Footprint Evaluation for Volume Rendering

Lee Westover
Numerical Design Limited
The University of North Carolina at Chapel Hill

ABSTRACT

This paper presents a forward mapping rendering algorithm to display regular volumetric grids that may not have the same spacings in the three grid directions. It takes advantage of the fact that convolution can be thought of as distributing energy from input samples into space. The renderer calculates an image plane footprint for each data sample and uses the footprint to spread the sample's energy onto the image plane. A result of the technique is that the forward mapping algorithm can support perspective without excessive cost, and support adaptive resampling of the three-dimensional data set during image generation.

KEYWORDS: 3D Image, Volume Rendering, Reconstruction, Algorithms.

INTRODUCTION

Volume rendering is the direct display of data sampled in three dimensions. There are two principle approaches to volume rendering: backward mapping algorithms that map the image plane onto the data by shooting rays from pixels into the data space, and forward mapping algorithms that map the data onto the image plane.

* Author's current address:
Sun Microsystems Inc
PO Box 13447
Research Triangle Park NC
27709

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This distinction principly manifests itself in how and when reconstruction of the three-dimensional signal is done. Convolution can be thought of as either generating an output sample from many input samples or as spreading one input sample to many output samples. Backward mapping algorithms typically reconstruct the signal at a point in space by looking at that point's nearest data samples and performing some type of interpolation. Forward mapping algorithms differ in that they incrementally reconstruct the original signal by spreading each data sample's energy into space.

Forward mapping algorithms are important because they are easily made parallel. Since each data sample only needs to know about a small surrounding neighborhood of other samples, shading and transforming can be done in parallel for sub-sections of the data. With today's parallel machines having limited local memory, this data distribution gets around the backward mapping problem of having the entire data set at each node.

The reconstruction step is the most complicated part of the algorithm. The renderer must determine the screen space contribution of each sample point to the final image. A brute force method would perform a one-dimensional integration of the reconstruction kernel for every pixel for every input sample. If the renderer can calculate the screen space extent of the kernel, the number of integrations reduces to the number of samples times the number of pixels that fall within the extent. However, this is still an enormous number of integrations.

In an orthographic view, the footprint of the projected reconstruction kernel for any sample is a constant except for a screen space offset. This allows the renderer to build a footprint function table once and use the table for all samples. Since the table is discrete, the renderer builds it on a fine grid to prevent artifacts. Even with this modification, the renderer must perform N^2 integrations of the kernel where N is the number of grid cells in each table dimension.

©1990

This paper presents an algorithm that allows the renderer to use a pre-computed footprint function table to build the view-transformed footprint table for a particular view. This pre-computed table is called the generic footprint table because the renderer uses it to calculate the view-transformed table for any particular view. The renderer needs to calculate two things to build the view-transformed table. First, the renderer computes the screen space extent of the projection of the reconstruction kernel. Second, the renderer computes a mapping of this extent to the extent that surrounds the pre-integrated footprint table. Then for each cell in the grid of the view-transformed table, the renderer maps the cell to the generic table and samples the generic table to find the cell's value. Once the renderer builds the view-transformed table, it can use the table for all input samples. The renderer centers the table at the sample's projected screen location and samples the table at the center of each pixel that falls within the table's extent.

PREVIOUS WORK

Researchers have investigated the volume rendering problem in the last few years and these algorithms can be divided along many lines. Blinn [2], Kajiya [7], VanHook [14], Levoy [9] and Sabella [12] describe methods of ray tracing volume densities with algorithms that map pixels onto the data by shooting rays into the data. Frieder [4], Lenz [8], Drebin [3], and Westover [15] use compositing techniques that map the data onto the image plane. Lorensen [10], Upson [13], and Gallagher [7] have investigated various methods of fitting surfaces into each data cell and then rendering the volume as surfaces.

Another distinction between algorithms is whether the original signal is reconstructed and shaded at points of interest or whether the original data samples are shaded and then the shaded volume is reconstructed to form an image. Since shading is typically a non-linear process, interpolating the shaded volume can be problematic due to the high frequencies introduced by the shading model. On the other hand, this method only shades true data samples. Interpolating first, then shading, introduces new data samples into the data set, but shading happens at exact query samples. An enhancement to the algorithm presented in this paper can support either approach.

Footprint determination has much in common with texture map sampling. It is, however, almost the exact opposite problem. In texture mapping, a pixel is mapped into texture space and then all texture samples that lie within the mapped pixel's footprint are weighted and accumulated to form the single texture color [6]. In volume rendering, the footprint is used to spread a single samples contribution onto every pixel that lies within the mapped voxel's footprint. In both cases, the mapping of a sample from one space into a second space forms an elliptical footprint in the second space.

RENDERING ALGORITHM

The algorithm discussed in this paper is a forward mapping algorithm that shades at input samples, and reconstructs a final image from the shaded volume. This work differs from the original algorithm, described in Westover [15], in four ways. First, the initial algorithm combined the reconstruction step and the visibility step at each voxel. The new algorithm performs reconstruction for all samples in a sheet, where a sheet is defined as a plane through the data that is most parallel to the image plane. Each voxel in a sheet is added to a sheet cash. When all the voxels on a sheet are processed, the sheet is matted into the working image. Second, the algorithm now uses a generalized shading model, Abram [1], that supports many shading techniques including the one from the original algorithm. Third, many of the details of how footprints are calculated and used has changed, as described below. Forth, the new footprint method will allow the algorithm to support both perspective and adaptive refinement.

