# Data Parallel Volume Rendering Algorithms for Interactive Visualization

**Peter Schröder*** **Wolfgang Krüger**

Scientific Visualization Department of HLRZ
Gesellschaft für Mathematik und Datenverarbeitung
Sankt Augustin, Germany
{ps,krueger}@viswiz.gmd.de

**Abstract**

The increasing availability of parallel computing platforms has led to the development of parallel volume rendering algorithms. In the present paper we compare two algorithms for volume raytracing in a data parallel framework: a shearing technique and a line drawing technique. The two algorithms are primarily distinguished by the level of parallelism they exploit. Both algorithms have been implemented on the Connection Machine CM2 massively parallel computer and execute at speeds suitable for interactive volume rendering applications. Since considerable floating point resources are available on the CM2 we have used rendering algorithms based on transport theory. In the second part of the paper we examine some of the trade-offs involved between image quality and rendering speed when using high fidelity rendering algorithms.

**Keywords:** Volume Rendering, Massively Parallel, Transport Theory

*Author's current address: Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544

# 1 Introduction

With the increasing use of massively parallel computers in scientific applications the need for volume visualization on these platforms has increased. At the same time it has become desirable to make volume visualization interactive to match the speed of the scientific computations themselves and allow for such paradigms as interactive steering. While relatively fast algorithms for volume raytracing exist for workstations (see for example [Levoy, 1990; Westover, 1990; Laur and Hanrahan, 1991]) only little work has been published describing algorithms which run on massively parallel platforms [Schröder and Salem, 1991; Yoo *et al.*, 1991; Neumann, 1992; Schröder and Stoll, 1992].

When developing algorithms for interactive volume visualization on massively parallel hardware platforms a number of basic properties need to be taken into account. In our case the following three features stand out

- **SIMD data parallel programming model** due to the target of our implementation, the Connection Machine CM2. As a consequence it is difficult to incorporate such optimizations as early view ray termination without requiring a large communication overhead to perform the desirable load balancing.

- **Distributed memory** and consequently the relatively small amount of memory per processor. This precludes such measures as duplication of the data set at every processor[1]. Instead the distribution of data and the communication implied by it needs to be considered explicitly.

- **Cost of communication** is considerably higher for irregular (general) than for regular patterns. When communication cannot be avoided, such as when accessing values of voxels not held in a processor's local memory the algorithm should attempt to exploit all regularity in the access patterns.

Since the main goal in the development of the algorithms was interactivity it is tempting, for example, to account for low level features of the communication system or the specific number of processors in a given computer. However, we also consider scalability an important goal, since we

[1] Duplication at every processor is not desirable in any case since the amount of wasted memory grows with the number of processors. For machines with small numbers of processors duplication may, however, be a viable alternative.

are interested in algorithms which continue to give good performance as the size of future parallel machines increases. This is reflected in the fact that we use abstractions such as *virtual processors* (see below) and differentiate the cost of communication only between regular and irregular, rather than considering specific communication patterns vis-a-vis a given interconnect topology.

In the present paper we review two algorithms for data parallel volume raytracing. One is based on shearing techniques, the other on the idiom of line drawing. These algorithms have been implemented on the CM2 massively parallel super computer and we will analyze their performance and discuss some of the distinguishing features of the algorithms. For both algorithms we use a rendering technique based on the transport theory model introduced in [Krüger, 1991] and analyse various speedup techniques useful in interactive applications.

We begin with a short overview of the CM2 architecture and volume rendering in general. This is followed by one section each describing the two algorithms and a section comparing their respective features. In Section 2 we describe the rendering modes used and give examples of various short cuts in the evaluation of the implied path integrals with performance figures and images.

### 1.1 CM2 architecture overview

We have implemented the algorithms on a CM2 computer. The CM2 consists of 4k to 64k bitslice processors with one floating point unit for every 32 processors. The processors are organized in a hypercube with a toroidal topology [Hillis, 1985; Thi, 1987]. Each processor has from 256 kilo bits to 1024 kilo bits of local memory. The programming model supports *virtual processors*, allowing the programmer to abstract from the actual number of physical processors available. The mapping of virtual processors to physical processors is performed by the runtime system, allowing the same executable to run on machines with differing numbers of processors. The particulars of this mapping can be influenced by the programmer based on expected communication frequency. The concept of virtual processors is especially useful for volume rendering since volumes generally have more than 64k voxels. This feature of the runtime environment allows us to formulate the algorithm and its implementation independent of the actual number of processors available in a given system. The CM2 supports high level lan-

guages such as C* (a data parallel extension of C), which we used in our implementation. Using lower level access to the machine could boost absolute performance, but would leave us with code which would be difficult to adopt to changing machine sizes.

### 1.2 Volume rendering

Previous approaches to volume rendering can be classified into three groups

- Ray tracing, with two variants
    1. Object space methods, which resample the volume at sample points along rays [Levoy, 1988].
    2. Image space methods, which transform the volume to be axis aligned with the view direction [Drebin *et al.*, 1988].
- Splatting [Westover, 1989; Westover, 1990; Laur and Hanrahan, 1991]
- Cell projection [Wilhelms and Gelder, 1991; Max *et al.*, 1990; Shirley and Tuchman, 1990]

While both splatting and cell projection can have efficient parallel implementations we are concerned only with ray tracing methods in the present discussion. In the following two sections we describe two parallel ray tracing algorithms: an image space algorithm and an object space algorithm. The description is followed by a discussion of some of the advantages and disadvantages of the two methods.

### 1.3 Image space

In image space methods the volume is suitably transformed and resampled given a particular view point. This is generally accomplished by using multi-pass transformations to rotate and scale the volume as a set of slices, before applying the *over* operator to yield the final image. Schröder and Salem used this approach in [Schröder and Salem, 1991] for a massively parallel volume rendering algorithm on the Connection Machine. They use a shearing technique to transform and resample the volume such that the view rays are in effect parallel to one of the coordinate axes. The particular shearing technique used is based on pure shears [Paeth, 1986; Tanaka *et al.*, 1986] as opposed to shear/scale transforms [Hanrahan, 1990; Smith, 1987]. The

decomposition of a single axis rotation into pure shear transformations is given by

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} 1 & -\tan\frac{\theta}{2} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin\theta & 1 \end{pmatrix} \begin{pmatrix} 1 & -\tan\frac{\theta}{2} \\ 0 & 1 \end{pmatrix}$$

as distinguished from a two pass decomposition

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \tan\theta & \cos 2\theta \sec\theta \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ 0 & 1 \end{pmatrix}$$

For each row of the image the latter causes a scale and a shift, while the former only affects shifts. Assume for a moment that a given row of the image is laid out across a linear array of processors with one pixel per processor. A scale/shear transformation will cause a constant shift due to the shear part and a variable shift (based on the x coordinate of a given pixel) due to the scale part, causing each pixel to move by a *varying* amount. On the other hand if there is only a shear, i.e. a constant shift amount for an entire row, all processors can move their pixels over by a *constant* amount. This argument holds for each row in the image individually and thus for all of them. Moving data by a constant amount along an axis of a processor array qualifies as regular communication and can be executed much faster than the irregular communication of varying shifts. Executing an orthogonal viewing transformation thus becomes a sequence of single axis shears, three for each rotation axis, for a total of nine shears. Some of those can be combined without destroying the constant shift per row property. In practice it is typically faster to execute only the first two viewing rotations on the volume itself and apply the last rotation in the image plane, where only a number of processors proportional to the image size is involved. With these arguments in mind the implementation in [Schröder and Salem, 1991] uses five shears on the volume and three shears in the image plane to affect any orthogonal viewing transform.

