High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels

Klaus Mueller, Naeem Shareef, Jian Huang, and Roger Crawfis

Abstract—Splatting is a popular volume rendering algorithm that pairs good image quality with an efficient volume projection scheme. The current axis-aligned sheet-buffer approach, however, bears certain inaccuracies. The effect of these is less noticeable in still images, but clearly revealed in animated viewing, where disturbing popping of object brightness occurs at certain view angle transitions. In previous work, we presented a new variant of sheet-buffered splatting in which the compositing sheets are oriented parallel to the image plane. This scheme not only eliminates the condition for popping, but also produces images of higher quality. In this paper, we summarize this new paradigm and extend it in a number of ways. We devise a new solution to render rectilinear grids of equivalent cost to the traditional approach that treats the anisotropic volume as being warped into a cubic grid. This enables us to use the usual radially symmetric kernels, which can be projected without inaccuracies. Next, current splatting approaches necessitate the projection of all voxels in the iso-interval(s), although only a subset of these voxels may eventually be visible in the final image. To eliminate these wasteful computations we propose a novel front-to-back approach that employs an occlusion map to determine if a splat contributes to the image before it is projected, thus skipping occluded splats. Additional measures are presented for further speedups. In addition, we present an efficient list-based volume traversal scheme that facilitates the quick modification of transfer functions and iso-values.

Index Terms—Splatting, volume rendering, visualization, rectilinear grids.

1 Introduction

Volume rendering has gained great popularity in recent years as it allows the user to comprehend and visualize a volumetric dataset directly without requiring the generation of a polygonal iso-surface. Maintaining a volumetric representation also enables easy manipulation and interaction with the object: Volume morphing [6], sculpting [26], and surgical simulations are just a few examples of the immense potential that such a representation has to offer. Volume rendering is appropriate for any discrete dataset acquired from a formerly continuous object via sampling. Most medical imaging technologies, such as MRI, CT, Ultrasound, PET, and SPECT, fall into this category, but scientific simulations, such as CFD and FEM, also generate their output on a discrete grid (typically irregular or curvilinear).

Medical applications mostly acquire their data as an axial stack of 2D slices, where each slice is uniformly sampled on a square grid. Oftentimes, the axial distance between slices is larger than the sample distance within a slice and, thus, a voxel is not a cube, but an elongated box with a square base. Interpolation of intermittent slices is required if one desires a cubic grid. This interpolation can either be performed using a standard interpolation kernel, such as a Gaussian or polynomial function, or by using shape-based interpolation [7] that seeks to better preserve the object's original shape. In any case, the interpolated slice data increases the magnitude of the already large volume datasets and, in order to maintain computational efficiency, it is more preferable that the volume renderer can deal with the unequal grid scaling without requiring extra interpolated slices.

A popular volume rendering method is the Splatting technique, proposed by Westover [27], [28], [29]. Here, each voxel is represented by a 3D kernel, weighted by the discrete voxel value. The algorithm gains its speed by pre-integrating the 3D kernel into a generic 2D footprint. This footprint can be efficiently mapped onto the image plane and the collection of all projected footprints, weighted by the voxel values, then forms the final image. By mapping the footprint (image) onto a polygon, we can employ standard 2D texture mapping hardware for the projection process [2]. On the other hand, the footprint interpolation is also easily done in software with fast DDA and rasterization procedures [13], [16].

There are some disadvantages to the splatting approach: The use of pre-integrated kernels introduces inaccuracies into the compositing process since the 3D reconstruction kernel is composited as a whole, and not piecewise, as part of an interpolated sample along a viewing ray. Due to this circumstance, the colors of hidden background objects may bleed into the final image [27], [29]. The sheet-buffer method was proposed to eliminate this problem [28], [29]. Here, the splats are added within consecutive cache-sheets, represented by the volume slices most parallel to the image plane. The sheets are subsequently composited together in a back-to-front (or front-to-back) order. This method bears the disadvantage that disturbing popping artifacts appear in animated viewing. The artifacts occur at those viewpoint transitions in which the view plane becomes more parallel to a different volume face, causing the orientation of the compositing sheets to flip abruptly by 90°. One should be

• *The authors are with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210. E-mail: muellerk@acm.org.*

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 109341.

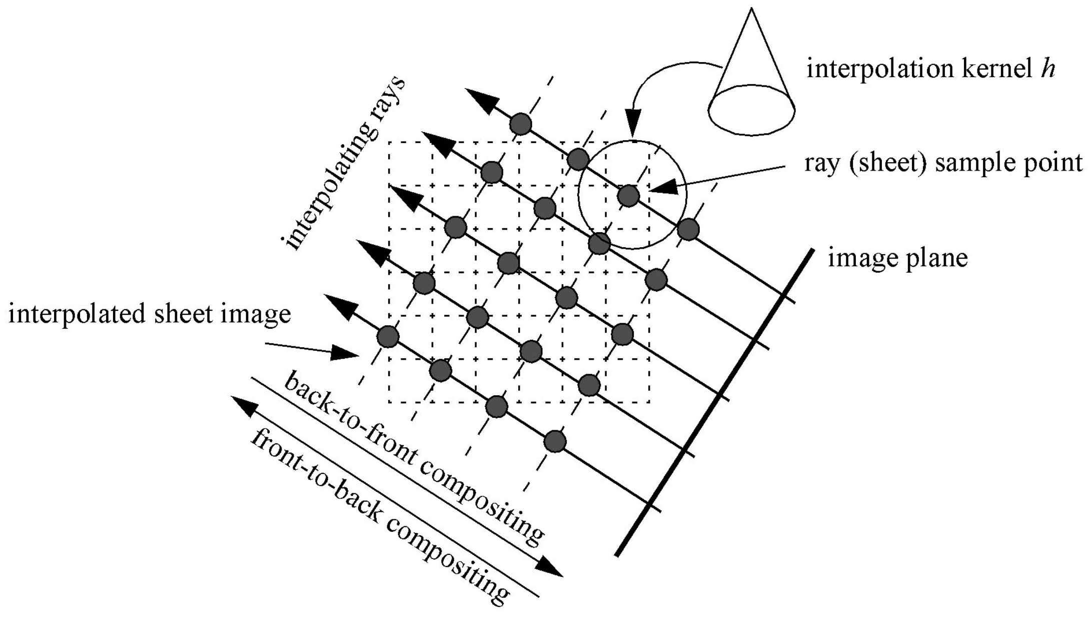


Fig. 1. Volume rendering by compositing equidistant interpolated sheets in front-to-back or back-to-front order. Sheet-interpolated volume rendering is equivalent to raycasting with simultaneous rays. A slice (i.e., ray) sample point is obtained by interpolating the discrete volume grid with an interpolation kernel (shown as a tent filter here). This reconstructs (within some margin of error) the original volume function at the sample position.

aware, however, that popping artifacts are not unique to splatting, but potentially occur for any algorithm that abruptly switches the direction in which the volume is composited. For example, the shear-warp algorithm [10] also exhibits this problem, as do algorithms that composite volume slices mapped as textures onto 2D polygons (see e.g., [24]).

Splatting can be performed either in object-order [27], [28], [29] or, as a ray-based approach, in image-order [16]. Each approach offers its own set of acceleration techniques: iso-voxel lists [3] and splat hierarchies [11] for the object-order technique, and space leaping [31], bounding boxes [23], and early ray-termination [4] for the ray-based approach. In terms of limiting the splatting effort to just the voxels within the iso-range(s), it is clear that space leaping and bounding boxes in the ray-based approach will rarely be as effective as the explicit iso-voxel list used in object-order approaches. However, early ray termination is also a very powerful acceleration technique, but currently confined to ray-based splatting only. In contrast, the object-order splatting techniques typically project all of the voxels in the iso-range, using a painter's algorithm type traversal, even though many voxels may not be visible in the final image. This is obviously wasteful in computational effort. Clearly, great savings would ensue if we could combine the advantages of early ray termination with those of iso-voxel lists. For this to happen, we would need a screen occlusion map in conjunction with a front-to-back object-order traversal. This map would then be referenced by each iso-voxel to determine its (partial) visibility. In this way, the expensive splatting operation could be avoided for any voxel that does not pass the partial visibility test. However, maintaining and indexing a screen occlusion map for splatting is not as simple as in point-based approaches such as Reynolds et al.'s Dynamic Screen [20] or Lacroute and Levoy's Shear-Warp algorithm [10], since a splat potentially affects many image pixels and not just one.