The algorithm consists of four main parts: transforming, shading, reconstruction, and visibility. For the algorithm to run in parallel, it is critical that each step in the process uses only local information. The renderer processes a sample by transforming the sample from input $<i,j,k>$ grid space to $<x,y,z>$ screen space. It then shades the sample using some shading rule that uses local information. The shaded sample is a $<x,y,z,red,green,blue,\alpha>$ tuple. Next the renderer determines the portion of the image the sample can affect and adds the sample's contribution to the sheet accumulator. The determination of the footprint function, the sampling of the footprint function, and the spreading of the sample's contribution is called splatting. The efficient determination of the effect and an efficient application of the footprint function is the topic of this paper. When all the samples that lie in a sheet are processed, the renderer mattes the sheet accumulator to the working image using a compositing operator [11]. Once all samples are processed, the working image becomes the final image.

FOOTPRINT FUNCTION

The volume reconstruction equation for a regular array of density values is:

$$signal_{3D} =$$

$$\iiint h_V(u-x, v-y, w-z)\ \rho(x,y,z) \sum \delta\ (x,y,z)\ dudvdw$$

where $h_V()$ denotes the volume reconstruction kernel, ρ denotes the density function, $\sum\delta$ denotes the comb

function, and u,v,w are the coordinates of the kernel.

Moving the summation outside the integral and evaluating the integral at point <x, y, z> results in:

$$signal_{3D}(x,y,z) = \sum_{D \in Vol} h_V(x-D_x, y-D_y, z-D_z)\, \rho(D)$$

where D ranges over the input samples that lie within the range for which the kernel, $h_V()$ is non-zero, and D_x, D_y, and D_z are the screen space coordinates of the sample <D>.

Instead of considering how multiple samples contribute to a point, consider how a sample can contribute to many points in space. The contribution at a point <x, y, z> by a data sample <D> is:

$$contribution_D(x,y,z) = h_V(x-D_x, y-D_y, z-D_z)\, \rho(D)$$

Therefore, the renderer can treat each data sample individually and spread its contributions to the output samples.

The total contribution at a given <x,y> location is the sum of the contribution along a ray through the kernel that is perpendicular to the screen with its origin at <x,y>. The sum is calculated as the integral along z of the ray. Projecting the sample onto the image plane at pixel <x,y> is:

$$contribution_D(x,y) = \int_{-\infty}^{\infty} h_V(x-D_x, y-D_y, w)\, \rho(D)\, dw$$

For a given sample, ρ is a constant and since ρ is independent of w, ρ can be moved outside the integral:

$$contribution_D(x,y) = \rho(D) \int_{-\infty}^{\infty} h_V(x-D_x, y-D_y, w)\, dw$$

Notice that the integral is independent of the sample's density. Since it only depends on the sample's <x, y> projected location, the function footprint is defined:

$$footprint(x,y) = \int_{-\infty}^{\infty} h_V(x,y,w)\, dw$$

where <x,y> denotes the displacement of an image sample from the center of the shaded sample's image plane projection.

METHOD

For orthographic views, the footprint of each sample is the same except for a screen space offset. Therefore, the renderer needs only to calculate the footprint function once for each view of the data set. Once the footprint is known, the renderer can sample the footprint function at each pixel that lies within the footprint's extent and contribute the appropriate amount to the pixel. The weight at each pixel is:

$$weight(x,y)_D = footprint(x-D_x, y-D_y)$$

where <D_x, D_y> denotes the sample's image plane projection and <x,y> denote the pixel's image plane location.

Sampling the footprint function involves an integration. Many kernels are difficult to integrate analytically and the renderer must use discrete methods. Since the renderer does not want to integrate this function many times for each sample, it builds a table on a fine grid and then performs table look-ups to evaluate the function. The renderer needs to determine two things to build the footprint table for a particular view. First, it calculates the screen space extent of the projection of the kernel, which in an orthographics view is constant for each input sample. All pixels that lie within the extent may be affected by the given sample. Second, the renderer calculates a mapping from the view-transformed extent to an extent that surrounds the projection of a generic kernel. The generic kernel table is calculated by a pre-processing program that runs once for a given kernel.

Since the pre-processor runs once, it does not matter how long it takes to compute the integration of the kernel. By using a pre-computed generic table, the renderer can easily change reconstruction kernels by reloading the generic table from disk.

Once the renderer builds the view-transformed table, the table is used by the renderer for each sample, by centering the table at each sample's projected screen position and calculating the screen space extent of the kernel by offsetting the extent of the projected kernel. For each pixel in the extent, the renderer samples the table to determine the amount of contribution for the pixel. The renderer builds the view-transformed footprint table on a grid that has many samples per pixel. Without oversampling rendering artifacts will occur.

Building the Generic Footprint Table

The method assumes that the extent of the reconstruction kernel is a sphere. If the extent is not a sphere, the pre-processor bounds the kernel by a sphere. For efficiency reasons, the bounding sphere should be as tight as possible. A loose fitting sphere will cause the pre-processor to build a generic table that has many zero entries, which causes the renderer to visit many pixels that are not affected by a given sample. For a spherical kernel, the radius of the sphere is equal to the width of the reconstruction kernel. This sphere, called the unit region sphere, defines the region a sample can affect. Within this region, on a discrete grid, the pre-processor integrates the kernel along the z direction and stores the result in a table. This table is called the generic footprint table. During image generation, the renderer determines the extent of the projection of the view-transformed

region sphere. In addition, the renderer determines a mapping of each point in that extent onto the extent surrounding the unit region sphere in order to build the view-transformed footprint table. The projection of the unit region sphere on the image plane is a circle. The mapping from view-transformed extent to generic extent is then a mapping from the projection of the view-transformed region to a circle.