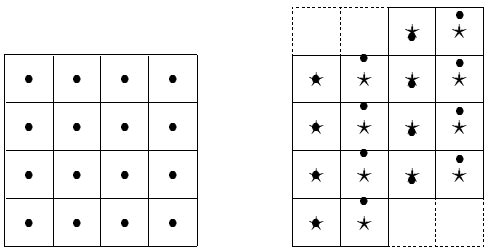

**Figure 1**: *Example of shearing a slice of the volume along its columns. The new sample points are indicated by asterisks while the sample values before the shear are shown as solid dots. Subpixel resampling is needed to compute the sample values after shearing.*

Since the shear amounts are typically non-integral, some subvoxel resampling needs to be performed to minimize aliasing artifacts (see Figure 1). The CM2 implementation used linear resampling between neighbors along each of the shear directions with satisfactory results. For linear resampling only the immediate neighbors need to be fetched, again, an operation which implies only regular communication. In a SIMD framework these operations are expressed rather simply since one need only conceptualize a single voxel when writing out the algorithm. The semantics of data parallelism insure that the "right thing" happens everywhere in the volume.

The implementation in [Schröder and Salem, 1991] did not actually move the data during the shear steps, but instead applied the equivalent computation to the associated 3D coordinates of each voxel. In this way no motion of data occurs during the resampling steps. As a result the data is properly resampled after the shearing steps but the view rays have not been aligned physically with one of the axes of the processor array.

Once the shading has been computed for the resampled volume one still needs to perform the compositing operation along the view rays. For this fast parallel prefix operators [Blelloch, 1990] may be used, which evaluate the *over* operator typically used for compositing in $O(\ln_2 n)$ time, where $n$ is the number of samples along the ray. On the CM2 these operators are only implemented along processor array axes and not along arbitrary chains through volumes. Thus the CM2 implementation needed to reorder the resampled values to be axis aligned. This latter step requires general communication.

| Machine size | Shear time | Reorder time | Render time | Total time |
|---|---|---|---|---|
| 8k | 1261 | 804 | 417 | 2482 |
| 16K | 633 | 400 | 213 | 1246 |
| 32k | 318 | 193 | 109 | 620 |
| 64k | 171 | 97 | 56 | 324 |

**Table 1**: *All times are given in ms for a dataset of* $128^3$ *voxels and measured on a CM200 super computer of the given size. The CM200 is a faster clock rate version of the CM2.*

This choice of trading the regular shifts during shearing against one general communication step at the end was motivated by initial performance estimates and our desire to stay with a high level implementation language rather then program the routing hardware directly. Even though the resampling steps only require regular communication there are a considerable number of them, while the final reordering is a single one and itself has regularity which is exploited in a transparent fashion by the routing algorithms. If one were to implement a similar algorithm on different SIMD hardware and with different runtime support the choice may well be to perform the individual shifts during resampling and avoid the final reordering.

The final program still spends about 50% of its time with routing but achieves multiple frames per second update rates. Table 1 gives timing results for a data set of $128^3$ voxels with shading performed after resampling. The times are differentiated by time spent in the shearing part of the algorithm, which includes all resampling; the final reordering time (general communcation step); shading and compositing (rendering) time. The times given do not vary with the data and are independent of rotation angle. The number of terms in the rendering equation, such as the number of isosurfaces does influence the rendering time. For the times quoted in Table 1 the rendering included volume and isosurface sources, volume absorption, and attenuation by one external light source.

## 1.4 Object space

In object space methods the volume is left in its local coordinate frame and instead rays are subjected to the inverse viewing transform. When tracing a single ray through the volume we are interested in evaluating a path integral along that ray [Krüger, 1991]. This is typically accomplished by discretizing the integral into a sum based on some step size along the ray. In this way each ray parameterizes a set of sample points along its length, some of which will lie inside the volume. In order to resample using tri-linear interpolation the eight grid points surrounding a given ray-sample need to be accessed. Since the mapping of samples onto voxels is not one-to-one—some voxels contain more than one ray-sample, others none—a SIMD implementation appears to require general communication. By using the idiom of line drawing on discrete grids [Braccini and Marino, 1980] and suitably chosing lines we can however give a mapping of ray-samples onto the volume, which is one-to-one.

In this section we describe the object space algorithm which was implemented by Schröder and Stoll [Schröder and Stoll, 1992] on the CM2 and the Princeton Engine. The current discussion will focus on the CM2.

The algorithm is based on the basic observation that a careful choice of ray alignment in the image plane leads to a sequence of ray-steps through the volume such that the volume is perfectly tiled by the rays. Furthermore, each ray is a translated version of a protytype ray with exactly the same stepping pattern through the volume. Yagel and Kaufman made this observation when they considered the design of custom hardware for volume rendering [Yagel, 1991] and reported on a software implementation on a serial machine in [Yagel and Kaufman, 1991; Yagel and Kaufman, 1992]. Cameron and Undrill [Cameron and Undrill, 1992][2] independently used the same idea for a volume rendering algorithm on the AMT DAP massively parallel computer.

To simplify the discussion we will explain the algorithm in 2D. All arguments used readily generalize to the 3D case. Each ray starts in the image plane and steps towards the volume. If we start the rays with a spacing in the image line (plane) such that their *vertical* distance is 1 (see Figure 2), then all rays will enter the frontmost column (face) of the image (volume) with the same pixel (voxel) local coordinate. We ignore for a moment those

[2] In this article they also describe their implementation of the image space method described above.

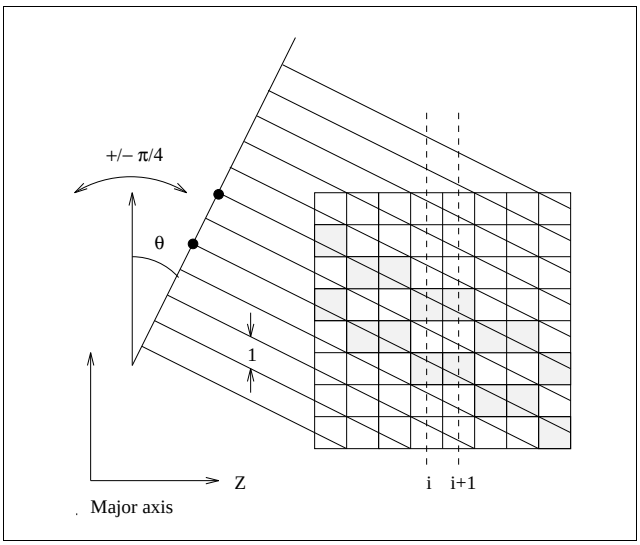


**Figure 2**: *Line drawing in 2D. All rays start from the image "line" and are chosen such that they intersect the outermost column of pixels at the centers. Each ray enumerates the voxels it steps through. An example for two non-adjacent rays is given with the shaded pixels. All rays move in lock step along the major axis. The sample points at step $i$ and $i+1$ respectively are given by the intersection of the dashed lines with the rays.*

rays which do not enter the image in the front (such as the rays on the top face in Figure 2). As a consequence the rays are spaced tighter than unit distance in the image line and the pixels they pierce perfectly tile the image without holes or overlaps (see the two non-adjacent rays whose pixels are grayed in Figure 2). Generating a rendering in this way gives us an oblique image which only needs to be rescaled in the image plane to yield the correct picture. Above we described the image space method as a particular decomposition of the viewing rotation matrix. Similarly for the object space method we get a decomposition of the following form

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

Where we have used letters to denote the products of sin and cos terms which arise when considering an arbitrary 3D rotation. The right side now consists of a scaling matrix in the image plane (x and y coordinates) times a transformation which shears the x and y coordinates and rotates the z coordinate.

