Appeared in the Proceedings of the *Visualization'96* conference, pp. 65-72

Fast Perspective Volume Rendering with Splatting by Utilizing a Ray-Driven Approach

Klaus Mueller[1] and Roni Yagel[1,2]

[1] Biomedical Engineering Center, [2] Department of Computer and Information Science
The Ohio State University, Columbus, Ohio

ABSTRACT

Volume ray casting is based on sampling the data along sight rays. In this technique, reconstruction is achieved by a convolution, which collects the contribution of multiple voxels to one sample point. Splatting, on the other hand, is based on projecting data points onto the screen. Here, reconstruction is implemented by an "inverted convolution" where the contribution of one data element is distributed to many sample points (i.e., pixels). Splatting produces images of a quality comparable to raycasting but at greater speeds. This is achieved by precomputing the projection footprint that the interpolation kernel leaves on the image plane. However, while fast incremental schemes can be utilized for orthographic projection, perspective projection complicates the mapping of the footprints and is therefore rather slow. In this paper, we merge the technique of splatting with principles of raycasting to yield a ray-driven splatting approach. We imagine splats as being suspended in object space, a splat at every voxel. Rays are then spawned to traverse the space and intersect the splats. An efficient and accurate way of intersecting and addressing the splats is described. Not only is ray-driven splatting inherently insensitive to the complexity of the perspective viewing transform, it also offers acceleration methods such as early ray termination and bounding volumes, which are methods that traditional voxel driven splatting cannot benefit from. This results in competitive or superior performance for parallel projection, and superior performance for perspective projection.

1 INTRODUCTION

In the past few years, direct volume rendering has emerged as a major technology in many visualization scenarios. A natural choice for its use is the viewing of data sets that are inherently volumetric, such as medical data produced by MRI, CT, or PET scanners, or scientific studies, such as the visualization of flow or terrain. Newer applications, such as volume-based interactive design and sculpting [8][16], require direct volume rendering to display the results of the design process. Yet another application for direct volume rendering is the interactive exploration of volume data in form of fly-throughs, that has recently shown great promise for medical diagnosis and surgical training [7].

The rendering speed requirements for interactive volume viewing and manipulation stand in great contrast to the complexity of the rendering task. In contrast to surface rendering where hardware accelerators are readily available at relatively low cost, true volumetric rendering hardware is still rather expensive and restrictive. Thus the development of software solutions running on standard computer hardware still forms an important area of research. In addition to rapid generation, we also desire the image to look realistic. The perspective viewing distortion is an important component in this strive for realism, as it can be utilized to convey some depth information.

[1] 270 Bevis Hall, Columbus, OH 43210, klaus@chaos.bme.ohio-state.edu, http://chopin.bme.ohio-state.edu/~klaus/klaus.html.

[2] 789 Dreese Lab, Columbus, Ohio 43210, yagel@cis.ohio-state.edu, http://www.cis.ohio-state.edu/hypertext/volviz/main.html

1.1 Modes of Volume Rendering

In this paper, we distinguish between three types of volume rendering modes: 1) Direct volume rendering in the low-albedo approximation; 2) Summation or X-ray rendering; and 3) Maximum Intensity Projection (MIP) rendering.

In direct volume rendering, we compute $I_\lambda(\boldsymbol{x},\boldsymbol{r})$, the amount of light of wavelength λ coming from ray direction $\boldsymbol{r}$ that is received at point $\boldsymbol{x}$ on the image plane:

$$I_\lambda(\boldsymbol{x},\boldsymbol{r}) = \int_0^L \phi_\lambda(s)\exp\left(-\int_0^s \mu(t)dt\right)ds \qquad (1)$$

Here L is the length of ray $\boldsymbol{r}$. We can think of the volume as being composed of particles that receive light from all surrounding light-sources and reflect this light towards the observer according to the specular and diffuse material properties of the particles. Thus, in Equation (1), ϕ_λ is the light of wavelength λ reflected at location s in the direction of $\boldsymbol{r}$. Since volume particles have certain densities μ (e.g., opacities), the light scattered at s is attenuated by the volume particles between s and the eye according to the exponential attenuation function. The process of merging colors and opacities along the ray is called *compositing*.

In summation rendering, we are only interested in the attenuation line integral accumulated along $\boldsymbol{r}$:

$$I_{Sum}(\boldsymbol{x},\boldsymbol{r}) = \int_0^L \mu(t)dt \qquad (2)$$

Summation rendering comes to bear in a class of iterative 3D reconstruction algorithms, the Algebraic Reconstruction Technique (ART) [5], that is often used to reconstruct a volumetric object from projectional image data acquired by CT, PET, or SPECT scanners. One ART iteration step consists of projecting the volume onto one of the image planes, computing the error between the rendered image and the acquired image, and backprojecting the error image in a "smearing" fashion onto the voxel grid. This procedure is repeated possibly many times at all projection angles for which data is available and continues until some convergence criterion is reached. Both projection and backprojection is performed using a variation of summed perspective volume rendering. Since a large number of projection operations is usually required for faithful 3D reconstruction, the volume renderer employed must be both fast and accurate. If the projection data were generated by a cone-beam projection source, as is usually the case in 3D CT, then the volume renderer must support the perspective viewing transform.

Finally, in MIP we seek the maximum density along $\boldsymbol{r}$:

$$I_{MIP}(\boldsymbol{x},\boldsymbol{r}) = max(\mu(s)),(0 \le s \le L) \ . \qquad (3)$$

MIP displays are often utilized in medical applications, e.g., in the display of volumes obtained by Magnetic Resonance Angiography (MRA). In MRA, a vascular structure is perfused with an opaque contrast agent while the MRI image acquisition is in progress. If a MIP rendered image is appropriately thresholded, then the object appears transparent with only the brightest (e.g. densest) features,

here the opacified arteries, shining through. Perspective viewing may be used to provide the viewer with the necessary depth cues for distinguishing the individual arterial structures, and, at the same time, mimics best the cone-beam viewing scenario found in traditional X-ray angiography.

1.2 Ray-Driven vs. Voxel-Driven Approaches

Direct volume rendering algorithms can be grouped into two categories: Forward projection methods in which rays are cast from the image pixels into the volume [10][15][18], and backward projection methods, such as the splatting algorithm introduced by Westover [20], in which volume elements are mapped onto the screen [19][21].

In raycasting, a discretized form of Equation (1) is computed by sampling the volume at certain intervals and compositing the sample's opacity and color with the present ray values. A sample value is obtained by reconstructing, e.g. interpolating, the continuous volume function at the sample location from the voxel grid values. This is a rather time consuming process given the large number of sampling steps and makes the use of sophisticated interpolation filters with an extent larger than the commonly applied trilinear kernel impractical. However, raycasting can be accelerated by restricting the interpolation operations to relevant volume regions. Common acceleration techniques include bounding polyhedra [17], space leaping [22][3], and hierarchical volume decomposition, such as octrees or pyramids [4][11]. In addition, since raycasting usually operates from front to back, compositing can terminate as soon as the accumulated opacity reaches unity. This is not valid for X-ray and MIP renderings which may use bounding polyhedra, but must then traverse the entire enclosed volume.