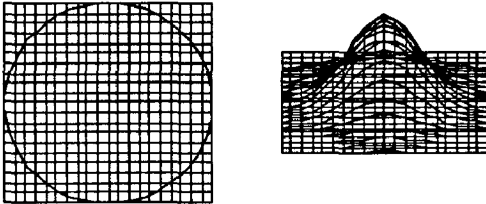


Figure 2. Generic Footprint Function Table

EXTENTS AND MAPPINGS

There are two basic cases for determining extents and mappings: the unit sphere maps to a sphere after applying the viewing transform, or the unit sphere maps to an ellipsoid. The result is a sphere when the input volume has equal spacings in each of the grid directions and the viewing transform has only uniform scaling. The result is an ellipsoid when the input volume has non-uniform spacing in each of the grid directions or the viewing transform has non-uniform scaling. Since a sphere is a special case of an ellipsoid, the renderer currently uses the elliptical method described below for all volumes.

Extent and Mapping for Spherical Kernels

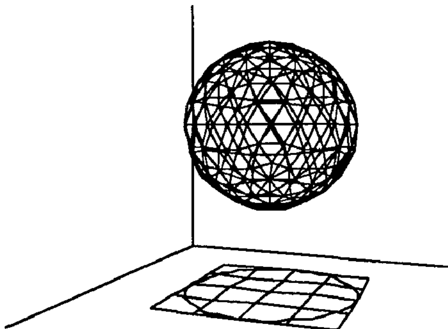


Figure 3. Spherical Kernel

Even when the kernel maps to a sphere, the renderer can not use the generic table directly and must build a view-transformed table. If the grid scale value and the view scale value are both 1.0, the generic table is used, otherwise the renderer builds a view-transformed. This makes a table access fall exactly at table entries and causes all the interpolations to only occur once.

Extent

Many input volumes have fewer samples per face than the desired number of pixels in the image. This means that the input sampling rate is much smaller than the output sampling rate and each input sample needs to cover many pixels. The renderer calculates the extent of a sample's effect by scaling the unit extent by the grid scale value and the view scale value.

The extent in both the x and y directions is:

$$extent = 2.0 * kernel_width * grid_scale * view_scale$$

Mapping

The mapping from scaled extent to unit extent is trivial in the case of a spherical result. The projection of the sphere onto the image plane is a circle. The mapping from one circle to another circle is a scaling by the ratio of the radii of the two circles. The mapping is:

$$mapping = \frac{1.0}{grid_scale_factor * view_scale_factor}$$

The renderer uses the mapping to map cells of the view-transformed footprint table to the generic footprint table. If the view is simply rotated and the scale factors do not change, the view-transformed footprint table can be used again.

Extent and Mapping for Elliptical Kernels

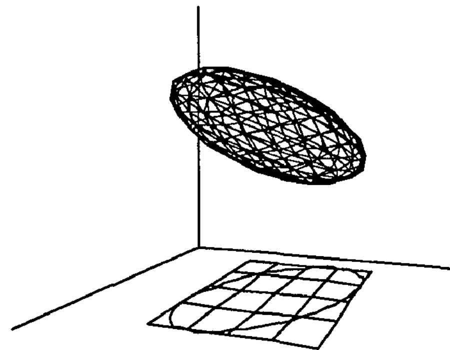


Figure 4. Elliptical Kernel

If the scalings in grid directions are different, the region sphere transforms into a region ellipsoid. The projection of the region ellipsoid is always a screen space ellipse. The extent of a kernel's effect is the extent of the projected ellipse, and the mapping from view-transformed table to generic table is a mapping from the projected

ellipse to the unit circle.

Extent

The region ellipsoid is found by transforming the unit region sphere by the grid scale transform and then by the viewing transform. By treating the unit region sphere as a quadric surface, the transformations become matrix multiplications.

Let the original unit sphere be U:

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

and let the grid scale transform be S:

$$S = \begin{bmatrix} S_i & 0 & 0 & 0 \\ 0 & S_j & 0 & 0 \\ 0 & 0 & S_k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and let the viewing transform be V:

$$V = \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The grid space region ellipsoid E is:

$$E = S * U$$

To transform the quadric surface the renderer calculates both the inverse viewing transform and its transpose. The resulting screen space ellipsoid R is:

$$R = V^{-1^T} * E * V^{-1}$$

with

$$R = \begin{bmatrix} A & D/2 & E/2 & 0 \\ D/2 & B & F/2 & 0 \\ E/2 & F/2 & C & 0 \\ 0 & 0 & 0 & -K \end{bmatrix}$$

This gives an ellipsoid defined by:

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz = K \quad (1)$$

By rearranging terms, completing the square, and solving for x and y, the renderer can calculate the screen space extent of the transformed ellipse. The x extent is:

$$x = \pm \sqrt{\frac{K}{A - \frac{D^2}{4B} - \frac{(E - \frac{DF}{2B})^2}{4(C - \frac{F^2}{4B})}}}$$

and the y extent is:

$$y = \pm \sqrt{\frac{K}{B - \frac{D^2}{4A} - \frac{(F - \frac{DE}{2A})^2}{4(C - \frac{E^2}{4A})}}}$$