In the following description we will always assume that the major axis of the projection is the Z axis. If this is not the case a cyclic permutation of the axes maps any other case into this one. In the CM2 implementation these permutations are achieved by a transpose operation. For hypercubes optimal algorithms exist to affect this transposition. In Figure 2 a sequence of sample points at step $i$ and $i + 1$ is indicated by the intersection of the dashed lines with the rays. Note that they all have the same pixel (voxel) local coordinate. Assuming that the major axis of the projection is layed out as an in-processor array the algorithm simply steps along this dimension with unit strides and possibly executes a shift of 1 in the orthogonal directions. Notice that once again this only implies regular communication since all rays proceed in lock step.

So far we have neglected the rays which do not enter the volume on the front most face as well as the question of what happens to those rays leaving the volume—due to the above mentioned orthogonal shifts—before reaching the far end. The answer is based on the observation that rays entering the image on the top (see Figure 3) correspond in a one-to-one fashion to rays leaving on the bottom. Conceptually we tile space with copies of the image (volume). The actual code does not create any copies, instead the same

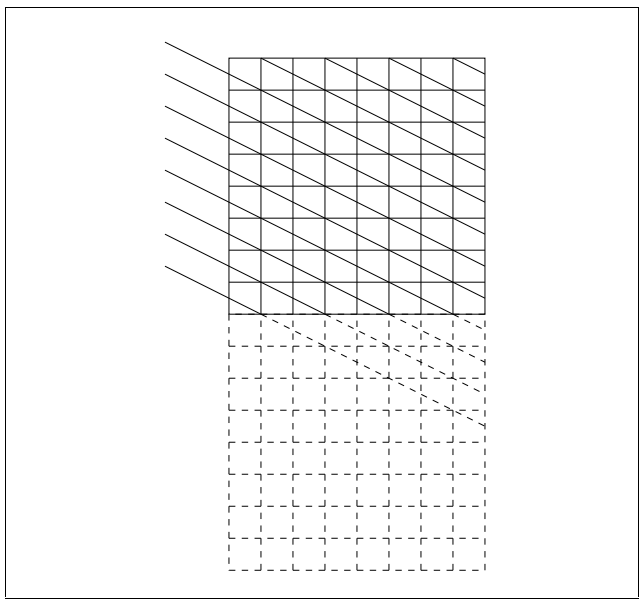

**Figure 3***: When only considering rays entering through the front most face, the rays entering at a later point (on top) correspond to initial rays leaving through the bottom in a one-to-one fashion. This can be understood as continuing the exiting rays through a periodic tiling of the original set of pixels (voxels).*

| Machine size | Render time | Rescaling time | Total time |
|---|---|---|---|
| 8k | 1256 | 34 | 1290 |
| 16K | 773 | 20 | 793 |
| 32k | 603 | 14 | 617 |
| 64k | 431 | 14 | 445 |

**Table 2**: *All times are given in ms for a dataset of* $128^3$ *voxels and measured on a CM200 super computer of the given size.*

effect is achieved by simply taking advantage of the toroidal connectivity of the processor interconnect. With this concept in mind we see that there are only rays which start on the frontmost face and continue until they exit the far face. In the 2D example any ray has at most two parts—an initial segment—and then, if and when it leaves—a segment corresponding to a new ray entering. For volumes there are at most three segments to a ray, since a ray can pierce up to three copies of the volume before exiting the far plane.

Since we do not perform a separate resampling of the volume, the shading and compositing (evaluation of the path integrals) are performed incrementally at every step. When all path integrals have been computed in this fashion we still need to apply the final image rescale. In the CM2 implementation this is implemented using general communication, but the rescale has enough regularity that it can be expressed as a sequence of regular communication steps [Schröder and Stoll, 1992].

Table 2 shows some representative performance timings for the ray parallel method. The timings are for a rotation of 30 degrees. While the shearing algorithm is insensitive to the rotation angle, the line drawing method computes more samples along a diagonal ray. We chose 30 degrees for the timings since it represents the average case over all rotation angles. The *Render time* includes all interpolation, shading, and compositing computations. In Table 1 we have treated *Shearing* (interpolation) and *Rendering* (shading and compositing) separately. For small machine sizes (up to 16k processors) there are at least as many rays as there are physical processors and the timings are considerably better for the object space method. Once there are more processors than rays the line drawing algorithm cuts the rays

along their length into sub extents to keep all processors busy. This incurs extra overhead during interpolation and requires a compositing step of the sub extents at the end. For a $128^3$ volume on a 64k processor machine there are 4 such subextents to each ray. Due to this we do not see the linear scaling observable in Table 1. For a $256^3$ data set on a 64k processor machine we get a total time of 1680 mS, which exhibits the expected scaling vis-a-vis the time for a $128^3$ data set on a 16k processor machine. Due to memory limitations in the shearing algorithm (see the discussion below) we were not able to run both sets of timings for $256^3$ data sets.

### 1.5 Comparison of methods

We will discuss four differences between the methods in turn

- Asymptotic scaling properties
- Memory consumption
- Decoupling of rotation and rendering
- Cost of different filters

The main distinguishing feature of object and image space parallel volume rendering algorithms is the level of parallelism they exploit. The image space method (shearing) exploits voxel parallelism, while the object space method (line drawing) exploits ray parallelism. In voxel parallelism the number of processors can grow with the number of voxels in a given volume, with an execution time proportional to $O(\ln_2 n)$, where $n$ is the side length of the volume with $n^3$ voxels total. In ray parallelism the execution time is proportional to the length of the rays, $O(n)$, and the number of processors can only grow with the cross section of the volume. This analysis holds for the virtual processor paradigm, in which the algorithm is always formulated as if there were as many processors as demanded by the particular level or parallelism exploited. For practical machine sizes however, in our case up to 65536 processors, the number of *physical* processors is effectively on the order of the number of rays. Hence the runtime system maps so many virtual processors on each physical processor that the two approaches tend to yield comparable run times.

Memory requirements are very different for the two approaches. When exploiting voxel parallelism, semantically each operation is performed everywhere in the volume at once, hence every intermediate variable must be

instanced at each voxel[3]. The implementation of the shearing algorithm has one `float` variable for the original data, three `float` variables for the gradient (used in shading), and four `float` variables for the resulting r, g, b, and $\alpha$ values. On top of this any intermediate variable consumes memory at a rate of $n^3$. This led to an implementation which could only fit a volume of $128^3$ on a 4k processor CM2 with 256 kilo bits per processor, giving a ratio of 1/16 for volume storage to total memory. In ray parallelism, on the other hand, storage during rendering is only consumed at a rate proportional to the number of rays. In the CM2 implementation the orginal volume and its gradient were still maintained at the volume level, but any data structures during rendering were only instanced a plane at a time. When computing the gradient on the fly a $256^3$ would fit on a 4k CM2 with 256 kilo bits of memory.