For our purposes it is important to realize that raycasting is relatively insensitive to the complexity of the perspective viewing transform: Perspective sight-rays emanating from neighboring pixels are just tilted relative to each other but remain linear and, therefore, do not differ conceptually from rays stemming from parallel projection. However, we must also realize that the diverging nature of the perspective rays leads to a non-uniform sampling of the volume: Volume regions more distant from the eye point are sampled less densely then regions closer to the eye. Thus small features further back in the volume may be missed and not displayed.

In splatting, a voxel's contribution is mapped directly onto the pixels, eliminating the need for interpolation operations. A voxel is modeled as a fuzzy ball of energy shaped by the interpolation function. Projection is performed by mapping the view-transformed, pre-projected 2D "footprint" of the kernel function to the screen. This footprint is usually implemented as a discrete lookup table, storing an array of equally spaced parallel line integrals that are computed across the kernel function either analytically or by quadrature. For non-compositing X-ray type projections, summing the individual voxel contributions by their footprint line integrals on the image plane is equivalent to accumulating this sum by interpolating samples from the grid voxels along individual rays. However, the line integrals across a voxel are now continuous or approximated with good quadrature, whereby in raycasting the computed line integrals are only discrete ray sums. Splatting also allows the efficient use of sophisticated interpolation functions of larger extent. For these two reasons, summed, non-compositing volume integrals as used for X-ray projection are considerably more accurate with splatting than with raycasting. However, compositing of color and opacity is only approximate. This stems from the circumstance that kernels must overlap in object space to ensure a smooth image, and thus reconstruction and compositing can no longer be separated. Fortunately, for volumes with reasonably dense sampling rates, the effects of this deficiency are usually hardly noticeable.

Traditional splatting allows orthographic projections to be rendered very fast by utilizing efficient incremental schemes. Unfortunately, these schemes are not applicable for perspective projection due to the non-linearity of the perspective viewing transform. Another drawback of splatting is that being a voxel-driven algorithm, it must visit every voxel in the grid. However, only voxels with density values within the range of interest need to be transformed and composited. Furthermore, since splatting is most commonly used in a back-to-front fashion, it is likely that many voxels will be transformed and shaded only to be occluded later by opaque structures located closer to the image plane, giving rise to many unnecessary computations. A possible front-to-back approach, eliminating this problem, would divide the volume into octrees and the image into quadtrees, transforming only the octree nodes that fall within a non-opaque region in the screen quadtree. However, the induced overhead for tree traversal may offset the savings gained. Another way of speeding up the splatting process is the use of larger splats in volume areas of little variation, as suggested by [9].

Approaches that utilize special graphics hardware have also been proposed. These solutions are relatively insensitive to the complexity of the perspective viewing transform, but require sophisticated graphics workstations. While [1] maps the voxel footprint as a texture onto a polygon and uses texture mapping hardware for its projection, [4] approximates the screen footprint as the projection of a Gouraud shaded polygon mesh. The quality of the rendered image or 3D reconstruction, respectively, is then bounded by the resolution of the texture memory (commonly not more than 12 bits per texel component) or the quality of the polygonal approximation, respectively.

We now describe an algorithm that merges voxel-driven splatting with the advantages of ray-driven approaches to enable fast and accurate perspective rendering. Although researchers in the field of medical imaging have proposed table-based pre-integrated raycasting long before the splatting algorithm became popular [6] (and later [12]), these efforts were limited to summed X-Ray type rendering, and the issue of comparing ray-driven and voxel-driven approaches in terms of speed and accuracy was never addressed. We also introduce the concept of pyramidal beams to compensate for the effect of diverging perspective rays.

In the following, Section 2 describes voxel-driven splatting, Section 3 describes ray-driven splatting, and Section 4 provides timings and images for both approaches and traditional raycasting.

2 VOXEL-DRIVEN SPLATTING

We now give a brief review of incremental schemes used for traditional voxel-driven splatting. In the following discussion we restrict ourselves to volumes that are sampled in a regular cubic grid. This may seem restrictive at first, but it is not unrealistic to assume that volumes sampled on general rectilinear grids (and some irregular grids) can always be resampled into such a grid configuration. The ellipsoidal kernel functions used by Westover then become simple spherical kernel functions which project identically for all viewing directions.

2.1 Orthographic Projection

For spherically symmetric kernel functions the same footprint table and mapping function can be used for all viewing angles. We chose a Gaussian kernel function with relevant radial extent of r=2 voxel lengths, sampled into a 2D footprint table of 1000×1000 entries. Nearest neighbor interpolation is used when mapping the footprint onto the image.

Consider Figure 1 where the orthographic projection of a voxel's footprint is depicted for the 2D case. As suggested by [1], we can think of the footprint as a polygon with a superimposed texture map that is placed in object space. Hereby the texture map is

given by the projected kernel function, e.g. the array of line integrals. For the remainder of our discussion we will refer to the footprint in object space as the *footprint polygon*, while the projection of the footprint polygon onto the image plane will be called the *footprint image*. Recall that splatting accumulates (with some limitations) the same value on the image plane as a ray would accumulate when traversing the volume. Thus, when projecting the footprint polygon to obtain the line integral for the pixel in the footprint image we must ensure that we position the footprint polygon orthogonal to the direction of the sight-ray in object space.

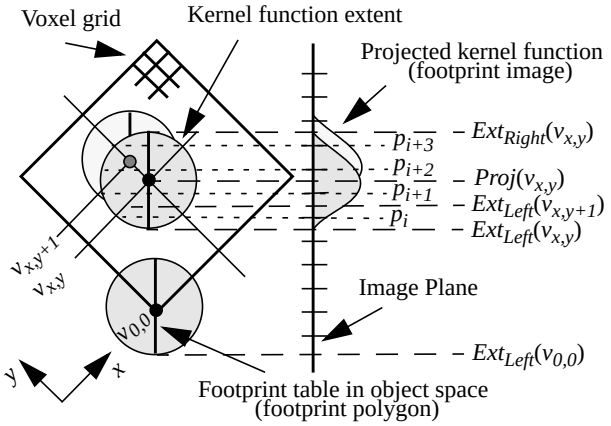


Figure 1: Orthographic splatting: Incremental mapping of the footprint polygons onto the image plane and incremental mapping of the image pixels within the footprint image into the footprint table.