Mapping

The renderer also needs to calculate the mapping from the projection of the region ellipsoid back to the unit circle. To do this, the renderer first calculates the screen space projection of the region ellipsoid which is an ellipse. To find the ellipse, first rewrite (1) as a quadratic in z. The quadratic is:

$$Cz^2 + (Ex + Fy)z + (Ax^2 + By^2 + Dxy - K) = 0$$

Points on the edge of the projection of R have only one root in this quadratic. There is only one root to the quadratic $aZ^2 + bZ + c = 0$ when $b^2 - 4ac = 0$ or in this case when:

$$(Ex + Fy)^2 - 4C(Ax^2 + By^2 + Dxy - K) = 0$$

Grouping the x^2, the y^2, and the xy terms gives the screen space projection ellipse P:

$$Xx^2 + Yy^2 + Zxy = K \quad (2)$$

where:

$$X = (A - \frac{E^2}{4C}) \quad Y = (B - \frac{F^2}{4C}) \quad Z = (D - \frac{EF}{2C})$$

Once the renderer calculates the screen space ellipse, it can define a transformation that takes points from the screen space ellipse into the unit circle. This is the inverse of the mapping that takes the unit circle into the screen space ellipse. To calculate the second mapping, the renderer needs to calculate two things: the amount to scale along the x axis and the y axis, and the amount of rotation about the view direction.

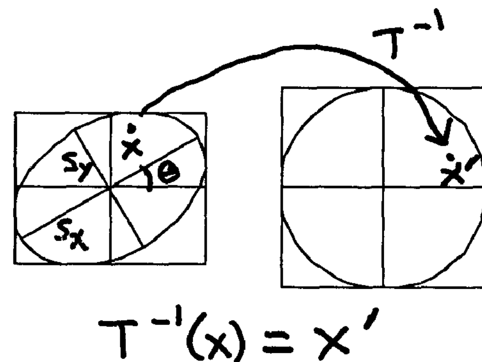


Figure 5. Ellipse to Circle Mapping

The renderer finds these values by solving for T such that:

$$P = T * U * T^T$$

writing P as:

$$P = \begin{bmatrix} X & Z & 0 & 0 \\ Z & Y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

and T as:

$$T = \begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and U as:

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

and solving for X, Y, and Z:

$$a^2 + b^2 = X \quad (3)$$

$$c^2 + d^2 = Y \quad (4)$$

$$ac + bd = Z \quad (5)$$

This looks like a problem since there are three equations and four unknowns, a, b, c, and d, but T is a matrix that is made up of a scale in the x and y directions followed by a rotation about the z axis. So:

$$a = (S_x)*\cos\theta \quad b = -(S_x)*\sin\theta \quad (6)$$

$$c = (S_y)*\sin\theta \quad d = (S_y)*\cos\theta \quad (7)$$

Plugging (6) and (7) into (3), (4), and (5) and applying some algebraic manipulation brings:

$$\frac{(X-Y)}{Z} = \frac{\cos\theta}{\sin\theta} - \frac{\sin\theta}{\cos\theta}$$

This seems to be a problem when Z is zero but upon investigation of P, when Z is zero, P is a scaling of U, therefore T is simply:

$$a = \sqrt{X} \text{ and } d = \sqrt{Y} \text{ and}$$

$$b = c = 0$$

When Z is non-zero, let:

$$G = \frac{(X-Y)}{Z} \text{ and } w = \frac{\cos\theta}{\sin\theta}$$

then:

$$G = w - \frac{1}{w} \text{ or } w^2 - Gw - 1 = 0$$

using the quadratic formula w is:

$$w = \frac{G \pm \sqrt{G^2+4}}{2}$$

Given w, θ is $arctan(\frac{1.0}{w})$. This gives both $\sin\theta$ and $\cos\theta$ and allows the renderer to solve for S_x and S_y using (6) and (7). S_x and S_y are undefined in the above equations when $\theta = 45$ degrees. When this occurs, the renderer cheats and rotates the view an additional 0.01 degrees about the view direction. This allows the renderer to calculate S_x and S_y with little if any effect on the image.

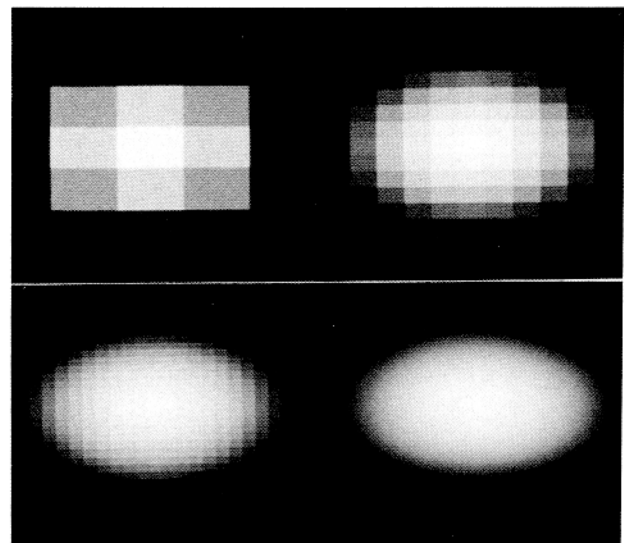
With θ, S_x, and S_y the renderer builds T by multiplying the identity matrix by a scale matrix of S_x and S_y, followed by a z rotation matrix of θ. The mapping from P into U is then the inverse of T: T^{-1}.