In the context of interactive visulatization the decoupling of rotation and rendering in the shearing algorithm can make a marked difference. A typical interaction scenario consists of selecting a view followed by repeated adjustments to the rendering parameters. Once the volume has been rotated, very fast repeated renderings are possible—17Hz (56 mS) on a 64k CM200 computer for a volume of $128^3$ voxels (see Table 1). This separation does not occur in the line drawing method since resampling and rendering are intimately linked and every change in rendering parameters incurs the entire cost of the algorithm. If both view and rendering parameters are changed this advantage can not be exploited.

In the shearing algorithm only a linear resampling filter was employed. In theory higher order filters could be used as well but they would be very difficult to realize in the current implementation. Since the CM2 implementation does not actually move the data during shearing but only manipulates the coordinate values accordingly the spatial neighbor relationships between voxels get "warped", requiring careful analysis to keep track of neighbors. In [Schröder and Salem, 1991] this analysis was performed only for immediate neighbors and proved fairly complicated. Keeping track of a larger neighborhood and its apparent motion under the same algorithm would be very difficult indeed. This is contrasted with the object space method, which leaves the volume undisturbed. In practice this means that neighbors with respect to a particular sample point are always easy to access.

[3] An implementation on a specific number of processors may be able to achieve the correct semantics while instancing variables only at each *physical* processor, rather than each voxel.

In summary, the shearing technique consumes more memory, only allows linear resampling, and rerenders the volume considerably faster when the view is held constant. The line drawing method has less favorable asymptotic scaling properties but performs faster overall for machine sizes with less physical processors than rays.

## 2 Rendering

In the previous sections we have described how the appropriate samples of the volume can be accessed in an efficient manner for volume ray tracing. In this section we discuss the transport theory model which forms the basis of the rendering and shading module in both algorithms.

Krüger has shown in [Krüger, 1991] that all known volume rendering modes can be understood as specializations of an underlying transport theory model of light propagation in a participating medium. Using this model the light impinging on the image plane is described as a path integral along the view ray

$$I(x,y) = \int_{S_{near}}^{S_{far}} Q(S) e^{-\int_{S_{near}}^{S'} \sigma_a(S') + \sigma_{sc}(S')\, dS'}\, ds \qquad (1)$$

$$Q(S) = q(S) + \sigma_{sc}(S) \int \rho_{sc}(\vec{\omega}' \rightarrow \vec{\omega}) I(S, \vec{\omega}')\, d\vec{\omega}' \qquad (2)$$

These equations are understood to be evaluated separately for each of r, g, and b wavelength.

The intensity in the image plane $I(x,y)$ (Equation 1) is described as the emission at each point $S$ along the ray scaled by the optical depth to the eye. The optical depth is a function of the total extinction coefficient $\sigma_t = \sigma_a + \sigma_{sc}$ composed of absorption and scattering coefficients. The generalized source $Q(S)$ (Equation 2) consists of the emission at a given point $q(S)$, typically a volume source and/or surface source, as well as an integral over all directions of the incoming intensity along those directions scaled by the scattering phase $\rho_{sc}$. In order to make the evaluation of this path integral manageable a low albedo approximation is generally performed. When assuming a low albedo the integral in Equation 2 reduces to a sum over the set of user supplied light sources.

It is important to note at this point that while the equations above are motivated by physical considerations and play to the physical intuition of the user, there is nothing particularly physical about viewing scalar fields

which do not correspond to light emission of physical objects. As a consequence we are not so much interested in "physical realism" as in mapping the parameters onto meaningful visual quantities, which facilitate the extraction of important information. For example, the numerical exactness of the computation of the above integrals is not as important as sampling considerations are.

## 2.1 Evaluation of the path integrals

In order to evaluate the path integrals we discretize them into an Eulerian sum (other rules such as Trapezoid are also easily implemented) and evaluate them front to back. This can be written as an iteration

$$I_n = \sum_{i=0}^{n} Q_i \prod_{k=1}^{i} \alpha_k \tag{3}$$

or as a recursion

$$\begin{aligned} I_0 &= 0 \\ \tilde{\alpha}_0 &= 1 \\ I_{i+1} &= I_i + (1 - \tilde{\alpha}_i) Q_{i+1} \\ \tilde{\alpha}_{i+1} &= \tilde{\alpha}_i \alpha_{i+1} \end{aligned} \tag{4}$$

where $Q_i = Q(S_{near} + i\Delta S)\Delta S$, $\alpha_i = \exp(-\sigma_t(S_{near} + i\Delta S)\Delta S)$, and $\Delta S = (S_{far} - S_{near})/n$.

When exploiting voxel level parallelism logarithmic time parallel prefix operators can be used to rapidly evaluate Equation 3. This was the approach taken in [Schröder and Salem, 1991]. For ray level parallelism the equivalent recursive formulation (Equation 4) is better suited, since we are stepping along the length of the rays in a serial fashion. The latter was used in [Schröder and Stoll, 1992].

For the evaluation of the above equations it is important to note that the multiplicative factor of $\Delta S$ in the definition of $Q_i$ is simply an overall scaling of the intensity. This is not true for the definition of $\alpha_i$. However so long as we use an arbitrary but *fixed* stepsize for a given rendering the mapping of $\sigma_t$ onto $\alpha_i$ can be performed outside the main loop, reducing the recursion to only a few multiplies and adds (recall that $I$ is typically computed for three representative wavelengths).

Since the sample points along the ray which arise from the discretization of the integrals in general do not lie at the original grid locations of the data some interpolation scheme has to be employed to find the values at the sample points from the values at the grid locations. Both the shearing method and the line drawing method used tri-linear interpolation, although for very demanding circumstances it may be desirable to use tri-cubic interpolation. Care must be taken to not smooth over important detail when employing higher order approximations. The advantage of tri-linear interpolation in the SIMD framework is the fact that only immediately neighboring processors need to participate in its computation. Higher order interpolation schemes need to access increasingly larger neighborhoods of points, albeit still in a regular fashion.

### 2.2 Parameter mappings

Our interest in the quickest turn around time between parameter adjustments and image delivery has led us to examine the trade-offs between image quality and rendering speed. High image quality as defined by

- preservation of detail
- minimization of artifacts
- combined visualization of different data features

depends in an essential way on the sampling and interpolation techniques, as well as on appropriate mappings of the interesting data features onto visualization parameters.

In the performance section below we compare results obtained by mapping data features onto the common source-attenuation terms in Equations 1 and 2 with the additional influence of surface source and reflection terms. Especially for an improved depth perception of iso surfaces these surface terms appear to be useful.

