High-Quality Volume Rendering Using Texture Mapping Hardware

Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter and Arie Kaufman[†]

Center for Visual Computing (CVC)[‡]
and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA

Abstract

We present a method for volume rendering of regular grids which takes advantage of 3D texture mapping hardware currently available on graphics workstations. Our method produces accurate shading for arbitrary and dynamically changing directional lights, viewing parameters, and transfer functions. This is achieved by hardware interpolating the data values and gradients before software classification and shading. The method works equally well for parallel and perspective projections. We present two approaches for our method: one which takes advantage of software ray casting optimizations and another which takes advantage of hardware blending acceleration.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: volume rendering, shading, ray casting, texture mapping, solid texture, hardware acceleration, parallel rendering

1 Introduction

Volumetric data is pervasive in many areas such as medical diagnosis, geophysical analysis, and computational fluid dynamics. Visualization by interactive, high-quality volume rendering enhances the usefulness of this data. To date, many volume rendering methods have been proposed on general and special purpose hardware, but most fail to achieve reasonable cost-performance ratios. We propose a high-quality volume rendering method suitable for implementation on machines with 3D texture mapping hardware.

Akeley [1] first mentioned the possibility of accelerating volume rendering using of 3D texture mapping hardware, specifically on the SGI Reality Engine. The method is to store the volume as a solid texture on the graphics hardware, then to sample the texture using planes parallel to the image plane and composite them into the frame buffer using the blending hardware. This approach considers only ambient light and quickly produces unshaded images. The images could be improved by volumetric shading, which implements a full lighting equation for each volume sample.

Cabral et al. [3] rendered 512×512×64 volumes into a 512×512 window (presumably with 64 sampling planes) in 0.1 seconds on a four Raster Manager SGI RealityEngine Onyx with one 150MHz CPU. Cullip and Neumann [4] also produced 512×512 images on the SGI RealityEngine (again presumably 64 sampling planes since the volume is 128×128×64) in 0.1 seconds. All of these approaches keep time-critical computations inside the graphics pipeline at the expense of volumetric shading and image quality.

Van Gelder and Kim [6] proposed a method by which volumetric shading could be incorporated at the expense of interactivity. Their shaded renderings of 256×256×113 volumes into 600^2 images with 1000 samples along each ray took 13.4 seconds. Their method is slower than Cullip and Neumann's and Cabral et al.'s because they must re-shade the volume and reload the texture map for every frame because the colors in the texture memory are view dependant.

Cullip and Neumann also described a method utilizing the PixelFlow machine which pre-computes the x, y and z gradient components and uses the texture mapping to interpolate the density data and the three gradient components. (The latter is implemented partially in hardware and partially in software on the 128^2 SIMD pixel processors [5].) All four of these values are used to compute Phong shaded samples which are composited in the frame buffer. They predicted that 256^3 volume could be rendered at over 10Hz into a 640x512 image with 400 sample planes. Although this is the first proposed solution to implement full Phong lighting functionality, it has never been realized (as far as we know) because it would require 43 processor cards, a number which can not easily fit into a standard workstation chassis [4].

Sommer et al. [13] described a method to render 128^3 volumes at 400^2 resolution with 128 samples per ray in 2.71 seconds. They employ a full lighting equation by computing a smooth gradient from a second copy of the volume stored in main memory. Therefore, they do not have to reload the texture maps when viewing parameters change. However, this rendering rate is for isosurface extraction; if translucent projections are required, it takes 33.2 seconds for the same rendering. They were the first to propose to resample the texture volume in planes parallel to a row of image pixels so that a whole ray was in main memory at one time. They mention the potential to also interpolate gradients with the hardware.

All of these texture map based methods either non-

[†]{dachille|kkreeger|baoquan|ingmar|ari}@cs.sunysb.edu
[‡]http://www.cvc.sunysb.edu

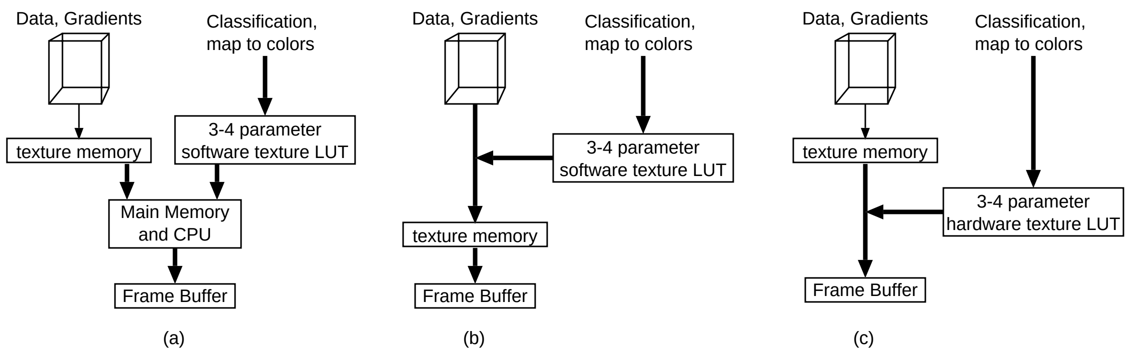

Data, Gradients
Classification, map to colors
texture memory
3-4 parameter software texture LUT
Main Memory and CPU
Frame Buffer
(a)
Data, Gradients
Classification, map to colors
3-4 parameter software texture LUT
texture memory
Frame Buffer
(b)
Data, Gradients
Classification, map to colors
texture memory
3-4 parameter hardware texture LUT
Frame Buffer
(c)