In orthographic viewing the projection and mapping operations are simple and can be done in an incremental fashion. Figure 1 shows this graphically for the 2D case. The volume is decomposed into slices, where the slice planes are the voxel planes most parallel to the image plane. To set up the incremental scheme, we have to compute the transformation matrix of the volume. Then, we project the footprint polygon of a voxel located at one of the volume vertices, say voxel $v_{0,0,0}$, onto the image plane. This yields a point $Ext_{LeftBot}(v_{0,0,0})$, the bottom left vertex of $v_{0,0,0}$'s footprint image on the projection plane ($Ext_{Left}(v_{0,0})$ in Figure 1). For all other voxels $v_{x,y,z}$, the bottom left projection extrema $Ext_{LeftBot}(v_{x,y,z})$ can be computed incrementally by adding appropriate elements of the transformation matrix and advancing in a voxel→row→slice fashion [13], whereby slices are processed from back to front. Once a point $Ext_{LeftBot}(v_{x,y,z})$ is known, then all other footprint image vertex points are easily computed by adding the kernel extent along the two image dimensions. In the 2D case of Figure 1 we get:

$$Ext_{Right}(v_{x,y}) = Ext_{Left}(x, y) + 2 \cdot extent_{kernel} \cdot$$

This yields a square on the image plane. Pixels that fall within the square are then determined by rounding operations. (In Figure 1, the interval of pixels $[p_i,p_{i+3}]$ that fall within the footprint image is [Ceil($Ext_{Left}(v_{x,y})$), Floor($Ext_{Right}(v_{x,y})$). Simple incremental additions are used to scan through the square, adding the voxel's opacity and color to the slice compositing sheet.

Thus, the splatting algorithm consists of two nested loops: the outer loop maps footprint polygons onto the image plane, while the inner loop maps the pixels within the footprint image into the footprint table. If we define N to be the number of voxels in the grid, and n the number of pixels that fall within a footprint image, then the computational complexity for the mapping operation footprint polygon to image plane is O(N) and the mapping operation footprint image to footprint table is O(n) for each voxel. Altogether there are about 10 additions and multiplication in the outer loop and 7 such operations in the inner loop.

2.2 Perspective Projection

In perspective, incremental arithmetic can still be utilized to map the pixels onto the footprint table (the inner loop of the splatting procedure), although we now require two divisions per pixel and a few more additions than in the orthographic case. However, for mapping the footprint polygon onto the image plane we no longer can employ incremental operations which is due to the non-linearity of the perspective transform.

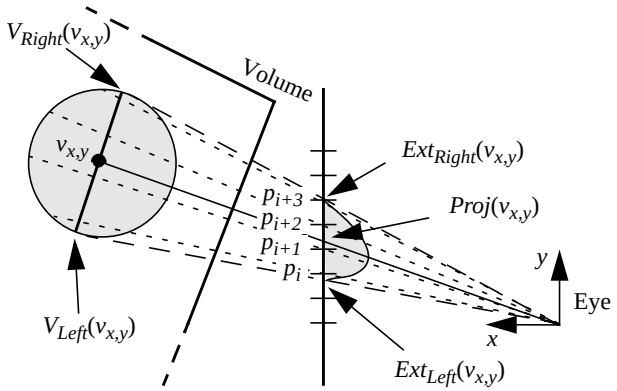


Figure 2: Perspective splatting: Mapping the footprint polygon onto the image plane and mapping the affected image pixels back onto the footprint table.

Consider Figure 2 which illustrates perspective splatting in 2D. Here we fixed the coordinate system at the eye point. The footprint polygon is placed orthogonal to the vector starting at the eye and going through the center of $v_{x,y,z}$. Note that this yields an accurate line integral only for the center ray, all other rays traverse the voxel kernel function at a slightly different orientation than given by the placement of the 2D (1D in Figure 2) footprint polygon in object space. This is illustrated in Figure 3 for the 2D case. Here we show the footprint polygon f_p (thick solid line) positioned orthogonal to the line connecting the eye point and the voxel center. The dotted line *r'* coincides with the ray that intersects pixel p_j on the image plane. Clearly, *r'* does not pierce the footprint polygon at a right angle. Since all entries in the footprint table are computed for rays intersecting the footprint polygon perpendicularly, the ray integral retrieved by the mapping operation is due to a ray along the solid line *r*. To obtain the correct value for *r'*, we should rotate the footprint polygon to position f_p' (thick dotted line) before we perform the mapping operation for p_j. This, however, is

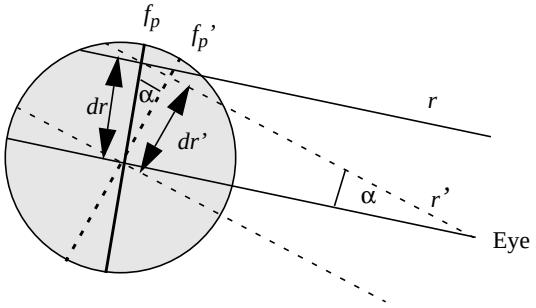


Figure 3: Perspective splatting: Accumulated line integrals vs. actual line integrals.

prohibitive as it would make the mapping operations computationally rather costly. Fortunately, the error is small for most cases. Since the splatting kernel is rotationally symmetric, the error solely due to the orientation difference of *r* and *r'* is zero. However, there remains an error with regards to the position at which the lookup table is indexed. This error is given by: $dr' - dr = dr(1 - \cos\alpha)$, which is nearly zero for most cases unless voxels close to the image plane are viewed at large view cone angles.

The coefficients of the footprint polygon's plane equation are given by the normalized center ray (the vector *eye*-$v_{x,y,z}$). From this equation we compute two orthogonal vectors *u* and *w* on the plane. Hereby *u* and *w* are chosen such that they project onto the two major axes of the image. Using *u* and *w*, we can compute the spatial *x,y,z* positions of the four footprint polygon vertices in object space ($V_{Right}(v_{x,y})$ and $V_{Left}(v_{x,y})$ in the 2D case depicted in Figure 2). These four vertices are perspectively projected onto the image plane. This yields the rectangular extent of the footprint image, aligned with the image axes ($Ext_{Right}(v_{x,y})$ and $Ext_{Left}(v_{x,y})$ in the 2D case). By expressing the intersections of the pixel rays with the footprint polygon in a parametric fashion we can set up an incremental scheme to relate the image pixels within the footprint image with the texture map entries of the footprint table. Hence the inner loop of the splatting procedure can still be executed efficiently.

The computational effort to map a footprint polygon onto the screen and to set up the incremental mapping of the pixels into the footprint table is quite large: Almost 100 multiplications, additions, and divisions, and two square root operations are necessary. This cost is amplified by the fact that this has to be done at O(*N*), unless voxels can be culled by thresholding. The inner loop requires 10 additions, multiplications, and divisions, and is therefore still reasonably fast. We found that with X-ray type summation rendering, perspective projection was about four times more expensive than orthographic projection.

3 RAY-DRIVEN SPLATTING

As indicated before, ray-driven algorithms are generally not sensitive to the non-linearity of the perspective viewing transform. We now describe an approach that uses this paradigm for splatting.