## 3 Performance

In order to explore the tradeoffs between performance and picture quality we have used two example data sets. The first is a medical CT data set of a human lumbar spine (Figures 4 to 9) and the second an industrial

| Interp. time | Shad. time | Integr. time | Rescal. time | Total time | M smpl. per sec. |
|---|---|---|---|---|---|
| $\Delta S = 1/4$ | | | | | |
| 13.5 | 14.46 | 1.05 | 0.27 | 29.28 | 4.37 |
| $\Delta S = 1/2$ | | | | | |
| 6.74 | 7.26 | 0.55 | 0.26 | 14.81 | 4.32 |
| $\Delta S = 1$ | | | | | |
| 3.36 | 3.65 | 0.28 | 0.25 | 7.54 | 4.24 |
| $\Delta S = 1$, no renormalization | | | | | |
| 3.36 | 2.77 | 0.28 | 0.25 | 6.66 | 4.80 |
| $\Delta S = 1/2$, no specular component | | | | | |
| 6.74 | 6.09 | 0.55 | 0.26 | 13.66 | 4.68 |
| $\Delta S = 1$, no specular component | | | | | |
| 3.36 | 3.06 | 0.28 | 0.25 | 6.95 | 4.60 |

**Table 3**: *Timings of the various components of the algorithm: interpolation (tri-linear), shading, integration (compositing), rescaling (in the image plane), and total time. The last column gives the aggregate number of mega samples per second. All timings are in seconds for the lumbar spine dataset ($128^3$ voxels) on an 8MHz CM2 with 16k processors and an image size of $512^2$.*

CT data set of an engine block (Figures 10 to 12). All timings in this section were made on a 16k processor CM2 with 32bit floating point support running at 8MHz clock frequency with the ray parallel algorithm. The dataset sizes were $128^3$ in both cases and all computations were performed in single precision floating point arithmetic.

### 3.1 Lumbar spine

For Figures 4 to 9 the rendering mode consisted of a volume source term to show the disks between the vertebrae and a surface source term at the interface to bone, to show the vertebrae themselves. The surface source term

was modeled as an exact iso surface with a diffuse shading component in the direction of the viewer and a specular component (Figures 4 to 7) due to an external, unshadowed light source. The scattering phase function is that of Phong. The gradient field necessary for the shading model was precomputed as a fourth order centered difference and interpolated tri-linearly to the sample sites. Supersampling in X/Y was 4/4 for an image size of 512 by 512 pixels.

The first set of experiments concerned the choice of stepsize. Figure 4 shows a rendering with $\Delta S = 1/2$ gridsize. The difference to a rendering with $\Delta S = 1/4$ (Figure 6) is barely perceptible, but the cost for the latter doubled (see Table 3). Setting $\Delta S = 1$ shows some artifacts due to the larger error in the determination of the iso surface location and consequently the orientation of the local surface gradient (Figure 5). In this case the time is halved at a relatively small loss of image quality acceptable during interactive selection of the proper rendering parameters. These artifacts increase when the gradient is not renormalized after interpolation (Figure 7), another common acceleration technique which does not yield perceptable artifacts at lower sampling rates. Removing the specular term from the shading equation increases the performance once again (Figure 8 and 9). Figure 8 shows a stepsize of $\Delta S = 1/2$, while Figure 9 uses $\Delta S = 1$.

### 3.2 Engine block

The second set of experiments used the engine block CT data set (Figures 10 to 12). This time the rendering mode does not use any exact iso surfaces, but rather gradient magnitude surface sources with one external lambertian light source. The mapping of the original scalar data field onto the rendering parameters is performed by a lookup table of 256 entries. Using a lookup table of modest resolution does give rise to some quantization artifacts, which can be distinguished in Figure 11 as subtle checker board patterns. The lookup table size can easily be increased without additional cost save for a minimal increase in memory. However, we have not found this to be necessary in practice since supersampling is typically not performed at the high rates used in these examples.

Since the aspect ratio of this data set is 1/2/2 we supersample more in X (Figure 11 and 12) for a total of 8 by 4 supersamples. The stepsize along the rays is $\Delta S = 1$ since the spatial detail in this data set is not high enough to warrant a step size smaller than the gridsize. In contrast

| Interp. time | Shad. time | Integr. time | Rescal. time | Total time | M smpl. per sec. |
|---|---|---|---|---|---|
| Image size $512^2$ | | | | | |
| 3.36 | 3.30 | 0.38 | 0.42 | 7.46 | 4.29 |
| Image size 1024 by 512 | | | | | |
| 7.09 | 6.96 | 0.79 | 4.06 | 18.90 | 3.38 |
| Image size 1024 by 512, pre shading | | | | | |
| 7.09 | 0 | 0.79 | 4.06 | 11.94 | 5.36 |

**Table 4**: *Timings of the various components of the algorithm: interpolation (tri-linear), shading, integration (compositing), rescaling (in the image plane), and total time. The last column gives the aggregate number of mega samples per second. Notice in particular how the resampling time increases dramatically when a rotated view is chosen. All timings are in seconds for the engine block dataset ($128^3$ voxels, aspect ratio of 1/2/2) on an 8MHz CM2 with 16k processors. The stepsize along the ray was $\Delta S = 1$ for all timings.*

to the exact iso surfaces in the human spine data set, gradient magnitude surface sources allow for the precomputation of the shading for the entire volume. This allows us to move the shading out of the inner loop and interpolate only the shaded values themselves. This is in some sense akin to Gouraud shading where the shading function is only evaluated at the vertices. This approximation (Figure 12) introduces visible artifacts for high supersampling rates. Again this is not a problem in practice since during the interaction phase the images are typically much smaller and the breakup of detail does not occur. For "publication" quality images it remains desirable to switch back to shading on the interpolated values.

### 3.3 Discussion

During the interactive phase in which the user chooses rendering parameters to highlight the information of interest fast response is important. By not performing any supersampling on the data, rates of between 1 and 6 frames per second can be achieved on a 16k processor CM2 [Schröder and

Stoll, 1992]. Even at one frame per second iterative choosing of rendering parameters is greatly facilitated. Many of the acceleration techniques such as pre-shading and/or performing no interpolation at all (instead simply choosing the value at the nearest grid point) yield images without visible artifacts at low resolutions, but break up when the supersampling rate is increased. High resolution images are typically only rendered once the basic set of rendering parameters has been determined. At that point it is no hindrance to effective work to wait ten or even thirty seconds for the final image.

One interesting observation is that even for fairly involved rendering options such as exact iso surfaces with specular highlights the rendering time is not significantly worse than for lookup table based rendering modes. This suggests that the overhead of traversing the data and interpolating it still dominates the actual floating point computations by a large factor. Given that massively parallel platforms such as the CM2 have considerable floating point computation capacities there is little reason not to use a high fidelity shading model. This is in contrast to workstations with graphics accelerators which can typically perform comparatively fast renderings when only using lookup table based shading and simple compositing in hardware. For complicated shading models their floating point capacity, however, is inadequate.

# 4 Conclusion

We have reviewed two parallel volume rendering algorithms. An image and an object space method. Both yield high level language implementations on the CM2 supercomputer with interactive performance rates. The codes scale in both the number of processors and data set size, albeit at different asymptotic rates. We have discussed the circumstances under which one of the algorithms may be preferable over the other as a function of usage context, number of available processors, memory, and data set sizes.

Taking the ray parallel algorithm as an example we have examined the time spent in rendering as a function of various shortcuts. For the interactive phase of choosing the right rendering parameters many of the shortcuts do not impact image quality enough to be undesirable. However, when increasing supersampling rates artifacts are noticable and final renderings are typically made without these shortcuts.

Much work remains to be done particularly in the area of non-regularly meshed data sets. It is not clear whether algorithms for such data sets can be mapped in an efficient way onto the rigid confines of a SIMD architecture. We are hopeful, however, that efficient mappings can be found for the current generation of MIMD computers such as the CM5. A solution to this problem becomes more important as the size of computational grids grows and the connectivity allowed by them becomes less and less regular. Examples in this category include mega finite element models and global weather codes.