Figure 1: *Three architectures for texture map based volume rendering: (a) Our architecture, (b) Traditional architecture of Van Gelder and Kim, and (c) Ideal architecture of Van Gelder and Kim. The thick lines are the operations which must be performed for every frame.*

interactively recompute direction-dependent shading each time any of the viewing parameters change, compute only direction-independent shading, or compute no shading at all. Our method shades every visible sample with view-dependent lighting at interactive rates.

We do not adapt the ray casting algorithm to fit within the existing graphics pipeline, which would compromise the image quality. Instead, we only utilize the hardware where it provides run time advantages, but maintain the integrity of the ray casting algorithm. For the portions of the volume rendering pipeline which can not be performed in graphics hardware (specifically shading) we use the CPU.

In volume rendering by ray casting, data values and gradients are estimated at evenly spaced intervals along rays emanating from pixels of the final image plane. Resampling these data values and gradients is often the most time consuming task in software implementations. The texture mapping hardware on high-end graphics workstations is designed to perform resampling of solid textures with very high throughput. We leverage this capability to implement high throughput density and gradient resampling.

Shading is the missing key in conventional texture map based volume rendering. This is one of the reasons that pure graphics hardware methods suffer from lower image quality than software implementations of ray-casting. For high-quality images, our method implements full Phong shading using the estimated surface normal (gradient) of the density. We pre-compute the estimated gradient of the density and store it in texture memory. We also pre-compute a lookup table (LUT) to store the effect of an arbitrary number of light sources using full Phong shading.

The final step in volume rendering is the compositing, or blending, of the color samples along each ray into a final image color. Most graphics systems have a frame buffer with an opacity channel and efficient blending hardware which can be used for back-to-front compositing. In the next section we present our architecture, in Sec. 3 we present our rendering optimization techniques, in Sec. 4 we compare our method to existing methods, in Sec. 5 we present our parallel implementation, and finally in Sec. 6 we give our results and draw conclusions.

2 Architectural Overview

Fig. 1(a) shows our architecture in which density and gradients are loaded into the texture memory once and resampled by the texture hardware along rays cast through the volume. The sample data for each ray (or slice) is then transferred to a buffer in main memory and shaded by the CPU. The shaded samples along a ray are composited and the final pixels are moved to the frame buffer for display. Alternatively within the same architecture, the shaded voxels can be composited by the frame buffer.

Fig. 1(b) shows the architecture that is traditionally used in texture map based shaded volume rendering. One of the disadvantages of this architecture is that the volume must be re-shaded and re-loaded every time any of the viewing parameters changes. Another problem with this method is that RGBα values are interpolated by the texture hardware. Therefore, when non-linear mappings from density to RGBα are used, the interpolated samples are incorrect. We present a more detailed comparison of the various methods in Sec. 4.

In Fig. 1(c), Van Gelder and Kim's [6] Ideal architecture is presented. In this architecture, the raw density and volume gradients are loaded into the texture memory one time only. The density and gradients are then interpolated by the texture hardware and passed to a post-texturing LUT. The density values and gradients are used as an index into the LUT to get the RGBα values for each sample. The LUT is based on the current view direction and can be created using any lighting model desired (e.g., Phong) for any level of desired image quality. This method solves the problems of the current architecture including pre-shading the volume and interpolating RBGα values. However, a post-texturing LUT would need to be indexed by the local gradient which would require an infeasibly large LUT (see Sec. 2.2).

2.1 Sampling

Ray casting is an image-order algorithm, which has the drawback of multiple access of voxel data, since sampling within the dataset usually requires the access of eight or more neighboring data points [2, 11]. Ray casting using texture mapping hardware the multiple voxel accesses by using the hardware to perform the resampling.

Graphics pipelines work on primitives, or geometric shapes defined by vertices. Traditionally, volume rendering has been achieved on texturing hardware systems by orienting polygons parallel to the image plane and then compositing these planes into the frame buffer as in Fig. 2.