3.1 Orthographic Projection

In ray-driven splatting, voxel contributions no longer accumulate on the image plane for all pixels simultaneously. In contrast, each pixel accumulates its color, opacity, and density sums separately. As in voxel-driven splatting, we ensure proper compositing by dividing the volume into 2D slices formed by the planes most parallel to the image plane. When a sight-ray is shot into the 3D field of interpolation kernels, it stops at each slice and determines the range of voxel kernels within the slice that are traversed by the ray. This is shown in Figure 4 for the 2D case: The ray originating at pixel p_i pierces the volume slice located at x_s at $y=y(i,x_s)$. The voxel kernels within the slice x_s that are intersected by the ray are given by the interval $[\text{Ceil}(y_{Left}(i,x_s)), \text{Floor}(y_{Right}(i,x_s))]$. We compute $y_{Right}(i,x_s)$ as:

$$y_{Right}(i, x_s) = y(i, x_s) + \frac{extent_{kernel}}{\cos(\alpha)} .$$

The computation for $y_{Left}(i,x_s)$ is analogous. After determining the active voxel interval we must compute the indexes into the voxel footprint table. This can be efficiently implemented by realizing that the index into the footprint table of a grid voxel *v* located at coordinates (y_v,x_v) is given by the distance *dr* of the two parallel lines (planes in 3D) that traverse *v*'s centerpoint and the slice intersection point of the ray at $y(i,x_s)$, respectively (see Figure 5). We find:

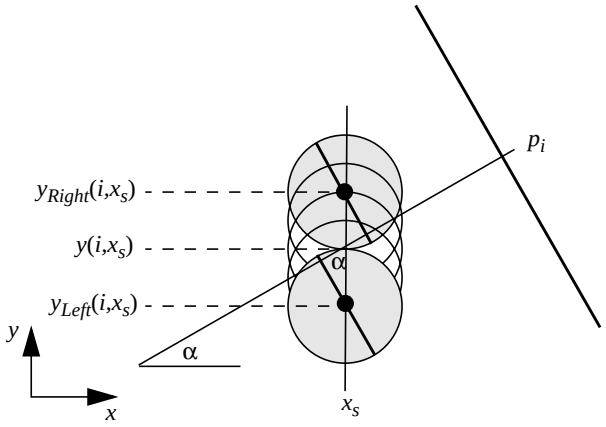


Figure 4: Ray-driven splatting: Determining the range of voxels within a given compositing plane that are traversed by a ray originating at pixel p_i.

$$\begin{aligned} dr &= a \cdot x_s + b \cdot y(i, x_s) - a \cdot x_s - b \cdot y_v \\ &= (b \cdot (y(i, x_s) - y_v)) \end{aligned} \quad (4)$$

where *a* and *b* are the coefficients of the implicit line equation $a \cdot x_s + b \cdot y(i, x_s) = 0$ and are also given by the components of the (normalized) ray vector. Maintaining the variables $y_{Left}(i,x)$, $y_{Right}(i,x)$, and *dr* along a ray can all be done using incremental additions.

For the 3D case, we need to replace the linear ray by two planes. A 3D ray is defined by the intersection of two orthogonal planes cutting through the voxel field. The normal for one plane is computed as the cross product of the ray and one of the minor axes of the volume. The normal of the second plane is computed as the cross product of the ray and the normal of the first plane. Thus, the two planes are orthogonal to each other and are also orthogonal to the voxel footprint polygons. Intersecting the horizontal plane with a footprint polygon and using plane equations in the spirit of Equation (4) results in the horizontal row index dr_{row} into the footprint

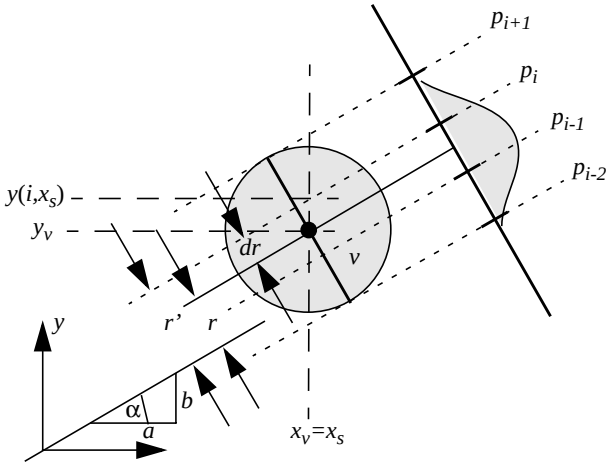


Figure 5: Ray-driven splatting: Computing the index *dr* into the footprint table.

table, while the intersection with the vertical plane yields the vertical column index dr_{col}. Using these two indexes, the value of the ray integral can be retrieved from the footprint table.

Note that, in contrast to voxel-driven splatting, a voxel may now be accessed and composited more than once for different rays. Since we do not want to shade the voxel each time it is accessed, we either pre-shade the volume before rendering it for many views, or we shade on the fly and cache away the result. Since we are using phong shading in our renderings, we shade voxels on the fly and maintain an additional shaded volume where we store the color of a shaded voxel once it is available. In systems where memory is scarce, one could also use a hash table.

There are now three nested loops: The most outer loop sets up a new ray to pierce the volume, the next inner loop advances the ray across the volume slice by slice and determines the set of voxels traversed per slice, and finally, the most inner loop retrieves the voxel contributions from the footprint tables. For orthographic projection, the plane equations have to computed only once since all rays are parallel. This reduces setting up a new ray to a few simple incremental operations. The cost of advancing a ray across the volume and determining the footprint entries is comparable to the cost of rotating a kernel and splatting it onto the image plane in the voxel-driven approach. We found the time for parallel projected summed X-ray rendering to be only slightly higher for ray-driven splatting than for voxel-driven splatting, however, when compared to traditional raycasting the saving were at the order of 3 with the added benefit of better accuracy. For direct volume rendering, the benefits of early ray termination in the ray-driven approach dominate over the extra cost for setting up the ray when an adequate bounding volume is used.

The ray-driven approach changes the splatting algorithm from voxel order to pixel order. Thus, the most outer loop is of O(P), with P being the number of image pixels. This has the advantage that the complexity of any extra work that has to be done for perspective projection (e.g. recomputing the two planes that define the ray in 3D) is roughly one order of magnitude less than in voxel-driven splatting.

3.2 Perspective Projection

The only change required for perspective, as hinted before, is to move the computation of the two ray defining planes into the outer most loop since each ray has a different orientation in space. This amounts to about 50 extra additions, multiplications, and divisions, and three square roots per pixel. As was also mentioned before, however, the complexity of this step is only O(P), thus the extra work required for perspective has a much smaller scaling factor than in voxel-driven splatting. And indeed, we found that the extra computational cost for perspective in ray-driven splatting is rather small. Note also that ray-driven splatting does not introduce inaccuracies. As a matter of fact, it prevents the situation illustrated in Figure 3 by design.