This paper is organized as follows: First, in Section 2, we discuss some relevant background material. Then, in Section 3, we examine the popping artifacts that occur in traditional sheet-buffered splatting and summarize our image-aligned sheet-buffered splatting approach, previously presented in greater detail in [18] that eliminates these artifacts. Next, in Section 4, we extend our algorithm to rectilinear grids and describe a new grid-warping technique that removes the original splatting approach's requirement for ellipsoidal kernels. This avoids the inaccuracies of this approach, but retains its efficiency. Next, in Section 5, a new list-based volume traversal method is described that allows both sheet-buffered splatting methods, i.e., the traditional method and our new algorithm, to execute efficiently while enabling quick modification of transfer functions and iso-ranges. Then, in Section 6, a novel object-order, front-to-back traversal scheme is described that uses the object-order equivalent to early ray-termination, i.e., early splat-elimination, to limit the splatting effort to those splats that actually contribute to the final image. Considerable speedups result from this scheme. In Section 7, we present timings and show images that were obtained with the new enhancements. Finally, we discuss future work in Section 8.

2 Preliminaries

The basic element in most volume rendering applications is the volume rendering integral in its low-albedo form, first described by Kajiya and Von Herzen [9]. For each pixel ray, we compute $I_\lambda(x, r)$, the amount of light of wavelength λ coming from ray direction r that is received at point x on the image plane:

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

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 toward the observer according to the specular and diffuse material properties of the particles [14], [21]. Thus, in (1), ϕ_λ is the light of wavelength λ reflected at location l in the direction of $\boldsymbol{r}$. Since volume particles have certain densities μ (translated to opacities), the light scattered at l is attenuated by the particles between l and the eye according to the exponential attenuation function.

Usually, (1) cannot be solved analytically and a discretized form involving a Riemann sum is used. This can be accomplished with raycasting, where a ray samples the volume at equidistant points, integrating the sampled colors and opacities from front to back (see the individual rays in Fig. 1). A ray sample is obtained by placing an interpolation kernel h at the ray sample location and weighting the surrounding volume grid samples by the kernel function. The interpolated values can either be used directly or as an index into transfer functions for color, opacity, and also gradients.

The process of integrating colors and opacities along the ray is called *compositing*. The exponential form of the attenuation in (1) is commonly approximated by the **over** or **under** operators, respectively, described by Porter and Duff [19]. These operators approximate the exponential function by the first two terms of its Taylor series expansion, $1 - \int \mu(t)dt$. This is commonly modeled by $1 - \alpha$, where α is called opacity. In a front-to-back traversal, the **over** operator composites a newly interpolated back sample with the current partial integral, representing the front sample in this context. The compositing process is a weighted sum of the two samples, based on their opacities. For the **over** operator (the **under** operator is just the complement), a composited color c_0 is computed from a back sample with color/opacity (C_B, α_B) and a front sample (C_F, α_F) with the following expression:

$$c_0 = C_B\alpha_B(1 - \alpha_F) + C_F\alpha_F = c_B(1 - \alpha_F) + c_F. \qquad (2)$$

Here, the colors written in lowercase denote sample colors that were premultiplied by their respective sample opacities. By using the **over** operator in sequence, all sample values along a ray can be composited in this way. Note that the newly composited color, c_o, is implicitly premultiplied by the composited opacity $\alpha_0 = \alpha_B(1 - \alpha_F) + \alpha_F$.

Porter and Duff [19] used their framework to composite multilayer cel images. Volume rendering can be represented in this framework as well. By interpolating a volume into a stack of (image) sheets, aligned parallel to the projection plane, we can render a projection image by compositing these sheets in back-to-front (or front-to-back) order [5]. This sheet-interpolated volume rendering approach is illustrated in Fig. 1 and it is easy to see that we obtain results equivalent to those obtained with raycasting, just that now all rays are being traced simultaneously. Hence, the approximation of the volume rendering integral of (1) will not change and the orientation of the interpolated sheets in Fig. 1 is the most appropriate for a discrete approximation of (1). Note that, as in raycasting, the sheet (i.e., intersheet sample) spacing must be chosen sufficiently small to avoid aliasing effects.

The sheet-interpolation approach is taken by various implementations that utilize 3D texture-mapping hardware for volume rendering, such as [25] and [1], and the approach was also mentioned by Westover in his dissertation [29] as the ideal approximation to volume rendering. However, it is an approach that does not lend itself well to acceleration methods (other than 3D texture mapping), since the sheets slice the volume data indiscriminately of their value.

As was mentioned in the introduction, splatting gains its speed by reordering the volume rendering integral so that each voxel's contribution to the integral can be viewed isolated from the other voxels. In splatting, an interpolation kernel is placed at each voxel location. This enables one to view the volume as a field of overlapping interpolation kernels h which, as an ensemble, make up the continuous object representation. A voxel v_j's contribution is then given by $v_j \cdot \int h(l)dl$, where l follows the integration of h in the direction of the ray. If the viewing direction is constant for all voxels or if h is radially symmetric, we may pre-integrate $\int h(l)dl$ into a lookup table, i.e., the kernel footprint, and use this table for all voxels. We can then map the voxel footprints, weighted by the voxel values, to the screen, where they accumulate into the projection image [28]. Thus, in contrast to raycasting, splatting considers each voxel only once (for a 2D interpolation on the screen), and not several times (for a 3D interpolation in world space). Additionally, as an object-order approach, only the relevant voxels need to be considered, which, in many cases, constitute only 10 percent of the volume voxels [30]. Apart from these computational savings, we also achieve greater accuracy: First, the ray integrals across a voxel are now continuous (or approximated with good quadrature) and not a Riemann sum as implied by the point sampling process of raycasting. Second, the efficient pre-integrated kernel representation allows splatting to easily use qualitatively better interpolation kernels (with larger extents) than the trilinear filter typically employed by raycasting. The downside of splatting is that the use of pre-integrated kernels restricts a proper approximation of the volume rendering integral, since the 3D reconstruction kernel is composited as a whole, and not piecewise, as part of an interpolated sample along a viewing ray. Our approach alleviates this problem.

3 Image-Aligned Sheet Buffer-Based Splatting

As was mentioned in the introduction, the first splatting algorithm used the composite-every-sample approach, which violates the sheet-interpolation model of Fig. 1 in that a volume sample point is not first reconstructed based on the values of the surrounding voxels before its visibility is determined. Instead, each voxel is independently composited on the image plane, without spreading its contribution along the main viewing axis [29]. The result of this approximation is a continuous sparkling of colors in

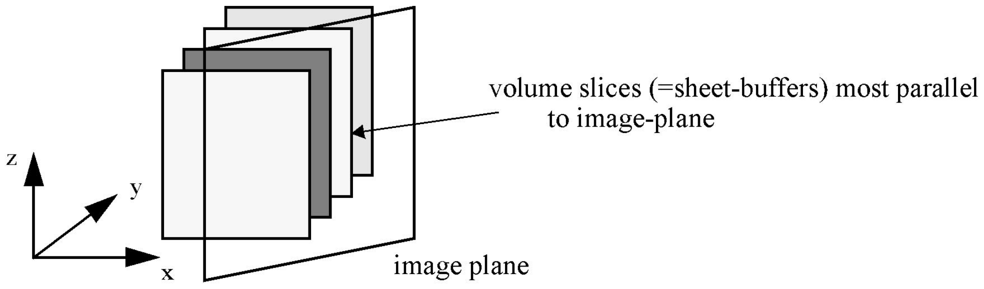


Fig. 2. Axis-aligned sheet-buffered splatting. The volume is decomposed into sheets, where the sheets are comprised of the volume slices most parallel to the image plane (here, those along the y-axis). The colors and opacities of all voxel kernels within a sheet are added into a sheet-buffer and the sheet-buffers are composited in front-to-back or back-to-front order.

animated viewing. The sheet-buffer method was prescribed to eliminate these artifacts and we will describe it next.