Because of the way that the texture hardware interpolates values, the size of the original volume does not adversely affect the rendering speed of texture map based volume ren-

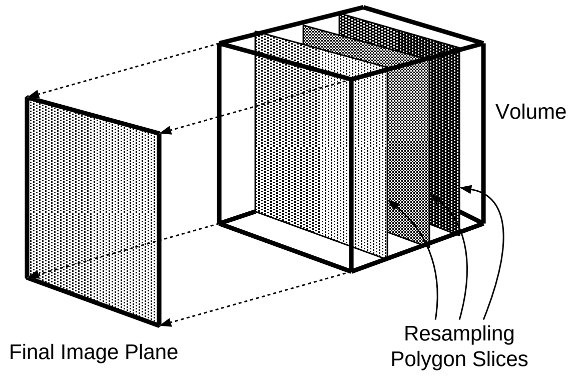


Figure 2: *Polygon primitives for texture based volume rendering when the final image is oriented parallel to one of the faces of the volume*

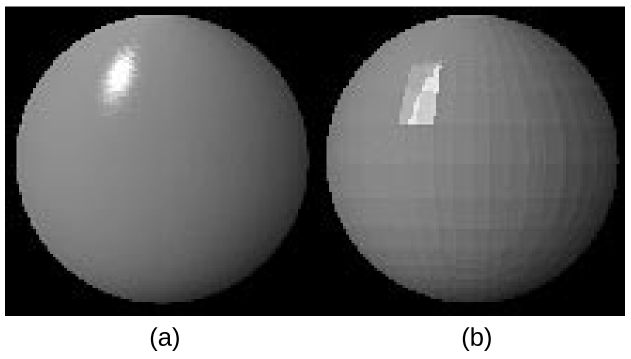


Figure 3: *Sphere rendered using (a) 8-bit fixed-point Phong shading calculations, and (b) with a 5-bit, 4-index LUT*

derers. Instead, the image size and number of samples along each ray dictate how many texture map resamplings are computed. This is true as long as the volume data fits in the texture memory of the graphics system. A typical high-end graphics system is equipped with 64 MBytes of texture memory which holds volumes up to 256^3 with 32-bits per voxel. Newer hardware supports fast paging between main and texture memory for higher virtual texture memory than is physically available [12, 8].

2.2 Shading Options

The 3-4 parameter LUT presented by all three architectures in Fig. 1 is used to optimize the computation of the lighting equation for shading of samples. The LUT summarizes the contribution of ambient, diffuse, and specular shading for every gradient direction in the LUT.

We present alternatives to compute the shading of the re-sampled points along the rays. Van Gelder and Kim implied that a 3-4 parameter LUT within the graphics pipeline could be used. Even if there were only four viewing parameters to consider, four 8-bit indices into a LUT would mean 256^4 = 4 Gigaentries in the table. Since this is an RGBα table it would consume 16 GBytes of memory. Furthermore, it would require 4 Gigacalculations to compute the LUT. If the same calculations were used on the resampled data, then a $400 \times 400 \times 256$ projection of a volume could be shaded with 40 Megacalculations, or two orders of magnitude less than computing the LUT. If the table is to be indexed by only four parameters (G_x, G_y, G_z, density value) then the table would need to be recomputed every time any light or viewing parameter changed, or every frame in the usual case. Trade-offs could occur to also use eye and light position as indices, but the table is already much too large. Reducing the precision brings the table downn to a much more manageable size. However, that deteriorates the image quality. Fig. 3(a) shows a sphere generated with an 8-bit fixed-point Phong calculation and Fig. 3(b) with a 4-index Phong LUT with 5-bits per index and 8-bit values. Five bits is about the largest that can be considered for a manageable lookup table since $32^4 \times$4Bytes = 4 MBytes.

Fortunately, with the Phong lighting model it is possible to reduce the size of the LUT by first normalizing the gradient and using a Reflectance Map [14]. With this method, the Phong shading contribution for $6n^2$ surface normals is computed. They are organized as six n^2 tables that map to the six sides of a cube with each side divided into n^2 equal patches. Each sample gradient vector $G_{x,y,z}$ is normalized by its maximum component to form $G_{u,v,index}$, where index ennumerates the six major directions. A direct lookup returns RGBα intensities which are modulated with the object color to form the shaded sample intensity.

Trade-offs in image quality and frame rate occur with the choice of shading implementation. We have chosen to implement reflectance map shading because it delivers good image quality with fast LUT creation and simple lookup.

2.3 Pre-computation of Volume Gradients

To be able to compute accurately shaded volumes we pre-compute the G_x, G_y and G_z central difference gradient values at each voxel position. Our voxel data type is then four 8-bit values which we load into an RGBα type texture map, although the fields are really three gradient values and raw density. These gradient values are then interpolated along with the raw density values to the sample positions by the 3D texture mapping hardware. Assuming a piecewise linear gradient function, this method produces the same gradient values at the sample locations as if gradients themselves were computed at unit voxel distances from the sample point. The gradient computation needs to occur only once for any volume of data being rendered, regardless of changes in the viewing parameters. Since the gradients are processed off-line, we have chosen to compute high-quality Sobel gradients at the expense of speed. Computing gradients serially off-line for a 256^3 volume takes 12 seconds on a 200MHz CPU.

3 Rendering Optimization Techniques

Software ray casting algorithms enjoy speedup advantages from several different optimization techniques; we consider two of them. The first is space leaping, or skipping over areas of insignificant opacity. In this technique, the opacity of a given sample (or area of samples) is checked before any shading computations are performed. If the opacity is under some threshold, the shading and compositing calculations are skipped because the samples minimally contribute to the final pixel color for that ray.