But there persists one problem with perspective viewing in general: Due to the diverging rays, small features further away from the eye could be missed. In Figure 6a we see that traditional raycasting is especially prone to exhibit this behavior as it must use interpolation kernels of small reach and, in addition, only point-samples the voxel space. Figure 6b illustrates that ray-driven splatting has greater potential in detecting small objects since we may use wider interpolation kernels in an efficient manner and ray-sampling is, for all practical purposes, continuous. However, even with larger interpolation kernels, we cannot fully eliminate the well-known aliasing effects of the progressively coarser sampling rate as rays diverge: Objects may still either be missed (such as the second object in Figure 6b) or show up very faintly. Both of these effects result in object flicker in animated viewing or incorrect X-ray projection images, respectively.

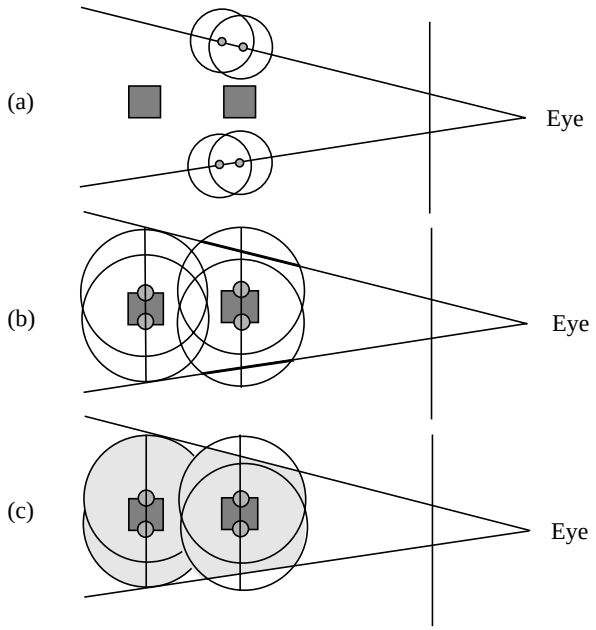


Figure 6: The effect of diverging rays in perspective: (a) In raycasting, both objects are missed due to the small extent of the interpolation kernel. (b) In ray-driven splatting, due to the wider interpolation kernel, the object closer to the eye now contributes, although to a small extent, to the image. (c) By using summed area footprint tables and fan ray beams (pyramidal beams in 3D), both objects fully contribute to the image.

Non-homogeneous perspective sampling may also have degenerate effects in the 3D reconstruction from cone-beam projection images, especially for reconstructions at low object contrast. Since the value of a pixel in an acquired X-ray projection image is equivalent to the attenuation of a pyramid shaped X-ray beam traversing the volume, this beam needs to be approximated as close as possible when computing the summed projection images in the reconstruction process.

The problem's remedy is to maintain a constant sampling rate everywhere in voxel space. Solutions proposed for raycasting include the spawning off of additional rays as one penetrates further into the volume [14]. Splatting, however, offers a much more attractive solution. Similar to anti-aliasing methods proposed in texture mapping, one can create summed area footprint tables [2] and use these in the splatting procedure. This requires tracing the pyramidal volume extruded from the rectangular pixel into the volume and intersecting it with the spherical voxel basis functions. Figure 6c illustrates for the 2D case that in this way all objects can be captured and contribute in proper amounts to the image. Note, that summed area tables imply a 0th degree lowpass filtering of the image.

Let us now explain this approach in more detail: A pixel viewing pyramid is bounded by four planes. Each plane is bounded by one of the pixel edges on the image plane, and the two viewing rays emanating from the pixel vertices on either side of the pixel edge. To obtain the plane equation we simply take the cross product of one of the pixel vertex rays and the pixel edge vector. Figure 7a illustrates this process.

Just like in the line integral case, intersecting a plane pair with the footprint polygon in space yields an index point into the footprint table. By intersecting each of the four plane pairs with the footprint table we obtain four table indexes. The indexed table val-

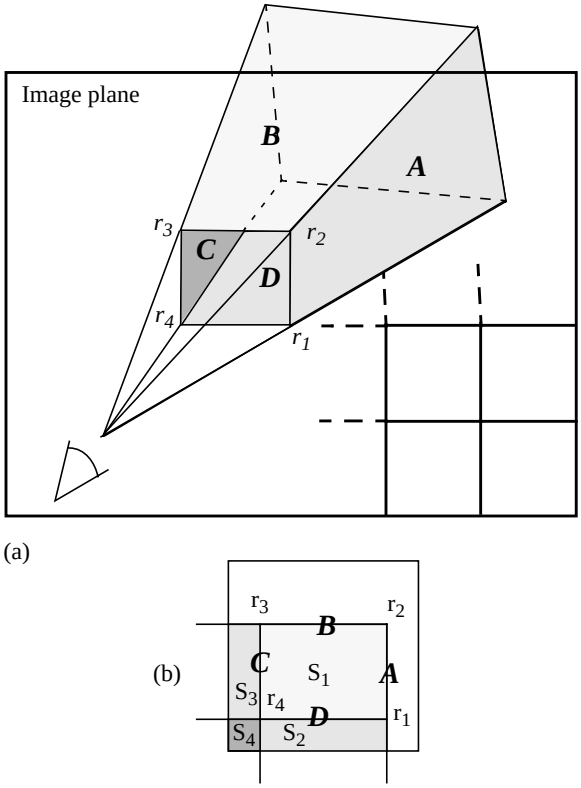


Figure 7: Ray-driven splatting with a summed area footprint table. (a) Pixel viewing pyramid: rays r_1,...,r_4 are emanating from the pixel vertices, together with the pixel edges they form four planes, A,..,D (b) Planes A,..,D intersect the summed area footprint table segmenting it into four areas S_1,..,S_4, $S_4 \subset S_2$,$S_3 \subset S_1$.

ues are then combined according to the well-known summed area table (SAT) formula such that (see Figure 7b):

$$weight = \frac{SAT[S_1] - SAT[S_2] - SAT[S_3] + SAT[S_4]}{Area(S_1) - Area(S_2) - Area(S_3) + Area(S_4)}$$

It must be added, however, that the method bears slight inaccuracies. This is due to the fact that for most pyramid orientations one or more of the plane pairs have non-orthogonal normals, and hence these plane pairs fail to intersect the footprint table polygon as two perpendicular lines. This violates the summed area table index condition. However, the deviation from 90˚ is usually relatively small since we always flip the viewing direction as soon as the angle between the direction of the ray pyramid center ray and the major viewing axis exceeds 45˚.

Note however, that errors also occur with voxel-driven splatting when summed area footprint tables are used. These errors are additional to the inaccuracies reported in Section 3.2. In voxel-driven splatting, a pixel maps onto the (summed) footprint table as a non-rectangular four sided polygon which must be approximated by a rectangle. Thus, similar errors to those found in the ray-driven case occur.

4 RESULTS

The colorplates show a collection of images obtained with ray-driven splatting and raycasting. Since ray-driven splatting is equivalent to voxel-driven splatting, we only present images obtained with the new method. The Brain dataset is the UNC "MRbrain", interpolated into a cubic grid. The resulting volume has a size of 256×256×145 voxels. The Scalp data set was obtained from an MRA study and has a size 256×256×247 voxels. As was mentioned before, MRA studies oftentimes use MIP rendering to display the contrast enhanced arterial structures.