3.1 Artifacts of Traditional Axis-Aligned Sheet Buffer-Based Splatting

In Westover's sheet-buffer method [28], the volume is decomposed into sheets, where the sheets are comprised of the volume slices most parallel to the image plane (see Fig. 2). Hence, we refer to this algorithm as axis-aligned sheet buffered splatting. For all voxels in a sheet, their color and opacity footprints are added into a color and an opacity sheet buffer, respectively, and the resulting sheet images are composited in back-to-front (or front-to-back) order. Thus, due to this inter-sheet reconstruction process, the axis-aligned sheet-buffer splatting method comes significantly closer to the discrete volume rendering model of Fig. 1 than the composite-every-sample approach.

Disturbing popping artifacts occur when the main orientation of the sheet-buffers abruptly switches from one volume axis to another, which happens when the image plane becomes more perpendicular to another volume face. An example of this artifact is shown in the rendering of a binary volume cube illuminated with a headlight, shown in Fig. 3. In Fig. 3a, we show the cube viewed at a 44.8° angle. Notice that the left face is much brighter than the right face. Fig. 3b shows the cube viewed at an angle of 45.2°, right after the orientation of the sheet-buffers have switched. Here, the right cube face is much brighter than the left face. Neither of the two renderings are correct, since both faces should have the same shade and neither should be as bright. This switching of the bright areas from the left to the right is very visible in animated viewing, and constitutes what we refer to as a "popping" artifact.

In a previous paper [18], we illustrated by means of a detailed numerical example how these artifacts are generated. This discussion is summarized in the caption of Fig. 3. We shall now proceed to derive a solution to the popping problem.

3.2 New Image-Aligned Sheet Buffer-Based Splatting to Eliminate Popping

We have justified before that the discrete volume rendering model of Fig. 1 can be regarded as a good approximation to the analytical volume rendering integral of (1). In this model, parallel slices cut across the volume, interpolating it into a sequence of 2D images which are then composited back-to-front or front-to-back. The interpolation planes are aligned parallel to the projection plane. We observe that the axis-aligned sheet-buffered splatting method violates the discretized volume rendering model in two ways: 1) the contributions of a voxel are added all at once, instead of being composited along a ray, and 2) the sheet buffers do not maintain a constant spatial orientation with respect to the image plane. We conclude that a splatting algorithm that does not suffer from popping artifacts must eliminate these two violations.

We must realize that splatting has a somewhat different volume representation than that implied by the discretized volume rendering model. In splatting, the volume is not an array of discrete data points that are used to support 3D interpolation, rather, it is a field of overlapping 3D spherical interpolation kernels, each chopped into (sphere) sections by the parallel slicing planes. The thickness of these kernel sections is determined by the distance between the slicing planes (i.e., distance between interpolation planes in Fig. 1). Since we add to the sheet buffer all kernel material that is bounded by two slicing planes and not just the kernel cross-section that is cut by a slicing plane, we must view a pair of slicing planes as a slicing slab of certain width (or thickness).

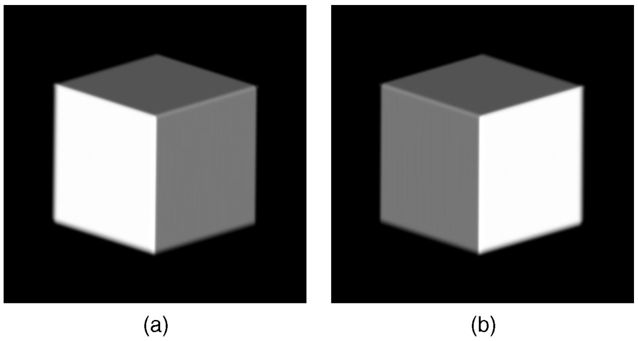


Fig. 3. An opaque cube is rendered with sheet buffer-based-splatting at orientations of (a) 44.8° and (b) 45.2°. Only surface voxels contribute to the image, since all others have zero gradients. In (a), the sheet buffers are parallel with the left cube face and, in (b), they are parallel with the right cube face. The bright face is due to the voxel kernels all added together in one sheet. The dark face is due to kernels that lie in consecutive sheet-buffers. It was shown that the addition of a number of kernels yields a brighter color than compositing the same number of kernels [18].

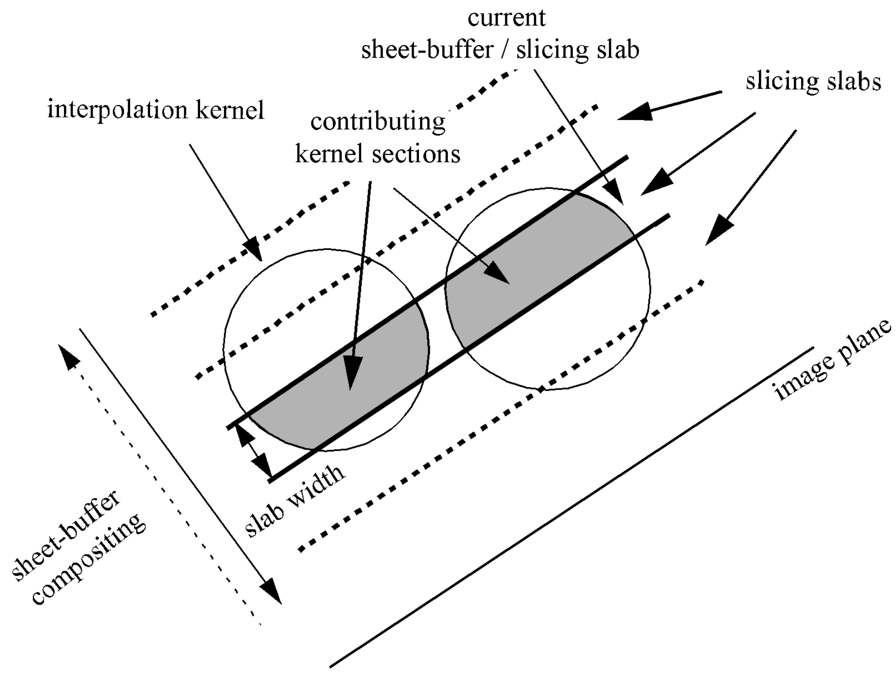


Fig. 4. Image-aligned sheet-buffered splatting. All kernel sections that fall within the current slicing slab, formed by a pair of interpolation planes, are added to the current sheet buffer.

Fig. 4 illustrates our approach. The sheet buffer is now parallel to the image plane and only the contributions of the kernel sections that fall within the extent of the current slicing slab are added to the sheet buffer. Then, similar to the axis-aligned sheet-buffer method, once a sheet buffer has received all contributions, it is composited with the current accumulation buffer and the algorithm progresses to the next slicing slab.

This new method requires pre-integrated kernel sections. Since the slicing planes may intersect an interpolation kernel at any radial distance, this would require an infinite number of pre-integrated slab sections. Alternatively, we approximate this with a discrete set of slab positions (see Fig. 5). Nearest-neighbor interpolation is used to pick the most appropriate pre-integrated kernel section when a slab intersects a voxel kernel. We use $S = 128$ such sections, spaced apart by $\Delta s = (kernelRadius + slabWidth/2)/128$. Note that symmetry allows for the reuse of sections for part of the kernel width. In addition, we can also trim the size of each slice footprint to the relevant extent of the integrated slice function.