A second optimization technique employed by software ray casting is the so-called early ray termination. In this technique, only possible in front-to-back traversal, the sampling, shading and compositing operations are terminated once the ray reaches full opacity. In other words, the ray has reached the point where everything behind it is obscured by

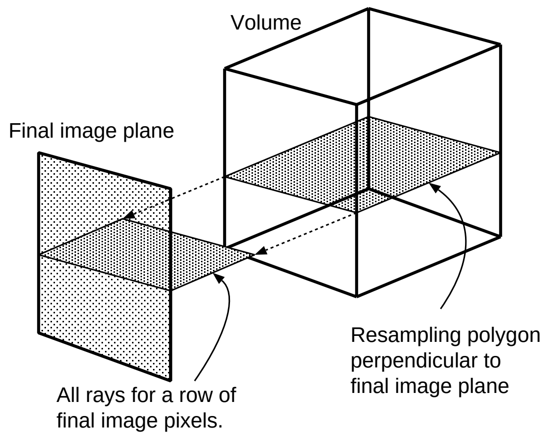

Volume
Final image plane
Resampling polygon perpendicular to final image plane
All rays for a row of final image pixels.

Figure 4: *Polygon primitives for texture based volume rendering with rows of rays using the Planes method*

other objects closer to the viewer. Since we are using the hardware to perform all interpolations, we can only eliminate shading and compositing operations. However, these operations typically dominate the rendering time.

Below, we propose two methods to apply these optimization techniques to speed up the computation of accurately shaded volume rendering utilizing texture mapping hardware.

3.1 Planes: Compositing on the CPU

Previous texture map based volume rendering methods resample a volume by dispatching polygons down the graphics pipeline parallel to the image plane. The textured polygons are then blended into the frame buffer without leaving the graphics hardware [1, 7, 3, 4, 6]. Fig. 2 shows the common polygon resampling direction.

In contrast, since we propose to shade the samples in the CPU and take advantage of the two optimization techniques discussed earlier, we wish to have all the samples for a ray in the main memory at one time. For this reason we have chosen an alternative order for accessing the resampling functionality of the 3D texture map hardware. Polygons are forwarded to the graphics pipeline oriented in such a way that they are coplaner with the rays that would end up being a row of pixels in the final image plane. Fig. 4 shows the polygon orientation for this method.

Once the data has been loaded back into the main memory, the raw density value, and three gradient components are extracted and used in a reflectance map computation to generate the Phong shaded RGBα for each sample. The samples are composited front-to-back taking advantage of early ray termination and skipping over low opacity samples. Similar to the shear-warp approach [10], the composition is now an orthogonal projection with no more resampling. The ray composition section is therefore computed as quickly as in the shear-warp approach. In fact, this method can be viewed as resembling the shear-warp method where we let the texture mapping hardware perform the shearing and perspective scaling. Furthermore, our method does not require a final warp since the planes are already resampled into image space. This not only speeds up the total processing over shear-warp, but removes a filtering step and thus, results in higher image quality. Algorithm 1 renders a volume using the Planes method.

```
Load texture rotation matrix
Resample first plane into frame buffer
Read first plane from frame buffer into memory
loop over all remaining scanlines
  Resample next plane into frame buffer
  loop over all columns in previous plane
    Initialize integration variables
    while within bounds and ray not opaque
      Lookup opacity in tranfer function
      if sample opacity > threshold
        Lookup shade in reflectance map
        Composite ray color OVER sample
      end if
    end while
    Store ray color in previous row of image
  end loop
  Read next plane from frame buffer into memory
end loop
Shade and composite final plane as above
```

Algorithm 1: *Planes method for texture map based volume rendering*

Notice that we interlace the CPU and graphics hardware computation by initiating the texture mapping calculations for scanline $y+1$ before doing the shading and compositing on scanline y in the CPU.

Table 1 presents rendering times for various volumes and image sizes. The *Translucent Solid* is a 64^3 volume that is homogenous and translucent with a 1/255 opacity. This establishes a baseline of how long it takes to process an entire volume. Since there are no transparent voxels, every sample is shaded (i.e., the low opacity skipping optimization is not utilized). Additionally, the rays do not reach full opacity for 191 samples inside this volume, so for most cases in the table, the early ray termination optimization does not take effect. The *Translucent Sphere* is a radius 32 sphere of 4/255 opacity in the center of a transparent 64^3 volume. In this volume, the effect of the low opacity skipping optimization becomes apparent. The *Opaque Sphere* is the same sphere, but with a uniform opacity of 255/255. This is the first volume to take advantage of early ray termination and the rendering times reflect that. These first three volumes were created as theoretical test cases. The next three MRI and CT scanned volumes are representative of the typical workload of a volume rendering system. All three of these contain areas of translucent "gel" with other features inside or behind the first material encountered. Renderings of the *Lobster*, *MRI Head*, *Silicon* and *CT Head* datasets on a four processor SGI Onyx2 are shown in Figs. 5, 6, 7 and 8, respectively.