## Acknowledgements

The first author would like to thank Lewis Tucker of Thinking Machines Corporation for providing access to tens of thousands of processors. The human spine CT data set was provided by Dr. A. Kern, Radiological Institute of FU Berlin.

## References

[Blelloch, 1990] Blelloch, G. *Vector Models for Data Parallel Computing.* Artificial Intelligence Series. MIT Press, Cambridge, MA, 1990.

[Braccini and Marino, 1980] Braccini, C. and Marino, G. Fast Geometrical Manipulations of Digital Images. *Computer Graphics and Image Processing*, 13:127–141, 1980.

[Cameron and Undrill, 1992] Cameron, G. G. and Undrill, P. E. Rendering Volumetric Medical Image Data on a SIMD-Architecture Computer. In *Proceedings of Third Eurographics Workshop on Rendering*, May 1992.

[Drebin *et al.*, 1988] Drebin, R. A., Carpenter, L., and Hanrahan, P. Volume Rendering. *Computer Graphics*, 22(4):65–74, August 1988.

[Hanrahan, 1990] Hanrahan, P. Three-Pass Affine Transforms for Volume Rendering. *Computer Graphics*, 24(5):71–78, November 1990.

[Hillis, 1985] Hillis, D. W. *The Connection Machine.* MIT Press, 1985.

[Krüger, 1991] Krüger, W. The Application of Transport Theory to the Visualization of 3-D Scalar Fields. *Computers in Physics*, pages 397–406, July 1991.

[Laur and Hanrahan, 1991] Laur, D. and Hanrahan, P. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Computer Graphics*, 25(4):285–288, July 1991.

[Levoy, 1988] Levoy, M. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Levoy, 1990] Levoy, M. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[Max *et al.*, 1990] Max, N., Hanrahan, P., and Crawfis, R. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics*, 24(5):27–33, November 1990.

[Neumann, 1992] Neumann, U. Interactive Volume Rendering on a Multicomputer. In *Proceedings of 1992 Symposium on Interactive 3D Graphics*, pages 87–93. ACM, ACM Siggraph, March 1992.

[Paeth, 1986] Paeth, A. W. A Fast Algorithm for General Raster Rotation. *Proceedings Graphics Interface*, pages 77–81, May 1986.

[Schröder and Salem, 1991] Schröder, P. and Salem, J. B. Fast Rotation of Volume Data on Data Parallel Architectures. In Nielson, G. M. and Rosenblum, L., editors, *Proceedings Visualization 1991*, pages 50–57. IEEE, IEEE Computer Science Press, October 1991.

[Schröder and Stoll, 1992] Schröder, P. and Stoll, G. W. Data Parallel Volume Rendering as Line Drawing. In *Proceedings of Workshop on Volume Visualization*, pages 25–32. ACM Siggraph, October 1992.

[Shirley and Tuchman, 1990] Shirley, P. and Tuchman, A. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics*, 24(5):63–70, November 1990.

[Smith, 1987] Smith, A. R. Planar 2-Pass Texture Mapping and Warping. *Computer Graphics*, 21(4):263–272, July 1987.

[Tanaka *et al.*, 1986] Tanaka, A., Kameyama, M., Kazama, S., and Watanabe, O. A Rotation Method for Raster Image Using Skew Transformations. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 272–277, June 1986.

[Thi, 1987] Connection Machine Model CM-2 Technical Summary. Technical Report HA 87-4, Thinking Machines Corporation, April 1987.

[Westover, 1989] Westover, L. Interactive Volume Rendering. In *CH Volume Visualization Workshop*, pages 9–16. ACM Siggraph, ACM Press, May 1989.

[Westover, 1990] Westover, L. Footprint Evaluation for Volume Rendering. *Computer Graphics*, 24(4):367–376, August 1990.

[Wilhelms and Gelder, 1991] Wilhelms, J. and Gelder, A. V. A Coherent Projection Approach for Direct Volume Rendering. *Computer Graphics*, 25(4):275–284, July 1991.

[Yagel and Kaufman, 1991] Yagel, R. and Kaufman, A. Template-Based Volume Viewing. Technical Report TR.91.07.19, SUNY Stony Brook, 1991.

[Yagel and Kaufman, 1992] Yagel, R. and Kaufman, A. Template-Based Volume Viewing. *Eurographics*, 11(3):153–167, 1992. Proceedings of Eurographics 92.

[Yagel, 1991] Yagel, R. *Efficient Methods for Volumetric Graphics.* PhD thesis, SUNY Stony Brook, December 1991.

[Yoo *et al.*, 1991] Yoo, T. S., Neumann, U., Fuchs, H., Pizer, S. M., Cullip, T., Rhoades, J., and Whitaker, R. Achieving Direct Volume Visualization with Interactive Semantic Region Selection. In Nielson, G. M. and Rosenblum, L., editors, *Proceedings Visualization 1991*, pages 58–65. IEEE, IEEE Computer Science Press, October 1991.