Figs. 6a and b show the cube of Fig. 3, now rendered with the new image-parallel sheet-buffer method. A Gaussian kernel of radius 2.0 grid spacings and a slab width of 1.0 was used. We observe that the previous imbalance of brightness between the two cube faces no longer exists. The cube has the same (correct) shades for both image plane orientations, 44.8° and 45.2°, and the popping no longer occurs. Notice also that the overall brightness of the cube is reduced, as now the kernel contribution has been divided into four parts which are composited back-to-front, rather than being added all at once. This represents an improvement in the splatting method and brings it more in line with the ideal discrete volume rendering model. It, however, retains the advantages of splatting in that the ray integral is still continuous. We can also easily use a sparse data representation and not have to traverse potentially empty volume areas.

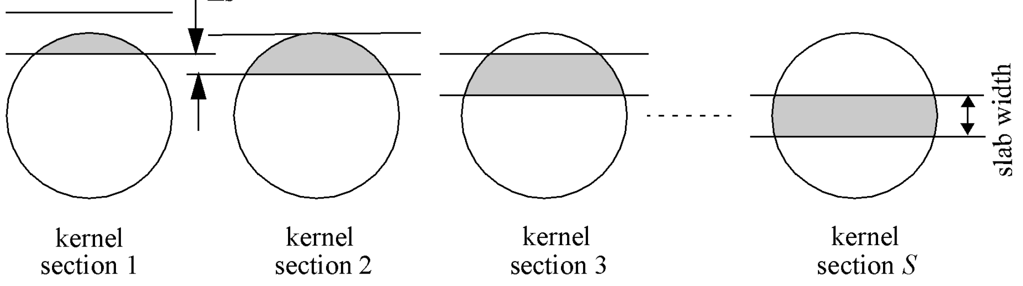


Fig. 5. Array of pre-integrated overlapping kernel sections (shaded areas). The integration width of the pre-integrated sections is determined by the slab width. The offset between adjacent slabs is Δs.

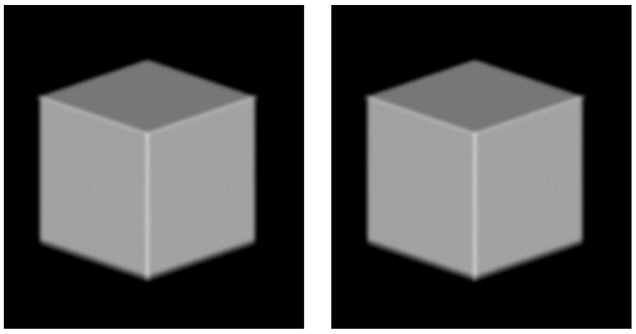


Fig. 6. Binary cube rendered with the new image-parallel sheet buffer splatting method at: (a) 44.8° and (b) 45.2°. In both cases, the slab width was set to 1.0. Both front faces now have equal brightness in both views. Not only is this correct, but it also implies that popping at 45° no longer occurs.

3.3 Enhancements

Following are two enhancements that are unique to our image-aligned sheet-buffer splatting method.

3.3.1 *Variable Slab Width*

Varying the slab width trades off speed vs. accuracy. Clearly, the thinner we make the slabs, the closer we approximate splatting to the continuous volume rendering integral of (1). However, the number of kernel sections to be mapped per voxel is given by $2 \cdot kernelRadius/slabWidth$. Thus, for smaller slab widths the number of kernel sections to be rasterized increases, along with the number of sheet buffers to be composited. On the other hand, thicker slabs decrease the number of footprint mappings, as well as the number of compositing operations, but will potentially lower the quality of the generated image.

If we would like to vary the slab width on the fly, we need to provide an array of pre-integrated kernel sections for every slab width we intend to use. A continuous variation of the slab width would require us to store a large number of kernel-section arrays, one for each quantized slab width. Nearest neighbor interpolation could be employed to pick the most adequate array for a given slab width. Clearly, this approach is both wasteful in storage and bound to be inaccurate.

Instead, we utilize a 2D array of 1D summed-area tables, one for each 2D raster position (x, y), in which we store the kernel integrals:

$$H(x, y, z) = \int_{t=0}^{z} h(x, y, t)dt. \tag{3}$$

Thus, in order to calculate the kernel integration between z and $(z + slabWidth)$ at footprint location (x, y), we just retrieve two entries, $H(x, y, z)$ and $H(x, y, z + slabWidth)$, and subtract the former from the latter. In this way, we can vary the slab width on the fly in a continuous manner. This facilitates a variety of unique acceleration methods, based on visual quality, as will be discussed later.

3.3.2 *Perspective Projection*

The algorithm is easily expanded to perspective; the volume traversal and kernel section selection remain the same. Splatting assumes parallel ray integration across a kernel and, hence, introduces errors for perspective projection where rays integrate a kernel along diverging paths. Due to the partitioning of the splats, we obtain a better approximation of the perspective integration. Although we still assume parallel integration within kernel sections, the mapping of the individual section footprints can be performed according to the linear perspective distortion function. Thus, we obtain a piecewise linear perspective kernel mapping. The necessary tilt of the kernels toward the direction of the traversing rays is also more accurately

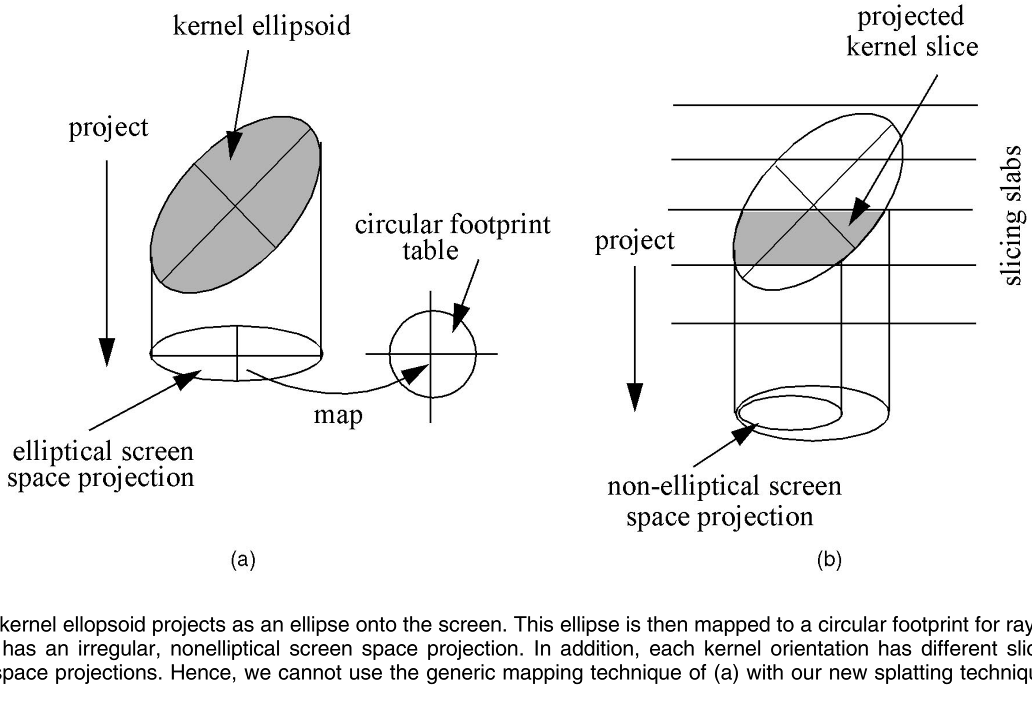


Fig. 7. (a) The full kernel ellopsoid projects as an ellipse onto the screen. This ellipse is then mapped to a circular footprint for ray integral retrieval. (b) A kernel slice has an irregular, nonelliptical screen space projection. In addition, each kernel orientation has different slices with different, unrelated screen space projections. Hence, we cannot use the generic mapping technique of (a) with our new splatting technique.