Color plate a) through c) show 260×260 pixel images of the Brain data set rendered as an isosurface with parallel projection, a 30˚ perspective viewing angle, and a 60˚ perspective viewing angle, respectively. We find that the 30˚ perspective rendering gives the head a more realistic look. For example, the curvature of the skull's edge above the right eye looks much more natural in 30˚ perspective than under parallel projection. The 60˚ peephole perspective may be too exaggerated for clinical use.

While a raycaster using trilinear interpolation and a stepsize of Δs=1.0 voxel lengths produces images of slightly inferior quality (at similar computational expense), it produces images of higher quality when sampling at Δs=0.3 (however, at much higher computational expense, as shown below). The qualitative difference between raycasting and ray-driven splatting is best illustrated at lower image resolution, a possible scenario when pre-viewing volumes or when working in an interactive environment. Color plate d) through f) show three 30˚ perspective renderings of the Brain dataset. Here the image resolution was reduced to 130×130 pixels. The image sequence illustrates that ray-driven splatting performs well, even at half the volume resolution, and that a raycaster must sample at around Δs=0.3 for similar results. This, however, comes at higher computational expense (see below).

Color plate g) and j) show the Scalp dataset rendered as an isosurface at orthographic projection and 30˚ perspective projection, respectively. (The Scalp images are of size 280×280). In color plate h), we see the same dataset rendered as a parallel X-ray projection, while color plate k) shows a cone-beam X-ray projection of the dataset. As was mentioned before, these kind of images are typically produced as intermediate results in the ART reconstruction algorithm for parallel-beam (0˚ orthographic projection) and cone-beam (30˚ to 60˚ perspective projections).

Color plate i) displays an image obtained by parallel MIP projection, appropriately thresholded to reveal only the brightest (i.e. densest) structures. Note that from this display it is quite hard to make a statement about the proper depth order of the three contrast-enhanced arteries in the foreground. Now consider color plate l) where the same dataset is MIP projected with a 60˚ perspective viewing angle. Clearly, the perspective projection makes it easier to distinguish the arteries in general and provides the viewer with certain depth cues that facilitate the labeling of the displayed arteries with respect to their 3D location in the human head. Moreover, a 60˚ perspective viewing angle is often encountered in traditional X-ray angiography, and therefore the 60˚ perspective MIP display mimics best the type of environment that the clinician may be used to from previous work experience in the catherization unit.

Table 1 summarizes the performance of both voxel-driven and ray-driven splatting for the various rendering modes and data sets. In light of the previously mentioned analogies of ray-driven splatting with respect to raycasting, we also provide the run times required for rendering the data sets with this method. However, since pyramid integrals cannot be efficiently implemented with raycasting, we will omit comparisons of ray-based splatting with raycasting in this category. For the orthographic views, e.g. Brain 0˚, timings are given for both parallel and perspective projection. In these cases, the perspective renderer was run at a very small perspective angle (say 0.5˚) to enable the comparison of the overhead required for perspective mapping without confounding it with the effects of a wider viewing angle.

In Table 1, the stepsize for raycasting was set to Δs=0.3 grid units and the sample values were obtained from the grid voxels by trilinear interpolation. A lookup table was utilized to save compu-

Image	Line integral, parallel projection					Line integral, perspective projection					Pyramid integral, perspective projection		
	Ray-driven	Voxel-driven		Raycasting (trilin., Δs=0.3)		Ray-driven	Voxel-driven		Raycasting (trilin., Δs=0.3)		Ray-driven	Voxel-driven	
	Time	Time	S_p	Time	S_p	Time	Time	S_p	Time	S_p	Time	Time	S_p
Brain 0˚	23.7	27.5	1.16	72.0	3.04	25.9	58.0	2.24	73.0	2.82	53.1	133.0	2.50
Brain 30˚	-	-	-	-	-	30.3	60.0	1.98	80.22	2.64	61.8	142.1	2.30
Brain 60˚	-	-	-	-	-	29.5	55.7	1.89	81.9	2.77	61.0	130.0	2.13
Scalp 0˚	71.0	40.5	0.62	297.1	4.18	67.5	80.0	1.18	298.3	4.41	128.6	182.3	1.41
Scalp 30˚	-	-	-	-	-	63.9	85.1	1.33	237.3	3.71	122.9	185.0	1.50
MIP 0˚	128.0	92.5	0.72	479.8	3.74	165.5	370.0	2.23	497.1	3.00	567.9	1260.0	2.21
MIP 60˚	-	-	-	-	-	170.6	395.2	2.31	497.0	2.91	560.5	1258.8	2.24
X-ray 0˚	120.2	89.5	0.74	480.4	4.00	155.5	362.4	2.33	495.2	3.18	557.9	1240.6	2.22
X-ray 60˚	-	-	-	-	-	162.5	387.1	2.38	496.8	3.05	558.5	1243.4	2.23

Table 1: Comparison of ray-driven splatting with voxel-driven splatting and raycasting. The run time is measured in seconds, the speedup S_p=Time$_{\text{competing method}}$/Time$_{\text{ray-driven splatting}}$. (Timings are for a 200MHz SGI Indigo^2 workstation.)

tations. Note that the timings for raycasting would have been significantly higher had the (wider) Gaussian interpolation kernel given in Equation (4) been used in place of trilinear interpolation. To accelerate the Brain and Scalp isosurface renderings, both raycasting and ray-driven splatting used simple bounding boxes and opacity-based early ray-termination, while voxel-driven splatting only mapped and composited voxels above the isovalue threshold. The X-ray and MIP renderings did not use any acceleration methods.

The timings reveal that ray-driven splatting is superior in both perspective summation and MIP rendering where it generally outperforms voxel-driven splatting by a factor of around 2.3 for both line- and pyramid integral renderings. For orthographic X-ray and MIP-type projection, ray-driven splatting is less efficient than voxel-driven splatting, but the gap is by far not as large than it is in its favor in the perspective case. Since for MIP and X-ray rendering no acceleration methods are used, these rendering modes are suited best to compare the two splatting approaches in term of pure mapping overhead.

For isosurface rendering, ray-driven splatting is highly dependent on the quality of the bounding volume of the object, which is typical for all ray-driven methods. In the Brain data set a fairly tight bounding box could be found, which results in superior rendering speeds even for orthographic projections. The Scalp dataset had a somewhat less tight bounding box, resulting in inferior performance when compared to voxel-driven splatting. The timings for perspective projection of the Scalp dataset show that voxel-driven splatting overcomes the expense of its more expensive mapping algorithm by being able to restrict that mapping overhead to those voxels whose values fall above the iso-threshold. In contrast, ray-driven splatting's advantage of early ray termination is offset by the fact that it must traverse many empty voxels within the loose bounding box until the isosurface is found.