The renderer uses T^{-1} to map cells of the view-transformed footprint table to the generic footprint table.

TABLE SIZES AND KERNELS

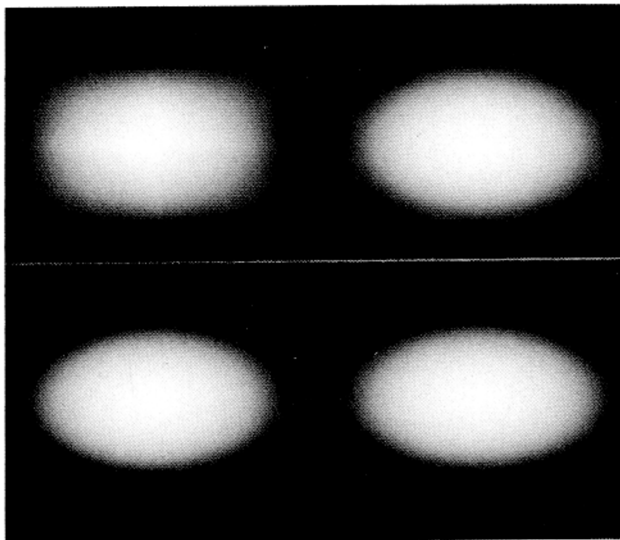
There are three parameters that can change in building footprint tables: the size of the tables, how the tables are accessed, and the table's underlying kernel.

There is a space versus quality tradeoff between the size of the footprint tables and the resultant artifacts in the images. Image 1 shows this tradeoff on an elliptical projection. Each picture in image 1 is of a single sample point scaled 120 by 60 by 60. The upper left picture in the image uses a view-transformed footprint table with 5 by 5 entries. The upper right uses a table that is 11 by 11. The lower left uses a table that is 21 by 21. The lower right uses a table that is 101 by 101. Notice how smoothness increases with table size.



There is a time versus space tradeoff in how the table is sampled. If the footprint table has a lot of entries, then nearest neighbor sampling works fine. If, on the other

hand, the table is coarse, then the renderer needs to interpolate samples from the nearest neighbors. Image 2 shows this tradeoff on an elliptical projection. Each picture in image 1 is of a single sample point scaled 120 by 60 by 60. The upper left picture in the image uses a view-transformed footprint table with 5 by 5 entries. The upper right uses a table that is 11 by 11. The lower left uses a table that is 21 by 21. The lower right uses a table that is 101 by 101. In each case, the renderer generates the table value with a bilinear function. Compared to Image 1, the footprint is much smoother on a lot smaller table. However, a reasonable table size is required to avoid bilinear artifacts.



The third thing to change is the kernel itself. The choice of kernel can drastically affect the quality of an image. Image 3 is a single sample as above with four different

kernels. *radius* is the normalized distance from the center of the kernel. The upper left has a cone function modeling the result of the z integration. The upper right has a Gaussian function as the model. The lower left has the first five lobs of a sync function as the model. The lower right has the bilinear function as the model.

Image 4 is a portion of a computed tomography study of a human head. The data is clipped to only show the left eye. The spread of the Gaussian kernel changes in each sub-image. In the upper left the Gaussian is scaled so that its tail stops 25 percent of the way to the next voxel (where 100 percent just touches the next voxel). This scale changes from 25 to 225 percent in steps of 25 percent from left to right and top to bottom. In the first images the kernels are too sharp and do not overlap leaving gaps. In the last images the kernels are very broad and over blur the images. All the images in the following section were generated with a Gaussian kernel with a sigma of 2.5 and a spread of 160 percent.

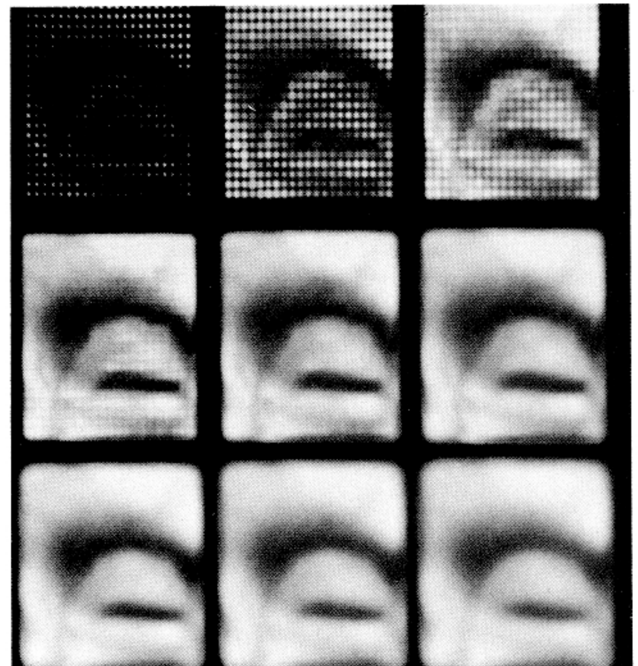
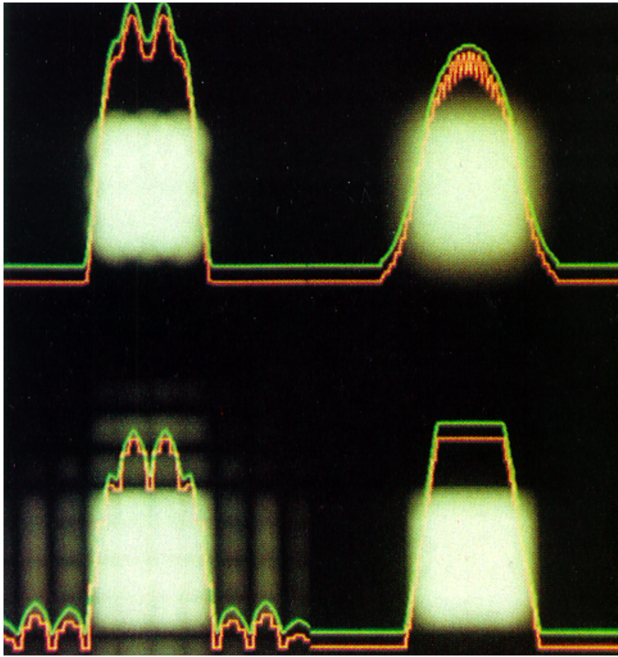