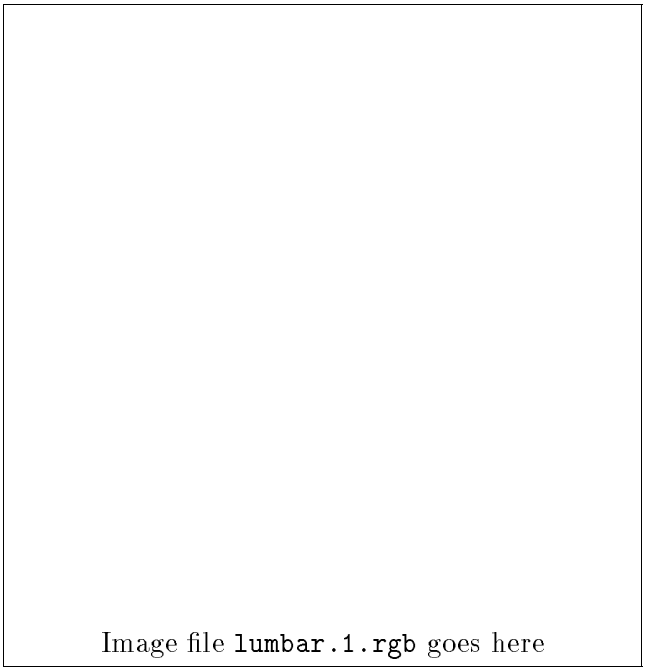


**Figure 4**: *CT data set rendered with* $\Delta S = 1/2$.

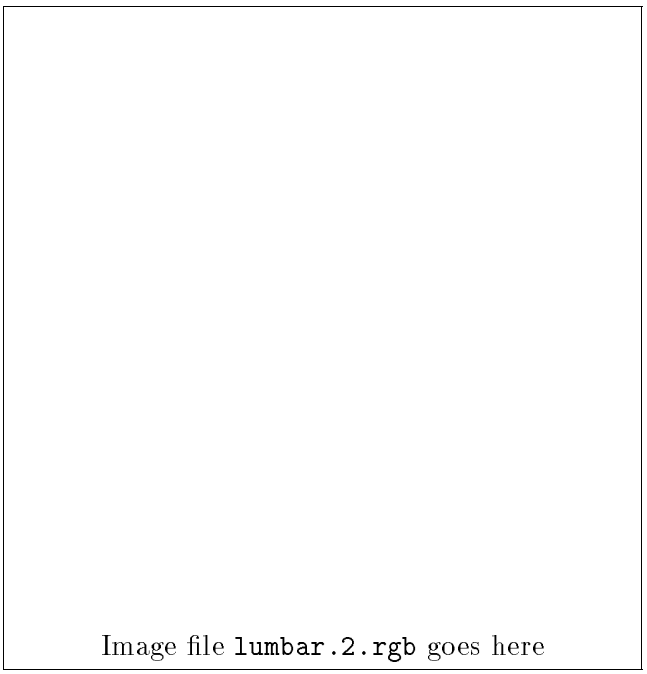


**Figure 5**: *CT data set rendered with* $\Delta S = 1$.

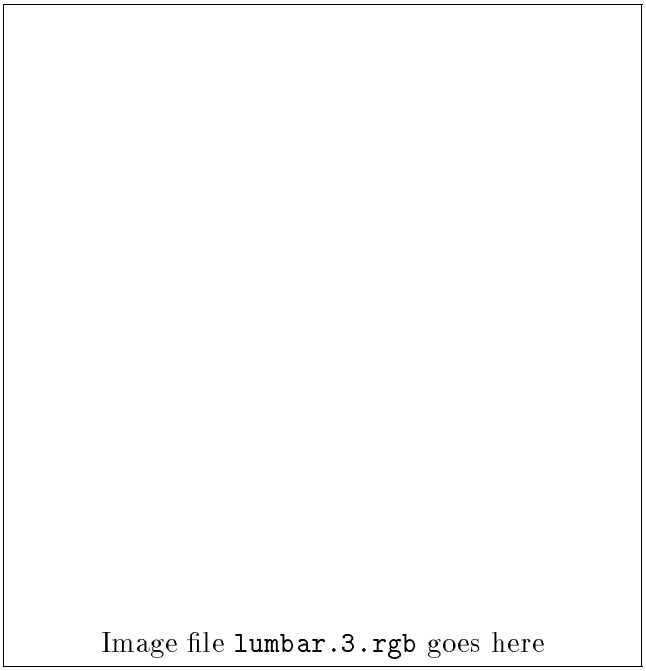


**Figure 6**: *CT data set rendered with* $\Delta S = 1/4$.

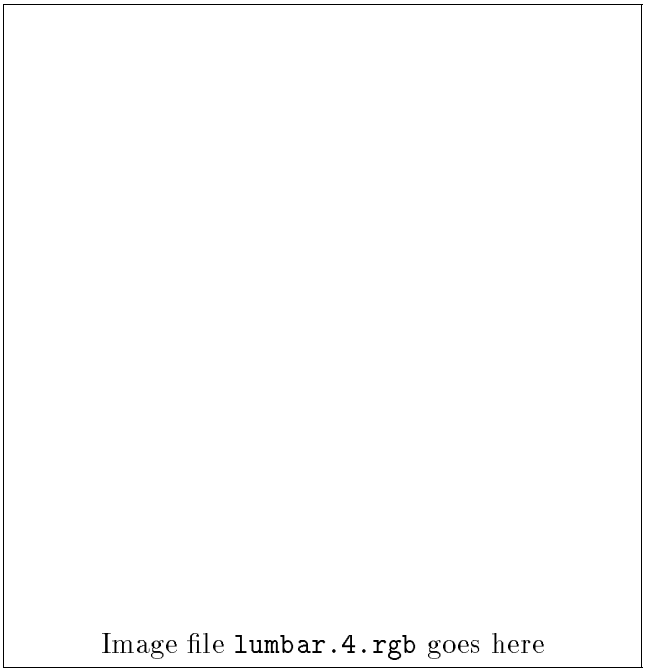


**Figure 7**: *CT data set rendered with $\Delta S = 1$ and no renormalization of gradients after interpolation.*

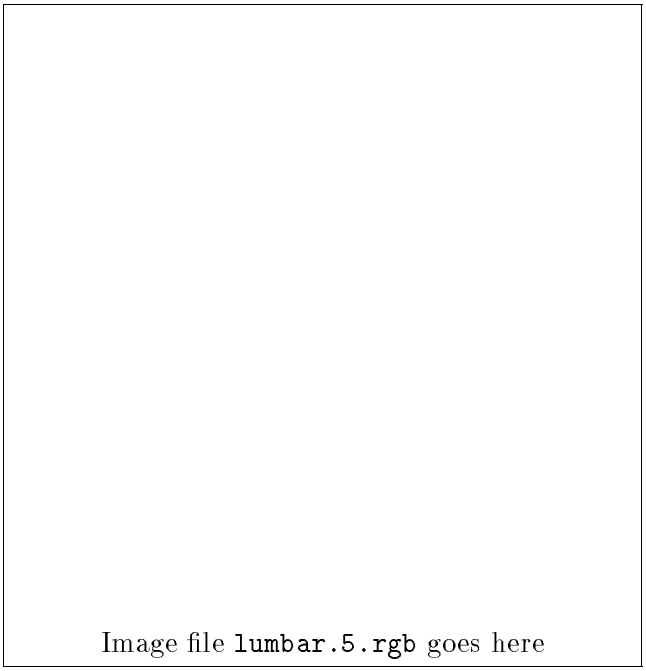