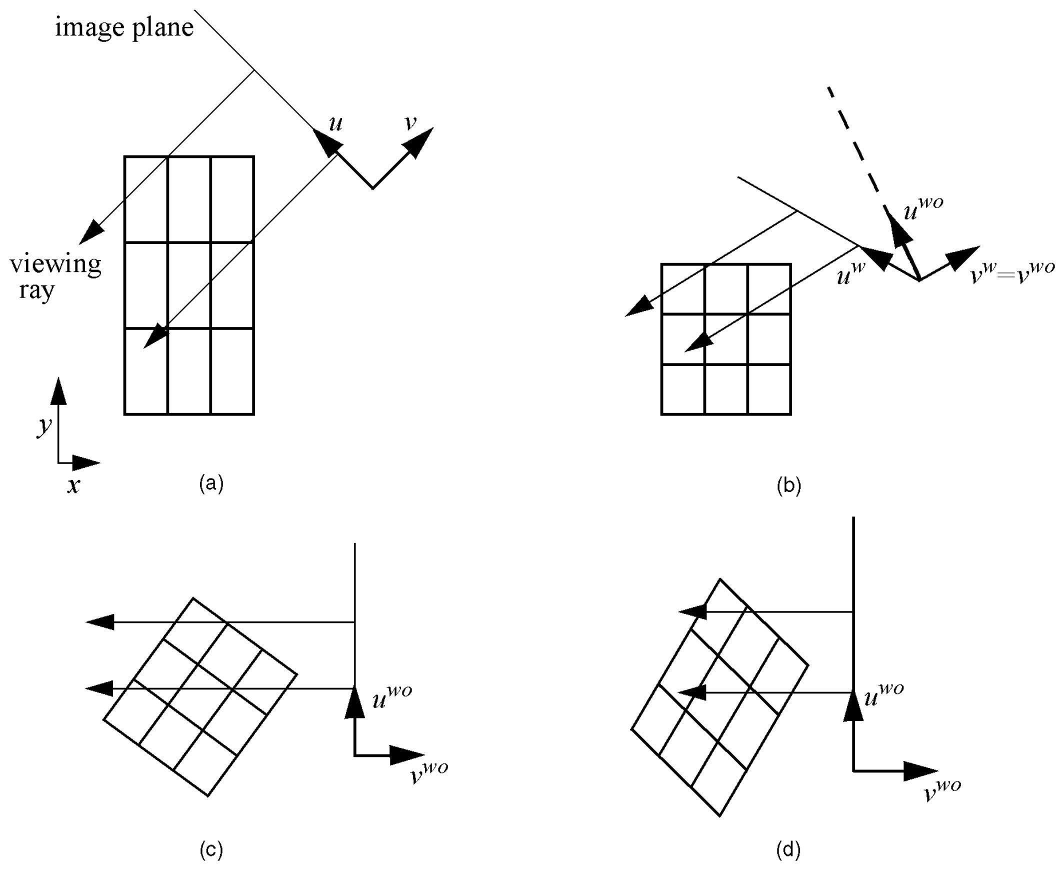


Fig. 8. Splatting with 2D rectilinear grid warping. (a) A rectilinear volume defined in (x, y) space and an image plane with viewing coordinate system (u, v). Two representative viewing rays are also shown. (b) The warp operation not only warps the grid shown in (a) into a cubic grid, but the image plane and the viewing rays are warped as well. The viewing coordinate system is warped into (u^w, v^w). Vector u^w is projected onto a plane that is orthogonal to the viewing rays, which generates u^{wo}. (c) The system is rotated for projection. (d) Finally, the system is scaled so that the image plane has the original size (most often comprised of unit-sized pixels).

implemented with sectioned kernels. (See [17] for more details on accurate perspective splatting).

4 Splatting Rectilinear Volumes Using Grid Warping

Rectilinear volume grids are more difficult to render using the splatting approach because at least one grid direction has unequal scaling. Under the splatting paradigm, this means that splatting kernels are ellipsoidal in shape because the spherical splat is stretched along the scaled directions. Two approaches have been suggested to render such volumes. A straightforward approach interpolates additional slices so that the resulting grid will be regular and the splat kernels will be spherical. This, however, requires the costly processing of more splats. Alternatively, Westover proposed splatting the rectilinear grid directly with elliptical shaped footprints. Since the footprint shape is view dependent, the elliptical screen projection of a splat is mapped back to a circular footprint (see Fig. 7a). This method does not easily extend to our splatting approach because now kernel slices are projected to the screen (see Fig. 7b). Since the shape of the kernel slices is view dependent, a single generic footprint table cannot be used to account for all mappings. In fact, a separate hierarchy of slice footprints is required for each kernel orientation. In most cases, storing (and loading) this many kernel slice projections is infeasible. Instead, we devise a method that warps the noncubic grid into a cubic grid so that the ellipsoidal kernels compress into spheres which project identically for all view orientations. This mapping requires rotation, shear, and scaling operations as well, since the warp operation warps both sampling grid and the image plane. For ease of understanding, we first illustrate the general idea for the 2D rectilinear volume case and then formalize the procedure for the 3D grid case.

4.1 Splatting a 2D Rectilinear Grid

The first step of our approach is to warp the rectilinear grid so that the ellipsoidal splat kernels transform into spherical kernels. Fig. 8a shows a 2D rectilinear grid where the spacing in the y-axis direction is twice that in the x direction and the image plane is defined by the unit vectors u and v. When the grid is warped into a cubic grid, shown in Fig. 8b, the image plane is warped as well and is now defined by vectors u^w and v^w, which have a length different from the original vectors. The viewing rays also change orientation, due to the warping operation, and are no longer perpendicular to the (warped) image plane. We would like to preserve orthographic projection of the volume. This can be achieved by erecting an image plane perpendicular to the