Image 5 shows each of the above kernels operating on a 3 by 3 by 1 grid of constant values. These kernels are approximations to the true z integration of a three-dimensional kernel. The view-transformed table as 10 by 10 entries. The patterns in the upper left image are the result of multiple kernels not summing to one at all points. The patterns in the lower left image are the result of ringing from the sync function at the edges of the sample space. Notice the sharp second order discontinuities at the corners of the image from the bilinear function at the lower right. Superimposed on the images are line drawings of a single scanline's grey value. The green line is when table values were interpolated from nearest neighbors. The red line is when just the single

nearest neighbor was used.



SAMPLE IMAGES

Image 6 is a single sample with an elliptical projection. The four views are of the sample with the volume rotated 0 degrees, 10 degrees, 30 degrees, and 45 degrees about the view direction. The ellipse does not change shape or size as the volume grid is rotated about the z axis.

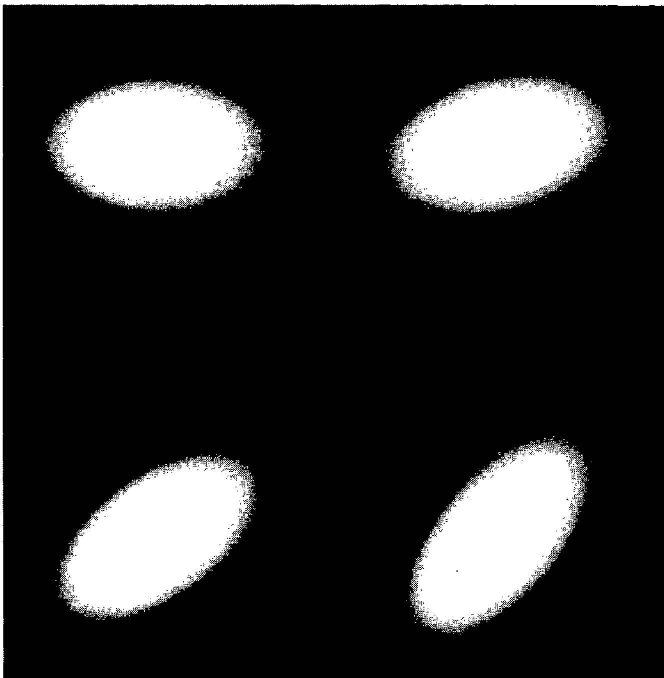


Image 7 is an image of ozone concentrations over the northeast corner of the United States in July 15, 1980. The input grid is 64 by 52 by 32 with very uneven spacing in the z direction compared to the x and y directions. The shading model is the emittance model with color and opacity based on concentration values. Since the algorithm works back to front, any image (in this case a texture map of state boundaries) can be used as a starting working image. The clouds are colored with blue being low concentration, going to green for intermediate concentration, and finally red where concentration exceeds the government's legal limits.

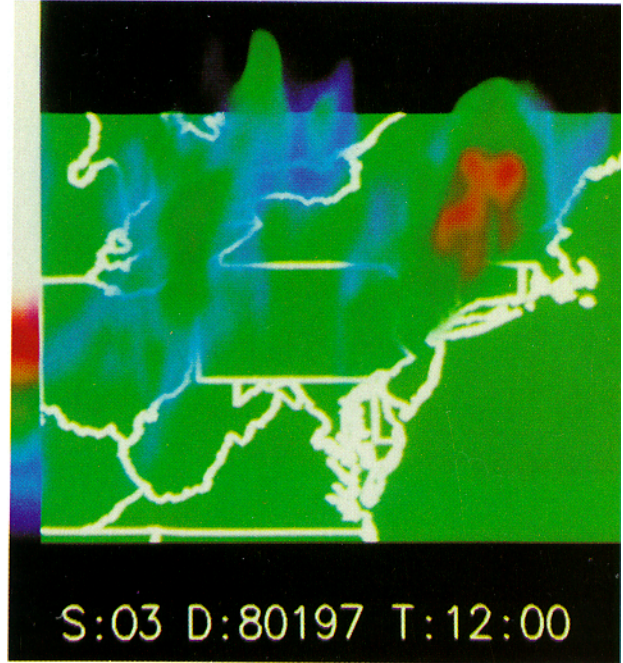


Image 8 is an image generated from the electron density of the p-orbitals of copper chloride. The input grid is 64 by 64 by 64 with even spacing in each grid direction. The viewing transform has only uniform scaling. The shading model is the emittance model with color and opacity based on density value. The underlying data has no surfaces and the image has a cloudy nature.