The image sizes cover a broad range (most are included for comparison to other methods; see Sec. 4). The number of samples along each ray is also included because the run time of image-order ray casting is typically proportional to the number of samples computed and not the size of the volume. To show this, we rendered the *Opaque Sphere* as a 32^3 volume in 0.13 seconds, as a 64^3 volume in 0.13 seconds, and as a 128^3 volume also in 0.13 seconds (for all of these we rendered 100^2 images with 100 samples per ray using the Planes method).

3.2 Blend: Compositing in the Frame Buffer

When we tested and studied the performance of the system we noticed that, depending on the volume data and transfer

Image Size $\times$ Samples per Ray	Translucent Solid 64^3	Translucent Sphere 64^3	Opaque Sphere 64^3	Lobster $128^2 \times 64$	Silicon 128×32^2	MRI Head 64×256^2	CT Head $128^2 \times 113$
$128^2 \times 84$	1.04	0.54	0.19	0.24	0.48	0.36	0.20
$200^2 \times$VolDepth	1.90	0.99	0.35	0.31	0.52	1.27	0.48
$200^2 \times 200$	5.55	2.69	0.73	1.22	2.21	1.76	0.67
$400^2 \times 128$	13.19	6.10	1.84	2.78	5.98	4.53	1.84
$512^2 \times$VolDepth	11.96	6.15	1.94	1.88	3.06	7.99	2.81

Table 1: *Renderings rates in seconds for the Planes method*

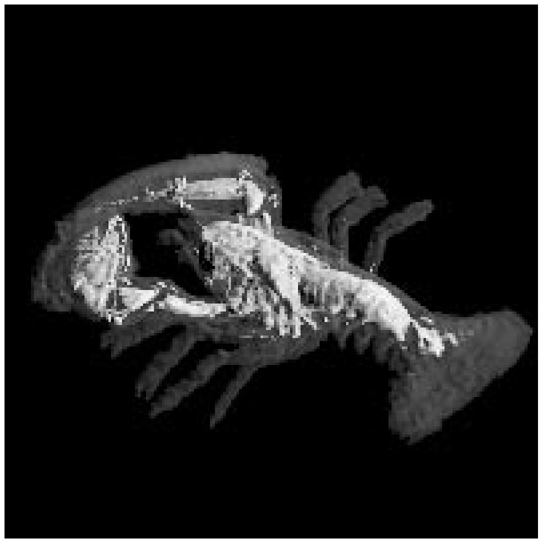


Figure 5: *CT scanned Lobster dataset with a translucent shell rendered in 0.26 seconds at 200^2 resolution (also in the color section)*

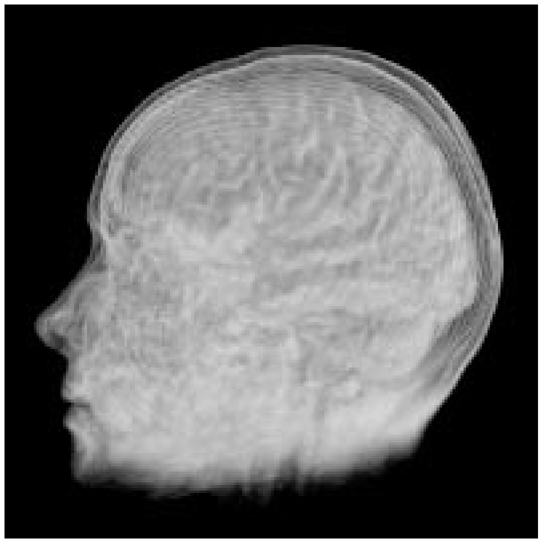


Figure 6: *MRI scanned Head dataset showing internal brain structures rendered in 0.43 seconds at 200^2 resolution (also in the color section)*

```
Load texture rotation matrix
Resample furthest slice into frame buffer
Read furthest slice from frame buffer into memory
loop over all remaining slices back-to-front
   Resample next slice into frame buffer
   loop over all samples in previous slice
      if sample opacity > threshold
         Lookup shade in reflectance map
         Write shade back into buffer
      else
         Write clear back into buffer
      end if
   end loop
   Blend slice buffer into frame buffer
   Read next slice from frame buffer into memory
end loop
Shade and Blend nearest slice as above
```

Algorithm 2: *Blend method for texture map based volume rendering*

function, there was still a substantial amount of time spent in the compositing portion of the algorithm. In fact, we found that the number of samples per ray before reaching full opacity and terminating is proportional to the time spent compositing. We propose to composite using the blending hardware of the graphics hardware by placing the shaded images back into the frame buffer and specifying the **over** operator. Of course, this requires that we return to using polygons that are parallel to the final image plane as in Fig 2. In this method, we can employ the optimization of skipping over low opacity samples by not shading empty samples. However, since the transparency values reside in the frame buffer's α channel and not in main memory, we can not easily tell when a given ray has reached full opacity and can not directly employ early ray termination without reading the frame buffer. Algorithm 2 renders a volume using this Blend method.