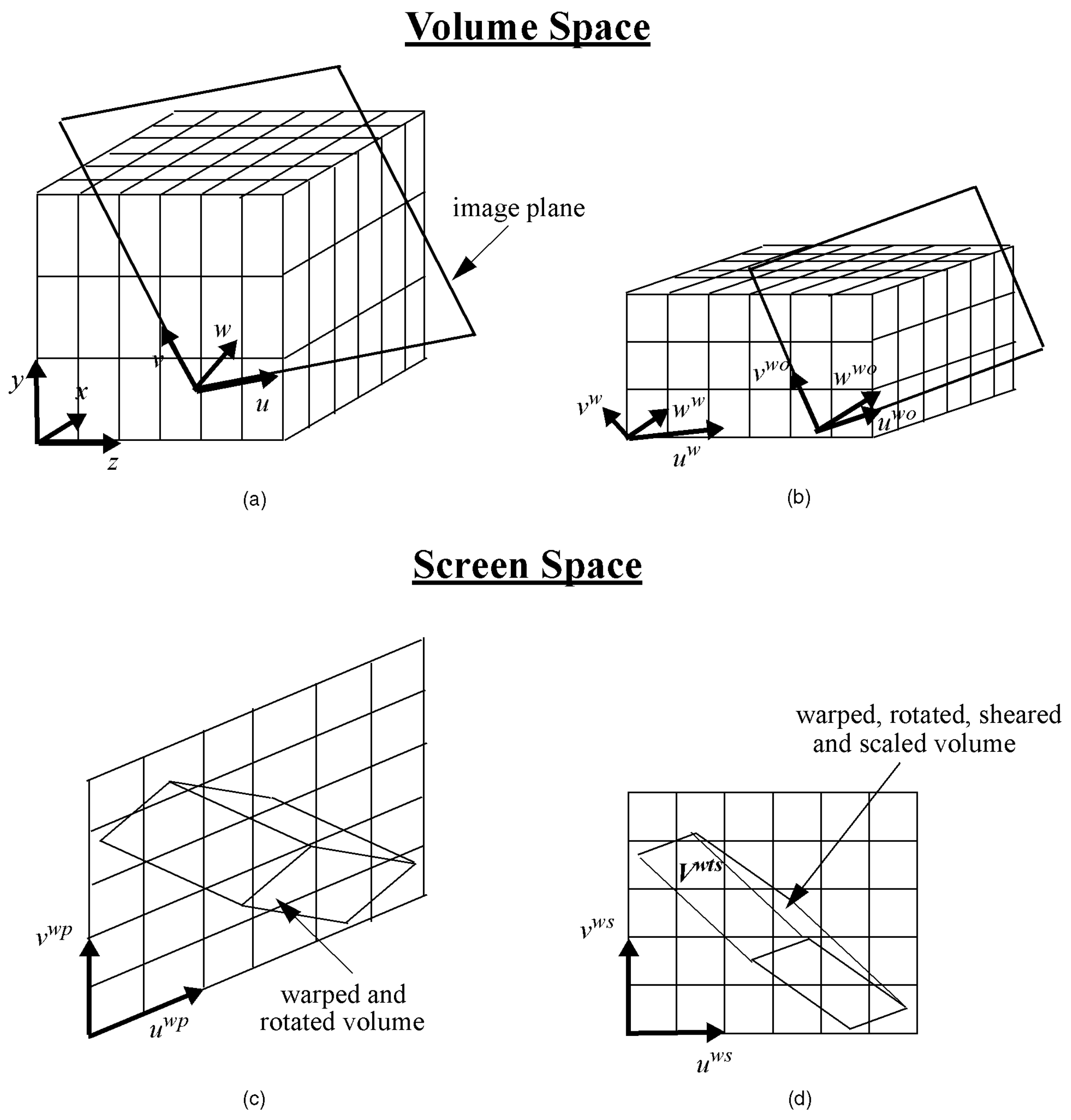


Fig. 9. (a) Original volume space V. the grid scaling along the y-axis is twice that of both the x- and z-axis. (b) Warped volume space V^w. V is warped into a cubic grid V^w, which also warps the rotation vectors and the image plane. (c) Warped image space V^{wt}. The volume has been rotated into image space. The image grid is nonsquare. (d) Shearing and scaling of the volume transforms the image grid into a square grid. Projection of the (distorted) voxel footprints is now simple.

warped viewing rays. This image plane is defined by projecting the warped image plane vector u^w onto this new, ray-orthogonal image plane, which produces vector u^{wo}. The system is now rotated such that the viewing axis $v^{wo} = v^w$ becomes horizontal for easy projection (see Fig. 8c). Vector u^{wo} defines the pixel size in the viewing plane and is not unit-length, as mentioned before. To return to unit pixel size, we need to scale the grid, along with the vertical image plane, in the direction of vector u^{wo} (see Fig. 8d). This operation scales the image plane back to its original size and stretches the footprints parallel to the image plane. Why? Recall, that the footprints must always be aligned perpendicular to the direction of the traversing rays, since they represent the kernel integration in this direction. Thus, before the scale, the footprints have already been aligned parallel to the image plane. The scale then simply stretches them (without changing their amplitude), which is most easily achieved in both hardware and software, representing the footprints as texture polygons.

4.2 Splatting a 3D Rectilinear Grid

Figs. 9a-9d illustrate the step-by-step procedure required to splat a 3D rectilinear grid. Fig. 9a shows a volume grid V in which the grid spacing along the y-axis is twice as large as the grid spacing along the x- and z-axis, and the grid aspect ratio vector is represented by $AR = (ar_x, ar_y, ar_z)$. The image plane is defined by the unit vectors u, v, and w (the viewing coordinate system), which also help define the world-to-screen space viewing transformation. Using these vectors, the viewing transformation matrix M_{view} that transforms the original rectilinear grid can be written as follows:

$$M_{view} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \\ \boldsymbol{w} \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix}. \tag{4}$$

Similar to the 2D case, to transform this rectilinear grid into a cubic grid, a linear warp is applied to the system that transforms $\boldsymbol{V}$ into a new grid $\boldsymbol{V}^w$ that has equal grid spacing in all dimensions (see Fig. 9b). The traversing rays, the image plane and the vectors $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ all undergo the same warp and the new warped viewing coordinate system is now defined by the vectors $\boldsymbol{u}^w$, $\boldsymbol{v}^w$, and $\boldsymbol{w}^w$. This gives rise to a viewing transform that incorporates the warping:

$$M^w_{view} = \begin{bmatrix} \boldsymbol{u}^w \\ \boldsymbol{v}^w \\ \boldsymbol{w}^w \end{bmatrix} = \begin{bmatrix} \frac{u_x}{ar_x} & \frac{u_y}{ar_y} & \frac{y_z}{ar_z} \\ \frac{v_x}{ar_x} & \frac{v_y}{ar_y} & \frac{v_z}{ar_z} \\ \frac{w_x}{ar_z} & \frac{w_y}{ar_y} & \frac{w_z}{ar_z} \end{bmatrix}. \tag{5}$$

Just like the 2D case, our 3D grid warping algorithm performs a rotate, then a shear (this is unique to the 3D case), and, finally, a scale operation. We now derive the matrices for each of these operations. For a rigid-body rotate operation, we must construct an orthonormal matrix consisting of the vectors $\boldsymbol{v}^{wo}$, $\boldsymbol{v}^{wo}$, $\boldsymbol{w}^{wo}$ (shown in Fig. 9b). These are obtained from $\boldsymbol{u}^w$, $\boldsymbol{v}^w$, and $\boldsymbol{w}^w$ as follows: First, vector $\boldsymbol{w}^{wo}$ is computed by $\boldsymbol{w}^w/|\boldsymbol{w}^w|$, the unit vector normal to the viewing plane. Vector $\boldsymbol{v}^{wo}$ is found by projecting the vector $\boldsymbol{v}^w$ onto the plane normal to $\boldsymbol{w}^{wo}$:

$$\boldsymbol{v}^{wo} = \frac{\boldsymbol{v}^w - (\boldsymbol{v}^w \bullet \boldsymbol{w}^{wo}) \cdot \boldsymbol{w}^{wo}}{|\boldsymbol{v}^w - (\boldsymbol{v}^w \bullet \boldsymbol{w}^{wo}) \cdot \boldsymbol{w}^{wo}|}. \tag{6}$$

The third unit vector $\boldsymbol{u}^{wo}$ is computed from the cross product $\boldsymbol{v}^{wo} \times \boldsymbol{w}^{wo}$, resulting in the orthogonal rotation matrix:

$$M^{wo}_{view} = \begin{bmatrix} \boldsymbol{u}^{wo} \\ \boldsymbol{v}^{wo} \\ \boldsymbol{w}^{wo} \end{bmatrix}. \tag{7}$$

Next, we define vectors $\boldsymbol{u}^{wp}$ and $\boldsymbol{v}^{wp}$ as the vectors $\boldsymbol{u}^w$ and $\boldsymbol{v}^w$ projected onto the image plane, respectively:

$$\begin{aligned} \boldsymbol{u}^{wp} &= \boldsymbol{u}^w - (\boldsymbol{u}^w \bullet \boldsymbol{w}^{wo}) \cdot \boldsymbol{w}^{wo} \\ \boldsymbol{v}^{wp} &= \boldsymbol{v}^w - (\boldsymbol{v}^w \bullet \boldsymbol{w}^{wo}) \cdot \boldsymbol{w}^{wo}. \end{aligned} \tag{8}$$

