicating that 12-bit textures are being used instead of 8-bit, and that Trilinear interpolation is being used instead of nearest neighbor.

[1] Hipip (High Potential Iron Protein) is from Louis Noodleman and David Case, Scripps Clinic, La Jolla, California.

The best quality image created in the table used enough slices to sample at least twice between each data point, with 12-bit textures and trilinear interpolation. Once the texture map was created and loaded, rendering was about 15 times faster than coherent projection. Guaranteeing one slice for each sample point encountered along any line of sight gives approximately a 100 times speed-up over coherent projection, using the faster 8-bit textures and point interpolation. Image quality shows very minimal deterioration. Even if the texture map must be recreated, because of a change in the transfer function mapping from data to colors, or a desire for faster or better quality images, the method is several times faster than regenerating coherent projection.

4 Discussion and Conclusions

Volume texturing is a fast and simple method for direct volume rendering of rectilinear volumes available to those with appropriate hardware. Images have very good quality. While our implementation presently uses either orthogonal or perspective projection, and stereo viewing can be added easily.

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. NCA2-430, and by the National Science Foundation, Grant Number ASC-9102497, and Grant Number CDA-9115268.

References

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

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. 1994. (submitted for publication).

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

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

[GL94] S. Guan and R. G. Lipes. Innovative volume rendering using 3d texture mapping. In *SPIE: Medical Imaging 1994: Images Captures, 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.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 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, July 1991.

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

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

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

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

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

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

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

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