Notice that we now resample along slices rather than planes. Also, there are two frame buffers, one for the slices of samples and another for the blending of shaded images. Since compositing is not performed in software, it is quicker than the Planes algorithm. However, because of the added data transfer back to the graphics pipeline for blending into the the frame buffer, and the fact that shading is performed for all voxels, this method does not always produce faster rendering rates.

Considering Table 2, the Blend method always produces better rendering rates for the first two columns, due to the fact that here the volumes are "fully translucent". In other words, since the rays never reach full opacity, the early termination optimization that the Planes method typically uti-

Image Size $\times$ Samples per Ray	Translucent Solid 64^3	Translucent Sphere 64^3	Opaque Sphere 64^3	Lobster $128^2\times64$	Silicon 128×32^2	MRI Head 64×256^2	CT Head $128^2\times113$
$128^2\times84$	0.83	0.48	0.48	0.26	0.46	0.35	0.29
$200^2\times$VolDepth	1.48	0.88	0.83	0.23	0.38	1.21	0.84
$200^2\times200$	4.64	2.63	2.66	1.31	2.50	1.83	1.49
$400^2\times128$	11.30	5.40	3.19	3.04	5.13	13.49	3.44
$512^2\times$VolDepth	9.29	5.46	5.14	1.32	2.36	7.26	4.68

Table 2: *Rendering rates in seconds for the Blend method*

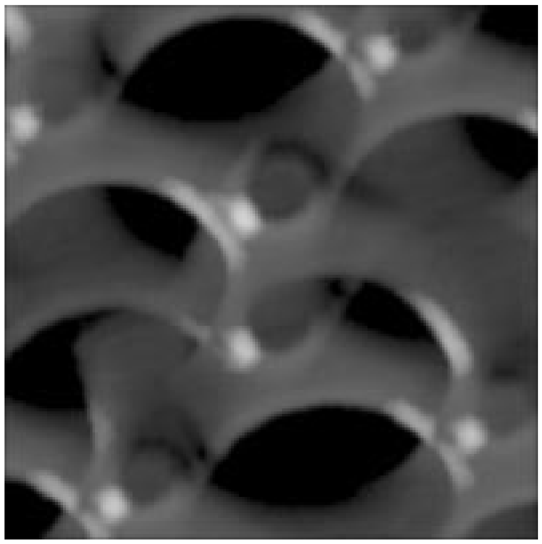


Figure 7: *Silicon dataset flythrough showing translucent surfaces rendered in 0.29 seconds at 200^2 resolution (also in the color section)*

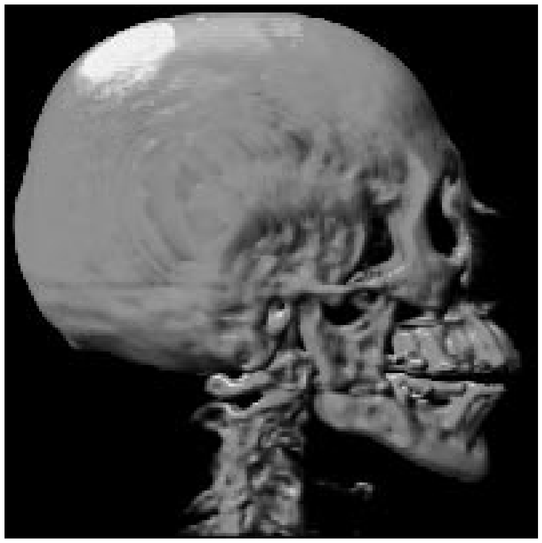


Figure 8: *CT scanned Head dataset showing bone structures rendered in 0.44 seconds at 200^2 resolution (also in the color section)*

lizes is unavailable. Since both methods must shade the same number of voxels and composite every sample on every ray, letting the graphics hardware perform this compositing is the quickest. However, for the *Opaque Sphere* the Planes method is always faster. This is because 78.5% of the rays intersect the sphere and the optimization from early ray termination is greater than the time gained from not performing compositing. We notice for the three "real" volumes, the Blend method is quicker when the number of samples along each ray is equal to the number of voxels in that dimension of the volume. When the sampling rate is close to the resolution of the image, the excessive slices that must be shaded and returned to the frame buffer again allow the early ray termination optimization in the Planes method to out-perform the Blend method.

In theory, which method would be optimal can be determined from the desired rendering parameters, volume density histogram, and transfer function. For a more opaque volume, the Planes method always produces better rendering rates. For transparent volumes, if there are many slices to render, the Planes method is usually quicker, while if there are few slices the Blend method is the better of the two. Yet in practice, we feel it may prove to be difficult to determine how "few" and "many" are defined. For this reason, we prefer the Planes method, since it is faster for all opaque volumes and for some of the translucent volumes.