These vectors span the 2D image grid, illustrated in Fig. 9c, onto which the warped and rotated volume $\boldsymbol{V}^{wt}$ is projected. Note that, in this figure, the rotated vector $\boldsymbol{w}^{wo}$ points out of the page and also that $\boldsymbol{u}^{wp}$ and $\boldsymbol{v}^{wp}$ are not orthogonal. Hence, this situation requires the footprints to be rasterized onto this non-orthogonal grid. Ray-driven splatting [16] can handle this situation, since here the individual rays can be spawned anywhere. However, for object-order splatting, we either need a rasterization routine that scans a footprint into a sheared image grid or an additional transformation that distorts a footprint so it can be rasterized on an orthogonal image grid with unit spacing. The latter is needed for an implementation that uses 2D texture mapping hardware for footprint projection. This type of grid can be obtained by shearing and scaling the grid spanned by $\boldsymbol{u}^{wp}$ and $\boldsymbol{v}^{wp}$. Alternatively, we can impose the same operations on the volume, which will then project correctly onto an orthonormal image grid with vectors $\boldsymbol{u}^{ws}$ and $\boldsymbol{v}^{ws}$. This gives rise to the volume $\boldsymbol{V}^{wts}$ shown in Fig. 9d. Shearing is only required in one grid direction, here the y-direction and is computed with:

$$M_{shear} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{u^{wp}_y}{u^{wp}_x} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

The following scale operation is computed with:

$$M_{scale} = \begin{bmatrix} \frac{1}{u^{wp}_x} & 0 & 0 \\ 1 & \frac{1}{|\boldsymbol{v}^{wp}|} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

These matrices can be concatenated into one volume transformation matrix as (using post-multiplication of the volume coordinates):

$$M_{vol} = M_{scale} \cdot M_{shear} \cdot M^{wo}_{view}. \tag{11}$$

The shearing operation distorts the spherical splatting kernels into ellipsoids, but, in contrast to Westover's splatting, the distortion is always coplanar to the image plane, i.e., the kernel ellipsoid's three axes are always in or normal to the image plane. The kernels are now all sliced the same way and, thus, slicing is viewpoint independent. This allows us to use a single footprint hierarchy for all volume orientations. Thus, footprint mapping is performed by first rotating the texture polygons so that they are parallel to the image plane and, then, shearing and scaling them according to $\mathrm{M_{shear}}$ and $\mathrm{M_{scale}}$. Note that, since the warping does not preserve dot products, shading must be performed in the unwarped volume space. In addition, gradients must also be estimated in the unwarped space.

The extension of this technique to perspective rendering is straightforward. We simply add the perspective matrix at the beginning of the matrix chain (11). We may have to rotate the footprint polygon into the ray direction prior to mapping, as is described in [17], to better approximate the diverging path in which the rays traverse the kernel. In addition, after a certain distance from the viewpoint, we will have to keep the size of footprint polygons constant (in perspective space) to compensate for the undersampling of the volume grid by the diverging grid of rays (refer to [17] for more details).

The ellipsoidal kernels in [28] are designed to fill the space in the rectilinear grid that would otherwise be occupied by the extra interpolated slices in the corresponding cubic grid. For summed X-ray rendering, identical images result when the ellipsoidal rectilinear grid splatting kernel equals the kernel resulting from convolving the ellipsoidal slice interpolation kernel with the spherical cubic grid splatting kernal. This is true for either axis-aligned and image-aligned sheet-buffer splatting. However, for a Sabella-model involving compositing, we need to be more careful. It can be shown that our grid warping technique yields the exact same footprint (for a complete kernel) as projecting an ellipsoidal kernel directly, via Westover's

TABLE 1
Data Structures Required for List-Based Splatting

List-Splat Arrays

Array	Array Magnitude	Description
$<value>$	N_{vol}	volume
$<value, index_{volArr}>$	N_{rel}	relevant voxels
$<x, y, z, value>$	N_{iso}	iso-voxel coordinates and values
$<g_x, g_y, g_z>$	N_{iso}	iso-voxel gradients
$<r, g, b, \alpha>$	N_{iso}	iso-voxel colors and opacities
$<z\text{-}distance>$	N_{iso}	iso-voxel screen-distance
$<bucket>$	*TotalNumBuckets*	bucket pool
$<bucket>$	*MaxNumSlices*	slice buckets

Bucket Structure

Element	Element Magnitude	Description
$<indx_{isoArr}>$	*BucketSize*	bucket voxel index array
next	1	pointer to next bucket
count	1	number of bucket voxels

N_{vol} is the size of the volume, N_{rel} is the number of relevant voxels in the volume (usually the nonzero voxels), and N_{iso} is the number of voxels that fall within the variable iso-range(s). A bucket holds an array of indexes into the iso-voxel arrays. Each bucket is assigned to one of the slicing slabs.

method. It can also be shown that both methods yield correct integrations, but there is still one more issue. Let's assume we have a 2D grid in which the y-axis spacing is larger than the x-axis spacing, and a ray, sampling this grid with spacing $\Delta l = (\mathrm{d}x^2 + \mathrm{d}y^2)^{1/2}$. The warping decreases the ray sample distance to $\Delta l^w = (x^2 + (y/ar_y)2)^{1/2}$ in the warped grid. To ensure homogeneous, uniformly spaced compositing for all orientations, we need to decrease the slab width in the warped grid accordingly. This is easily facilitated by using the summed-area footprints described in Section 3.3.1. Note that this solution is unique to the image-aligned splatting method. The traditional axis-aligned method is unable to account for this problem, as here the kernel is projected as a whole and not in sections. As a result, we notice strong color variations in animated viewing of volumes with larger aspect ratios.

5 List-Based Splatting

When exploring a volumetric data set, the user needs to be able to perform the following basic tasks quickly: 1) View the object at different orientations on the screen, and 2) vary the opacity, color, and gradient transfer functions to expose different aspects and structures of the dataset. The former constraint requires a fast volume viewing algorithm, while the latter necessitates a data organization that allows a change in both transfer functions and iso-ranges to come into effect quickly.

We have implemented a framework for splatting that satisfies both of these requirements. Our algorithm requires as the only preprocessing step an initial bucket-tossing of the volume data values. Once the volume data is bucket-tossed into a list, a binary search can be employed to extract the voxels that fall within one or more specified iso-intervals. Then, during rendering, only the extracted voxels are transformed, shaded, and displayed.

The algorithm presented here is a continuation of the work discussed in [3]. The original algorithm, however, capitalized on the fact that all voxels within the iso-range had similar colors and, therefore, no ordered back-to-front (or front-to-back) traversal was required. The new version of this method does not make this assumption. It is therefore suitable for a more general class of volume data sets. The new framework easily allows us to render a volume using both the axis-aligned sheet-buffer method and the new image-aligned sheet-buffer method for comparison purposes.

5.1 Basic Algorithm

Our algorithm employs several arrays, as shown in Table 1. The entire volume (of size N_{vol}) is first read into the $< value >$ array. Then, a $< value, index_{valArr} >$ array is extracted, containing all those voxels in the $< value >$ array that have values meaningful to the user. For instance, in medical applications, a voxel with a zero value is irrelevant and will never have to be projected. Thus, we do not need to store it in the $< value, index_{valArr} >$ array. In many applications, this cuts down the number of relevant voxels, N_{rel}, to 10-20 percent of N_{vol} [30]. Finally, the $< value, index_{valArr} >$ array is sorted by the (value) key. This completes the preprocessing step when the volume is loaded.

The user indicates one or several iso-ranges. A binary search on the $< value, index_{valArr} >$ array is used to determine the voxels at the iso-range boundaries. All voxels within the iso-range boundaries are copied into the $< x, y, z, value >$ array, where (x, y, z) is the object space coordinate of a voxel and (*value*) is its value. Using an optional transfer function lookup table, the $< g_x, g_y, g_z >$ gradient array is then computed. The $(index_{valArr})$ entry of the $< value, index_{valArr} >$ array is used to access the voxel neighbors in the $< value >$ array. Finally, we compute the values in the $< r, g, b, \alpha >$ array, employing a diffuse and specular shading model in conjunction with the gradient array and optional transfer function lookup tables. This last step is view-dependent.