**Figure 8**: *CT data set rendered with* $\Delta S = 1/2$ *and no specular component.*

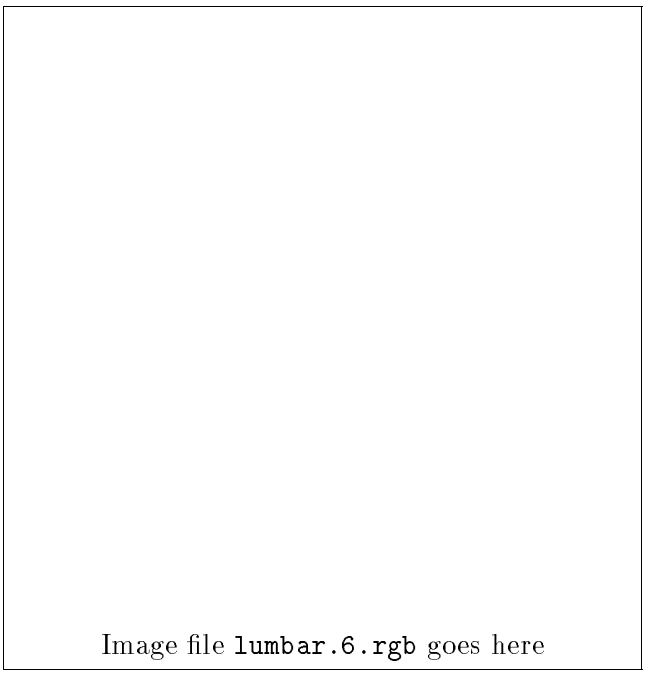


**Figure 9**: *CT data set rendered with* $\Delta S = 1$ *and no specular component.*

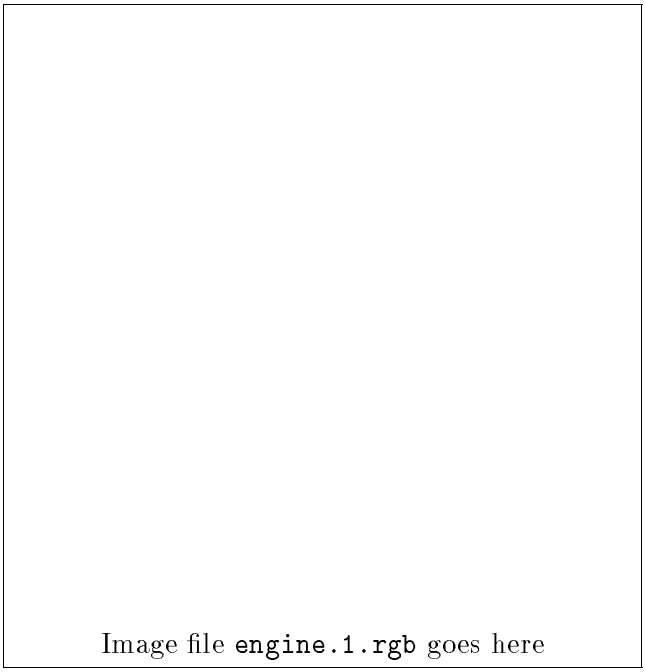


**Figure 10***: CT data set rendered with* $\Delta S = 1$.

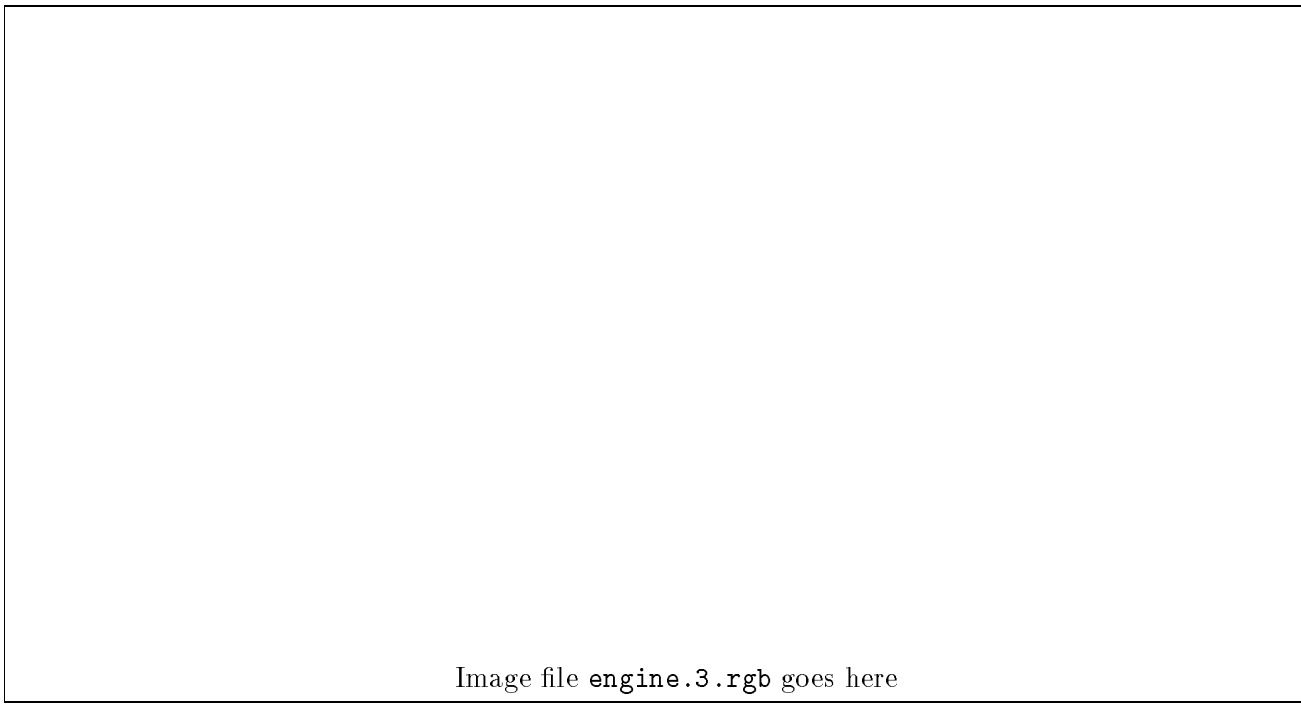


**Figure 11**: *CT data set rendered with* $\Delta S = 1$.

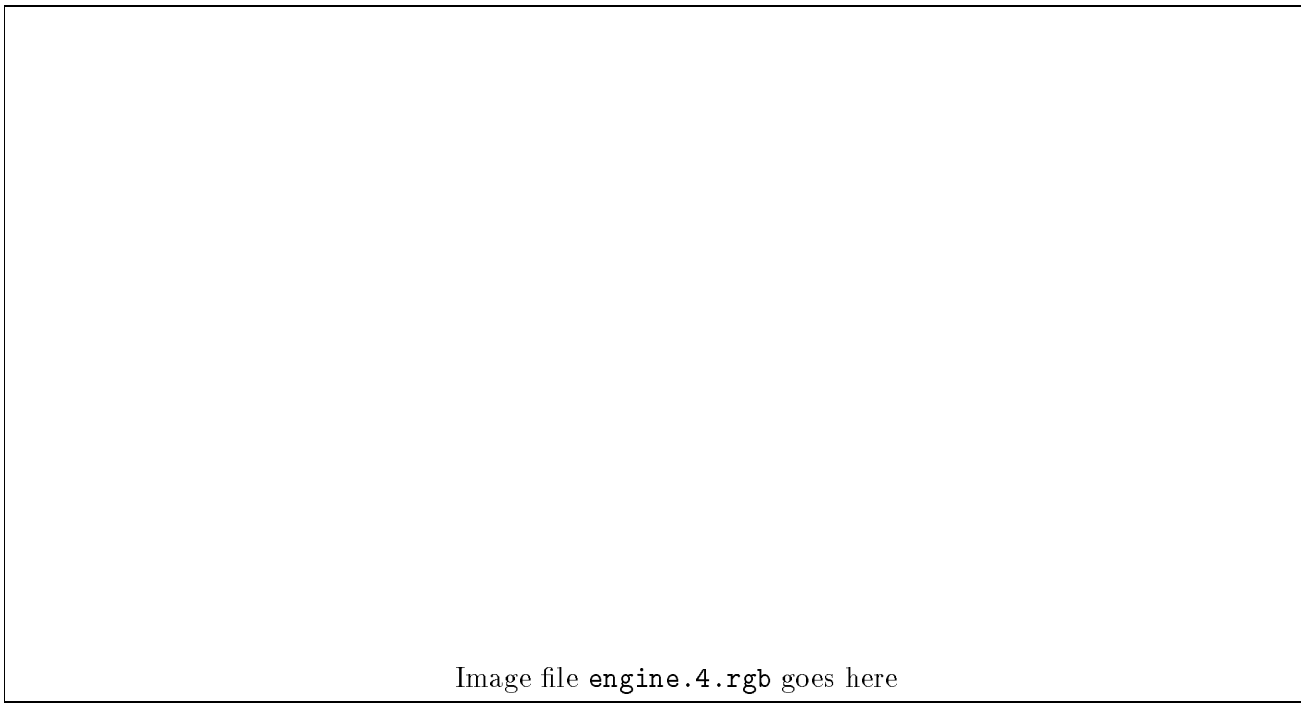


**Figure 12**: *CT data set rendered with* $\Delta S = 1$ *and preshading.*