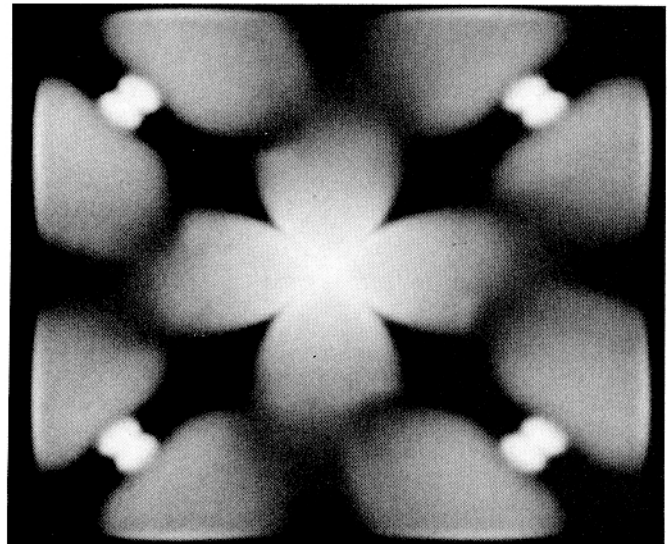


Image 9 is an isodensity surface from an electron density map of Staphylococcus Aureus ribonuclease. The initial

data set was 24 by 20 by 11 and resampled using a tri-cubic function during a preprocessing step. The resultant data set is 137 by 113 by 59 with even spacing in each grid direction. The data set was then shaded using Levoy [9] shading techniques into a shaded data set. This data set was rendered using the feed-forward renderer with a shading table that mapped grey scale shade and opacity one-to-one with the input data.

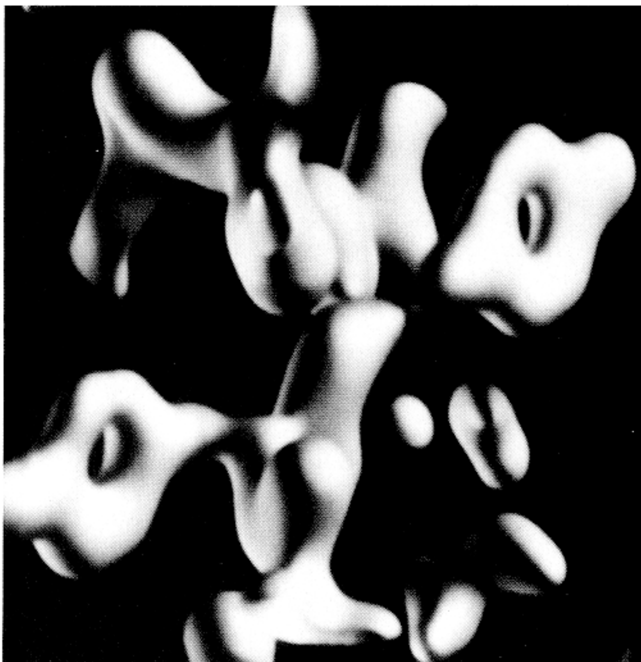
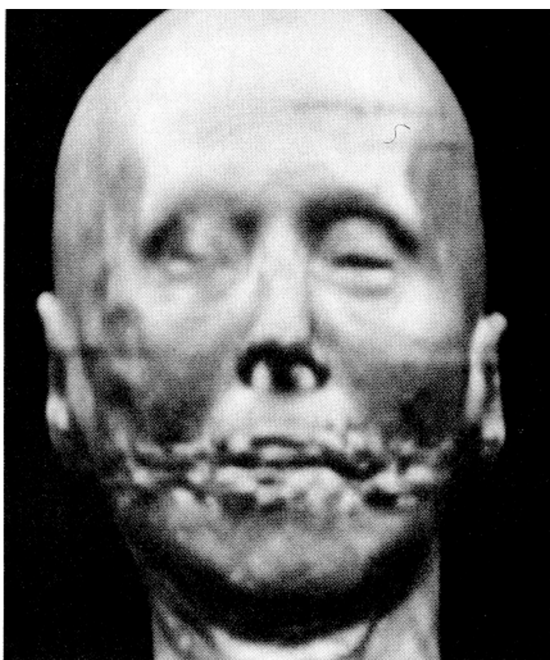
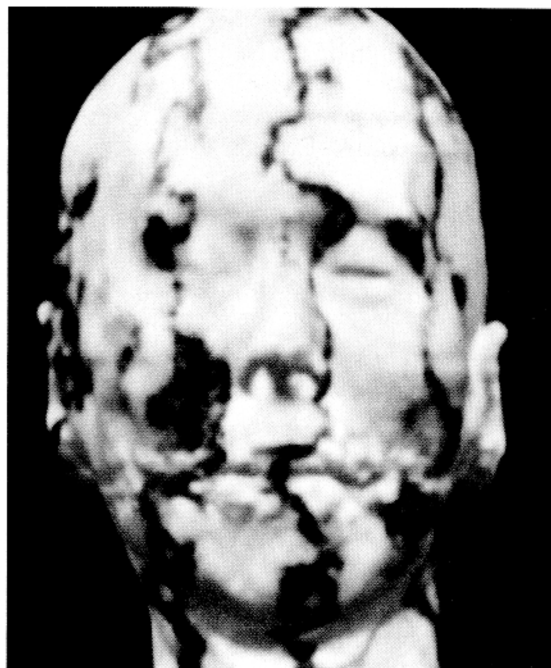


Image 10 and Image 11 are images of a computed tomography study of a human head. The input grid is 96 by 128 by 113 with even spacing in each grid direction. The shader uses local gradients as normals for a Phong shading model. In image 10 the color is grey.