4 Comparison to Other Methods

Here we compare the performance of our rendering algorithm to others presented in the literature, in terms of both image quality and rendering rates. The image quality comparisons point out quality trade-offs as they relate to lighting methods. We noticed in our testing on different volumes, that the number of samples more accurately determined the run time of the algorithm than simply the volume size. For this reason we have included image sizes and sample counts in our runtime tables (see Tables 1 and 2). We also noticed that the volume data and transfer functions greatly influence rendering rates. For our method this is probably more of an effect because we are utilizing runtime optimization techniques whose performance directly relies on the volume data and transfer functions.

Our Planes method renders the *Lobster* at a sampling resolution of 512×512×64 in 1.88 seconds. Our method is 19 times slower than the method of Cabral et al. However, their method does not employ directional shading or even view independent diffuse shading. This is a major limitation to their method since shading cues are highly regarded as essential to visual perception of shape and form. Our implementations with full Phong lighting with ambient, diffuse and specular components produces a much higher quality image.

In comparison to Cullip and Neumann [4], our method is again slower. Cullip and Neumann achieve better image quality than Cabral et al. by computing a gradient coefficient that is used to simulate diffuse highlights. This still is not as high an image quality as our full Phong lighting, and if the light geometry changes with respect to the volume, Cullip and Neumann's texture maps must be recomputed. Therefore, if the viewing geometry is dynamic, then our method obtains higher quality images, including specular highlights, at faster rates.

Our method produces an image of *Opaque Sphere* in 1.84 seconds with the Planes method, faster than Sommer et al.'s [13] isosurface rendering. For a translucent rendering of the *Lobster* our Planes method runs 12 times faster in 2.78 seconds. The image quality of both methods is equivalent since they both compute full lighting effects. As Sommer et al. pointed out, storing the gradients in the texture map has the disadvantage of limiting the size of the volume that can be rendered without texture paging, so our frame rate is limited by the amount of available texture memory, like all other texture map based methods.

Although Lacroute's shear-warp [9] is not a texture map based approach, we include a comparison, since it is one of the quickest methods for rendering with a full accurate lighting model on a workstation class machine. For example, shear-warp produces fully shaded monochrome renderings at a rate of 10 Hz, but, this is a parallel version of shear-warp running on a 32 processor SGI Challenge. Lacroute reports that a $128 \times 128 \times 84$ volume can be rendered in 0.24 seconds on one processor. Our Planes method renders a 128^2 image of the *Opaque Sphere* with 84 samples per ray in 0.19 seconds and the *Lobster* in 0.24 seconds. Our parallel implementation runs even faster (see Sec. 5). Since shear-warp must generate three copies of a compressed data structure per classification, interactive segmentation is not possible as is with our method. Shear-warp performs classification before bilinear resampling, whereas our method performs trilinear interpolation followed by classification. Additionally, our method performs arbitrary parallel and perspective projections in the same time while shear-warp takes up to four times longer for perspective projections.

5 Parallel Implementation

We have parallelized the Planes algorithm on a four processor Onyx 2 worksatition with Infinite Reality graphics. We constructed a master-slave model for the parallel processing where the master process implements the texture mapping interface to the graphics hardware and once a plane of orthogonal rays is resampled by the hardware, the work is farmed to a slave process for the raycasting. We use the shared memory symmetric multi-processor (SMP) functionality of the Onyx 2 and IRIX 6.4 operating system. The best speedup we can achieve with the parallel version is bound by the time it takes to perform the texture mapping for all the planes. This is because the texture mapping computation must be performed sequentially since there is only one graphics pipeline.

Figure 9(a) shows the rendering rates for one to four processors for various volumes. For all cases we rendered 128^2 images with 84 samples per ray. The time to perform the texture mapping of 128 128×84 planes is 0.12 seconds as shown on the graph. As can be seen, the rendering rates approach this theoretical best rendering time. Figure 9(b) presents speedup curves for the same datasets. The *Opaque Sphere* dataset is the rendered the fastest. However, it also achieves the poorest speedup because it quickly approaches the limit for raw rendering time imposed by the sequential texture mapping. On the other end of the spectrum, the *Translucent Sphere* achieves the best speedup performance although it suffers from the slowest rendering rates. This is because the CPU bound raycasting portion of the computation is the dominant percentage of the sequential time for this dataset. The *Lobster* dataset is representative of volume rendering applications and shows results between the two extremes.

Given enough processors, any dataset will eventually be limited by the time to perform the texture mapping. The number of processors required to reach this limit depends on the time it takes for the CPU portion (raycasting) of the algorithm to run and the fact that that portion relies heavily on software data dependant optimizations. The limit is reached when the number of processors is equal to T_c/T_t, where T_c is the time to perform the raycasting for one plane on the CPU and T_t is the time to texture map one plane in the graphics hardware.

6 Results and Conclusions

We have presented a method for high-quality rendering of volumetric datasets which utilizes the 3D texture map hardware currently available in graphics workstations. The method produces images whose quality is not only comparable to that of accurate software ray casting, but also the highest quality method currently available, at a substantially faster frame rate than that of software ray casting. Other methods achieve higher frame rates than ours, but either lack shading, lack directional shading, or require multiple processors.