The speedup gains of ray-driven splatting with respect to raycasting (using trilinear interpolation) are in the range of 3 to 4 for all cases. This comes at no surprise since, at a ray stepsize of 0.3, every voxel is considered about 3 to 4 times per ray, as opposed to only once in the ray-driven splatting approach. The additional overhead for convolution is masked by the fact that we only used a 2×2×2 trilinear interpolation kernel for raycasting and not the 4×4×4 Gaussian that was used for ray-driven splatting.

Table 2 compares the run times for ray-driven splatting (using the Gaussian kernel) vs. raycasting (using trilinear and Gaussian interpolation filters at stepsizes Δs=1.0 and Δs=0.3) for two different image sizes of the Brain 30˚ rendering. With trilinear interpolation and a stepsize of Δs=1.0 the run time for raycasting is about equal to ray-driven splatting (but renders images of lower quality, as shown above), and for a stepsize of Δs=0.3 the run times are about 3 times larger than with Δs=1.0 for both filters. This is consistent with our observation in Table 1. The larger Gaussian filter increases the run time substantially.

Image size	Raycasting with trilinear interpol.		Raycasting with Gaussian interpol.		Ray-driven splatting
	Δs=1.0	Δs=0.3	Δs=1.0	Δs=0.3	
260*x*260	32.3	80.2	113.4	320.4	30.3
130*x*130	9.3	21.6	28.4	94.9	9.1

Table 2: Comparison of run times for raycasting for two different image sizes of the Brain 30˚ rendering. Timings are given in seconds.

5 CONCLUSIONS

We have described the transformation of the traditional voxel-driven splatting algorithm into a ray-driven approach. This enables us to take advantage of the inherent advantages of ray-based methods: ease of perspective viewing, front-to-back projection with the

opportunity of early ray termination when the accumulated opacity reaches unity, and the use of bounding volumes to cull unimportant structures. Other optimizations commonly used for raycasting, such as space-leaping [3][22], adaptive screen sampling, and octree decomposition [11], can be utilized to further speed up our algorithm. The minimal additional overhead required for perspective viewing in ray-driven splatting (as opposed to a large computational expense in voxel-driven splatting) enables speedups of around 2.3 for summed X-ray-type rendering, MIP, and direct volume rendering of predominantly translucent objects. For isosurface rendering, the speedup depends on the dataset and its bounding volume. Given a tight bounding box, ray-driven splatting is competitive for orthographic projection, and superior for perspective projection. If the dataset has many voxels above the isovalue, ray-driven splatting is advantageous to use since it only processes visible voxels proximal to the eyepoint, while back-to-front compositing voxel-driven splatting must process all. With a more loose fitting bounding box, the performance of ray-driven splatting is less superior but still competitive. In terms of image quality, ray-driven splatting is equivalent to voxel-driven splatting in orthographic projection. In contrast to voxel-driven splatting which is slightly inaccurate in perspective projection, ray-driven splatting avoids these errors by design.

When compared to raycasting, ray-driven splatting gains significant performance advantages by using splatting's scheme of precomputing the ray integrals and storing them in tables. This eliminates the need for interpolating sample values along the ray as is required for raycasting. In addition to being able to compute these ray integrals at high resolution (i.e. stepsize), we can also efficiently use interpolation kernels superior to trilinear interpolation. Consistent speedups of 3 to 4 were observed with ray-driven splatting when compared to a raycasting algorithm that used trilinear interpolation and a step size of 0.3. Perspective raycasting undersamples volume regions farther away from the eyepoint due to the diverging nature of the rays. This problem can be eliminated in ray-driven splatting (and voxel-driven splatting) by utilizing summed area footprint tables and tracing the volume by pyramidal raybeams.

In this paper we assumed an underlying cubic grid. In the future, we would like to expand our algorithm for use in general rectilinear grids. Westover used ellipsoidal splats to solve this problem. However, since our incremental algorithm for mapping the footprint polygons onto the image pixels depends on the basis kernel functions being spherical, a generalized implementation of the ray-driven splatting algorithm should maintain this concept. A convenient way to achieve this would be to warp the non-cubic rectilinear grid into a cubic grid. This new grid would then be composed of spherical kernel functions, and if the rays spawned at the image pixels are warped accordingly, then the traversal of the new grid could be performed using the same incremental algorithms as described in this paper, without significant speed penalties.

We also plan to enhance ray-driven splatting by using tighter, more intricate bounding volumes, possibly generated by a PARC-like scheme [17]. We would also like to experiment with hierarchical volume decomposition methods, such as the one described in [4], to accelerate the volume traversal of the ray.

ACKNOWLEDGEMENTS

The authors would like to thank GE for supporting this project under CCH grant #950925 and OSU grant #732025. Many thanks also to the University of North Carolina at Chapel Hill for maintaining the extensive database of test volumes from which the MRbrain dataset was obtained. We also thank Don Stredney from the Ohio Supercomputer Center for providing us with the MRA Scalp dataset.

REFERENCES

[1] R. Crawfis and N. Max, "Texture splats for 3D scalar and vector field visualization," *Visualization'93*, pp. 261-266, 1993.

[2] F.C. Crow, "Summed-area tables for texture mapping," *Computer Graphics SIGGRAPH'84*, pp. 207-212, 1984.

[3] D. Cohen and Z. Sheffer, "Proximity Clouds, an Acceleration Technique for 3D Grid Traversal", *The Visual Computer*, vol.11, no. 1, pp. 27-38, 1994.

[4] J. Danskin and P. Hanrahan, "Fast algorithms for volume raytracing," *1992 Workshop on Volume Visualization*, pp. 91-98, 1992.

[5] R. Gordon, R. Bender, and G.T. Herman, "Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography," *J. Theoretical Biology*, vol. 29, pp. 471-482, 1970.

[6] K.M. Hanson and G.W. Wecksung, "Local basis-function approach to computed tomography," *Applied Optics*, Vol. 24, No. 23, 1985.

[7] L. Hong, A. Kaufman, Y. Wei, A. Viswambharan, M. Wax, and Z. Liang, "3D virtual colonoscopy", *1995 IEEE Biomedical Visualization Symposium*, November 1995, pp. 26-32.

[8] Y. Kurzion and R. Yagel, "Space Deformation using Ray Deflectors," *Proceedings of the 6th EurographicsWorkshop on Rendering*, pp. 21-32, 1995.

[9] D. Laur and P. Hanrahan, "Hierarchical splatting: a progressive refinement algorithm for volume rendering," *Computer Graphics*, vol. 25, no. 4, pp. 285-288, 1991.

[10] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29-37, 1988.

[11] M. Levoy, "Efficient raytracing of volume data," *ACM Transactions on Graphics*, vol. 9, no. 3, pp. 245-261, 1990.

[12] R.M. Lewitt, "Alternatives to voxels for image representation in iterative reconstruction algorithms," *Physics in Medicine and Biology*, vol. 37, no. 3, pp. 705-715, 1992.

[13] R. Machiraju and R. Yagel, "Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors," *SUPERCOMPUTING'93*, pp. 699-708, 1993.