In image 11, the shader uses the <*i,j,k*> grid values as three-dimensional texture indices. Here the data has surfaces, and the opacity tables are chosen to bring out the skin/air interface.



ENHANCEMENTS

Perspective

The above method can be enhanced to handle perspective with two modifications. First, the image plane footprint extent and mapping must be generated for each sample. These are then used to map each sample's screen space footprint into the generic footprint table. Second, the renderer approximates the screen space perspective projection of an ellipsoid by an ellipse. While this is not true in the general case, the ellipsoids that define regions of effect are typically small, on the order of a few pixels, and they seldom vary far from elliptical. With these two changes, the renderer can deal with the changing sampling rates of the input grid with respect to the screen brought on by the perspective transform.

Adaptive Re-Sampling

The above method can be further enhanced to rescale the grid spacing in areas of high variations. If nearby input samples vary widely, or produce colors that vary widely, the renderer can interpolate new values between the input samples along grid directions and adjust the appropriate grid spacing on the fly. The renderer builds a new footprint table each time the sampling rate changes. The kernel's projection quits being symmetric about the grid axis, but it only differs by a scale along the the axis. For example, the scale factor may be 4.0 for positive i and be 2.0 for negative i for samples that

border regions where adaptive sampling occurs. Samples within the region of resampling will each have identical kernel projections. The adaptive resampling will help the renderer alleviate the problem of the shading model introducing high frequency components. Additionally, the amount of subdivision the renderer performs is a time verses quality tradeoff control. Quick views undersample the volume, and produce coarse images. Images that require better quality just take longer to generate.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Turner Whitted, for our numerous discussions and his helpful insight, and Robert Whitton, for his invaluable help in converting some vague feelings and intuitions into concrete mathematical formulas. In addition, I would like to thank Greg Abram and Greg Gilley for helping me with incorporating our company's shading library into my volume renderer. I would like to thank Dana Smith and Greg Gilley for their help in proof reading this paper. They are responsible for much of the readability of the paper and none of its faults.

The copper chloride p-orbitals data is courtesy of Michael Pique, Scripps Clinic Molecular Biology, La Jolla, California. The computed tomography data is courtesy of Radiation Oncology of the University of North Carolina at Chapel Hill, North Carolina. The ozone concentration data is courtesy of the Regional Oxidant Model group of the Environmental Protection Agency, Research Triangle Park, North Carolina. The original Staphylococcus Aureus ribonuclease data is courtesy of Chris Hill of the University of York. The shaded Staphylococcus Aureus ribonuclease data is courtesy of Marc Levoy University of North Carolina at Chapel Hill, North Carolina.

REFERENCES

1. Abram, Greg, Turner Whitted. Building Block Shaders. Proceedings of SIGGRAPH'90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics 24*, 4(August 1990).
2. Blinn, Jim. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. Proceedings of SIGGRAPH'82 (Boston, Massachusetts, July 26-30, 1982). In *Computer Graphics 16*, 3(July 1982), 21-30.
3. Drebin, Robert, Lorne Carpenter, Pat Hanrahan. Volume Rendering. Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics 22*, 4(August 1988), 65-74.
4. Frieder, Gideon, Dan Gordon, Anthony Reynolds. Back-to-Front Display of Voxel-Based Objects. *IEEE Computer Graphics and Applications 5*, 1(January 1985).
5. Gallagher, Richard and Joop Nagtegaal. An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volumes. Proceedings of SIGGRAPH'89 (Boston, Massachusetts, July 31 - August 4, 1989). In *Computer Graphics 23*, 3(July 1989), 185-194.
6. Greene, Ned, Paul Heckbert. Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter. *IEEE Computer Graphics and Applications 6*, 6(June 1986).
7. Kajiya, Jim, Brian Von Herzen. Ray Tracing Volume Densities. Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27, 1984). In *Computer Graphics 18*, 3(July 1984), 165-174.
8. Lenz, Reiner, Bjorn Gudnumdsson, Bjorn Lindskog, Per Danielsson. "Display of Density Volumes", *IEEE Computer Graphics and Applications 6*, 7(July 1986).
9. Levoy, Mark. "Volume Rendering: Display of Surfaces from Volume Data", *IEEE Computer Graphics and Applications 8*, 5(May 1988).
10. Lorensen, William, Harvey Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics 21*, 4(July 1987), 163-170.
11. Porter, Thomas, Tom Duff. Compositing Digital Images. Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27, 1984). In *Computer Graphics 18*, 3(July 1984), 253-260.
12. Sabella, Paolo. A Rendering Algorithm for Visualizing 3D Scalar Data. Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics 22*, 4(August 1988), 51-58.
13. Upson, Craig, Michael Keller. VBUFFER: Visible Volume Rendering. Proceedings of SIGGRAPH'88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics 22*, 4(August 1988), 59-64.
14. VanHook, Tim. Personal Communication. September 1986.
15. Westover, Lee. "Interactive Volume Rendering" Proceedings of the Chapel Hill Workshop on Volume Visualization, May 1989.