Our method is accelerated by multiple processors, although the speedup is limited by the throughput of the serial graphics pipeline. Although shear-warp achieves higher rendering rates for multiprocessor machines, our method is faster on typical graphics workstations with 3D texture mapping and also supports interactive classification.

7 Acknowledgements

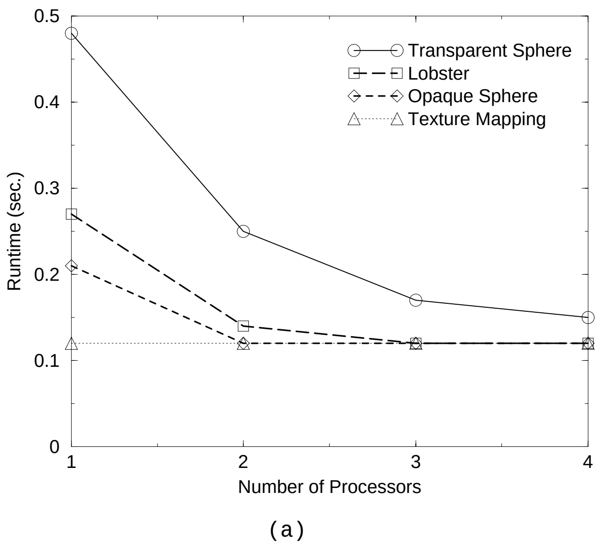
This work has been supported by the National Science Foundation under grant MIP9527694, Office of Naval Research under grant N000149710402, Mitsubishi Electric Research Lab, Japan Radio Corp., Hewlett-Packard, and Intel Corp. The Lobster dataset is courtesy of AVS, Inc. The Silicon dataset is courtesy of Oak Ridge National Lab. The MRI Head dataset is courtesy of Siemens. The CT Head is from the UNC database.

References

[1] K. Akeley. RealityEngine Graphics. In *Computer Graphics, SIGGRAPH '93*, Anahiem, CA, August 1993. ACM.

[2] R. Avila, L. Sobierajski, and A. Kaufman. Towards a Comprehensive volume Visualization System. In *Proceedings of Visualization '92*, Boston, MA, October 1992. IEEE.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Symposium on Volume*



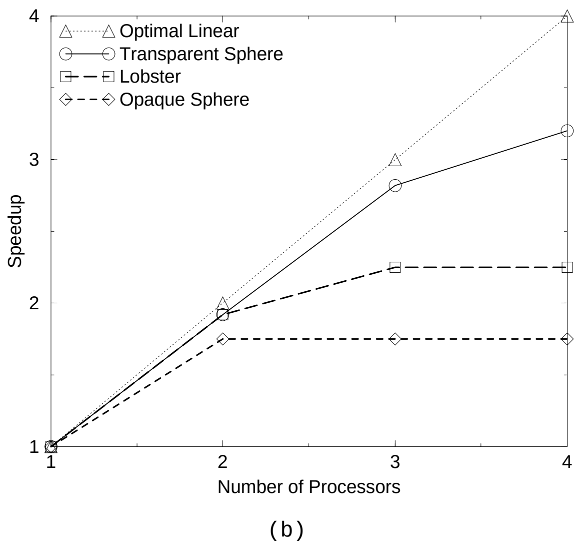


Figure 9: *(a) Parallel rendering rates for 128×128×84 samples and (b) Parallel speedup*

Visualization, pages 91–98, Washington D.C., October 1994. ACM.

[4] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill, 1993.

[5] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The Realization. In *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68, Los Angeles, CA, August 1997. Eurographics.

[6] A. Van Gelder and K. Kim. Direct Volume Rendering with Shading via Three-Dimensional Textures. In *Symposium on Volume Visualization*, pages 23–30, San Francisco, CA, October 1996. ACM.

[7] S.-Y. Guan and R. Lipes. Innovative volume rendering using 3D texture mapping. In *Image Capture, Formatting and Display*, Newport Beach, CA, February 1994. SPIE.

[8] M. J. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. In *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 45–56, Los Angeles, CA, August 1997. Eurographics.

[9] P. Lacroute. Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, September 1996.

[10] P. Lacroute and M. Levoy. Fast Volume Rendering using a Shear-warp Factorization of the Viewing Transform. In *Computer Graphics, SIGGRAPH '94*, pages 451–457, Orlando, FL, July 1994. ACM.

[11] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(5):29–37, May 1988.

[12] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Computer Graphics, SIGGRAPH '97*, pages 293–302, Los Angeles, CA, August 1997. ACM.

[13] O. Sommer, A. Dietz, R. Westermann, and T. Ertl. Tivor: An Interactive Visualization and Navigation Tool for Medical Volume Data. In *The Sixth International Conference in Central Europe on Computer Graphics and Visualization '98*, February 1998.

[14] J. van Scheltinga, J. Smit, and M. Bosma. Design of an On-Chip Reflectance Map. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware '95*, pages 51–55, Maastricht, The Netherlands, August 1995. Eurographics.