[14] K.L. Novins, F.X. Sillion, and D. Greenberg, "An efficient method for volume rendering using perspective projection", *Computer Graphics*, vol. 24, no. 5, Special issue on San Diego Workshop on Volume Visualization, pp. 95-102, 1990.

[15] P. Sabella, "A rendering algorithm for 3D scalar fields," *Computer Graphics*, vol. 22, no. 4, pp. 59-64, 1988.

[16] N. Shareef and R. Yagel, "Rapid previewing via volume-based solid modeling", *The Third Symposium on Solid Modeling and Applications, SOLID MODELING `95*, pp. 281-292.

[17] L.M. Sobierajski and R. Avila, "A hardware acceleration method for volumetric ray tracing," *Visualization'95*, pp. 27-34, 1995.

[18] H.K. Tuy and L.T. Tuy, "Direct 2-D display of 3-D objects," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 29-33, November 1984.

[19] L. Westover, "Footprint evaluation for volume rendering," *Computer Graphics SIGGRAPH'90*, vol. 24, no. 4, pp. 367-376, 1990.

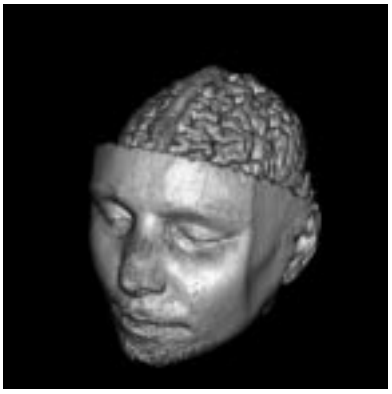
[20] L. Westover, "Interactive volume rendering," *1989 Chapel Hill Volume Visualization Workshop*, pp. 9-16, 1989.

[21] J. Wilhelms and A. Van Gelder, "A coherent projection approach for direct volume rendering," *Computer Graphics*, vol. 25, no. 4, pp. 275-284, 1991.

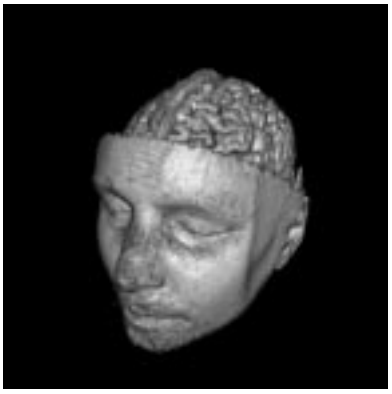
[22] R. Yagel, and Z. Shi, "Accelerating volume animation by space-leaping," *Visualization'93*, pp. 63-69, 1993.

Colorplates

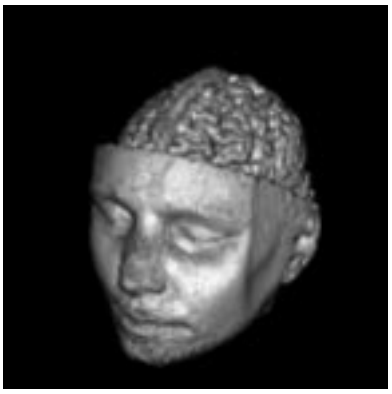
a. Brain 0˚ (ray-splat, line integral)



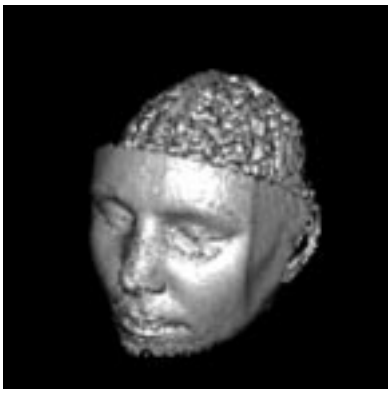
b. Brain 30˚ (ray-splat, line integral)



c. Brain 60˚ (ray-splat, line integral)

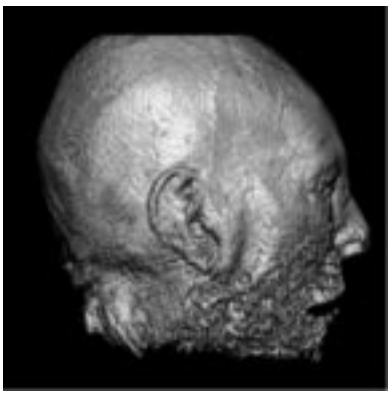


d. Brain 30˚ (ray-splat, low resolution)

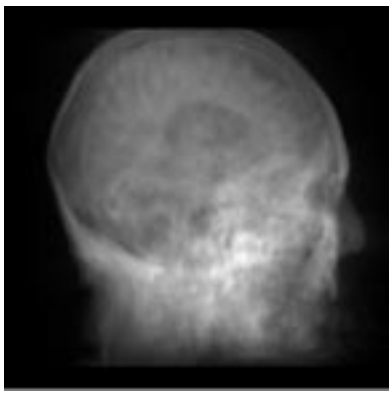


e. Brain 30˚ (raycast, Δs=1.0, low res.)

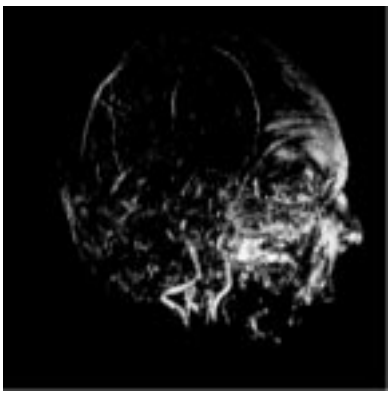
f. Brain 30˚ (raycast, Δs=0.3, low res.)



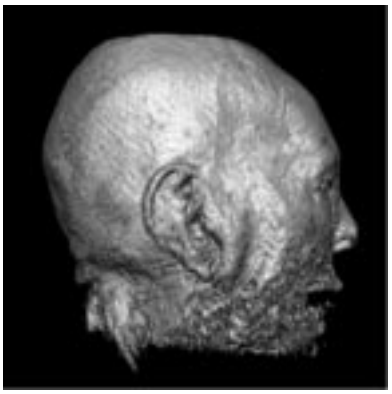
g. Scalp 0˚ (ray-splat, line integral)



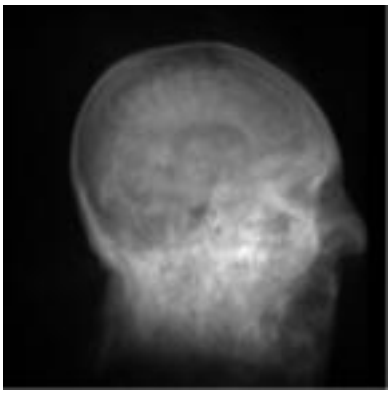
h. X-ray 0˚ (ray-splat, line integral)



i. MIP 0˚ (ray-splat, line integral)



j. Scalp 30˚ (ray-splat, line integral)



k. X-ray 60˚ (ray-splat, line integral)

l. MIP 60˚ (ray-splat, line integral)