For each view, we need to determine the range of sheet-buffers that each voxel kernel falls into. We do not require the depth-order within a sheet-buffer since the adding operation is commutative. Thus, it is sufficient to associate a bucket with each sheet-buffer that holds the intersected kernel sections for the current viewpoint. For this purpose, we define a bucket structure (Table 1, bottom portion) that keeps an array of indices $< indx_{isoArr} >$ into the iso-voxel arrays, along with an associated count and a pointer to the next bucket. This pointer is used as a link to a new bucket, should the current bucket exceed its capacity. Since it would be inefficient to dimension the bucket index arrays for the largest anticipated voxel load, we provide a pool of smaller buckets from which buckets are retrieved and linked whenever a slice bucket exhausts its limits.

The bucket toss procedure is straightforward. Each voxel in the iso-list is transformed, yielding its z-distance from the screen. Then, based on this z-distance, the voxel's index is added to the range of buckets it overlaps. Additionally, the z-distance is also written into the $< z - distance >$ array. Once all voxel slices have been assigned to buckets, they can be splatted to the screen, in front-to-back or back-to-front order.

This list data structure allows rapid changes in the transfer functions for gradients, opacities, and colors, as well as light source locations (if the viewpoint and iso-ranges remain constant). All that needs to be recomputed is the shaded value. No new bucket-tossing is required. Should the iso-value intervals change, then a new fast binary search fills in the data structures.

For typical volumes, the only large data structure in this framework is the $< value >$ array, which holds the entire volume. All other arrays hold typically 20 percent or less of N_{vol}. We can eliminate the $< x, y, z, value >$ array altogether by indexing the $< value, index_{volArr} >$ array directly, using the index of the first iso-voxel(s) in the iso-range(s) as an offset. When the volume has dimensions of powers of two, then the (x, y, z) values can be quickly computed via binary shifts of the $(index_{volArr})$ field. This saves the transfer of values into the $< value, index_{volArr} >$ array when the iso-range changes. If the user alters the iso-range smoothly or incrementally, then we may use an incremental update of the iso-data structures, such as the gradients, shaded colors, and z-distances. In this lazy-evaluation scheme, we only have to compute gradients, colors, and z-distances due to new voxels that have entered the active interval.

5.2 Implementation of Axis-Aligned Sheet-Buffer Based Splatting Using List-Splats

For the axis-aligned sheet-buffer method, it is most efficient to make up three bucket-arrays, one for each axis direction. Since the sheet-buffers are always aligned with one of the volume axes, no bucket-tossing is required for new view-points as long as the iso-intervals remain constant. The $indx_{isoArr}$ field points into the gradient, color, and coordinate arrays. To render an image, the appropriate bucket array is accessed and traversed up or down, depending on the viewing direction along the viewing axis and also depending on what volume traversal order was chosen: back-to-front or front-to-back.

6 Accelerated Splatting Using Early Splat Elimination

So far, all voxels in the iso-range(s) have to be projected. This is unnecessary since it is very likely that many voxels are occluded by other voxels located closer to the screen. To limit our rendering effort to just the visible voxels we would like to ensure a voxel splat's partial visibility before we send it down the rendering pipeline. In this section, we will describe an efficient technique that allows us to do this. Our scheme has a similar effect as early ray termination in ray-based approaches, hence the name *early splat elimination*.

Visibility culling to accelerate volume rendering applications has been used for quite some time. Reynolds et al. used a dynamic screen technique [20] in which, similar to the shear-warp algorithm [10], the object and image are run-length encoded, with the latter dynamically changing. This enables the renderer to quickly skip across opaque image scan line segments as well as empty object scan line segments. In both approaches, however, a voxel is considered a point that projects exactly onto one image row and pixel, which allows easy access and maintenance of the image RLE data structure. In splatting, on the other hand, a voxel footprint potentially covers an extended 2D area of pixels (Fig. 10). This makes interaction with an RLE structure difficult. In addition, the clever encoding scheme of both dynamic screen and shear-warp requires three volume copies and does not allow for easy modification of the transfer functions or lighting. Finally, Meagher [15] represents the object by an octree, while the image is encoded into a dynamic quadtree. Again, upkeep of the quadtree structure with footprints is difficult.

A splat has only an effect on the image if at least a partial area within its footprint has not yet been rendered opaque in the opacity buffer. (The opacity buffer is the opacity channel of the compositing buffer, which is updated every time a sheet-buffer is composited.) A naive algorithm

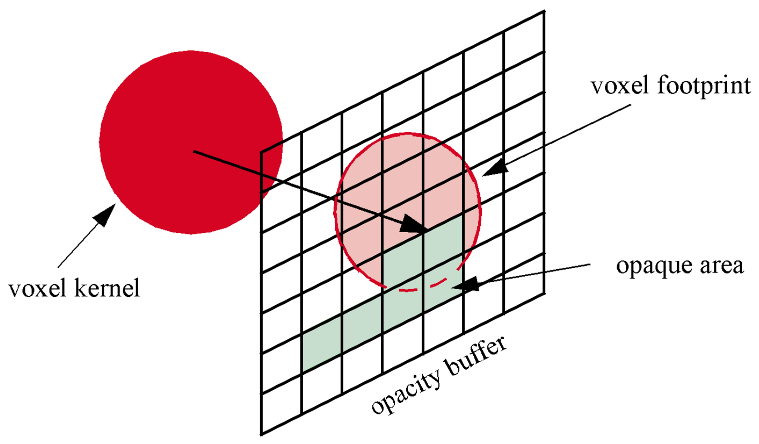


Fig. 10. Naive voxel occlusion test: All pixels in the opacity buffer that fall within the footprint's extent must be tested for opaqueness. The voxel is culled only when all pixels within the footprint extent are opaque.

would project the footprint's bounding box to the screen and check if the opacity of any of the pixels inside the projected box is below 1.0, in which case the footprint would be rendered (see Fig. 10). However, this procedure would be nearly as expensive as the rasterization of the footprint itself. Consider that a footprint's projection area is potentially quite large: about $5 \cdot 5 = 25$ pixels when the image and volume have the same resolution, and even larger for close-up views. Using a dynamic quadtree (similar to Meagher [15]), an update to the quadtree structure would be required for every footprint projected. This is also costly since many quadtree nodes may be affected by a single footprint.

An alternative and beneficial method would just perform a single test with the projected voxel center. This test, however, yields a potentially inaccurate result. This is shown in Fig. 11a for two representative voxels: Although the centers of both of these voxels project into opaque screen regions, only one of them is fully occluded and can be culled. We can devise a more effective scheme, outlined in Figs. 11b and c: The condition for a voxel to be fully occluded is that all pixels within its footprint bounding box must have opacity = 1.0. This means that the opacity average of all these pixels must be 1.0 as well. If the opacity average is less than 1.0, then there has to be at least one pixel in the bounding box that has an opacity of less than 1.0. Since the dimensions of the footprint bounding box are identical for all voxels that fall into a sheet-buffer, we can calculate an average opacity buffer by convolving the opacity buffer with a box filter of the size of a footprint bounding box. Each pixel in this average opacity buffer (or occlusion buffer) then represents the average opacity of all pixels within a neighborhood of the dimensions of the footprint bounding box. Thus, when indexed with the projected voxel center, the occlusion buffer will enable us to decide, with a simple comparison, if a voxel is fully occluded and can be culled or if it is only partially occluded and must be projected.

We are not restricted to using an opacity of 1.0 as the threshold at which voxels are culled. As a matter of fact, a very effective way to accelerate ray-based methods is to lower the opacity threshold at which the ray is terminated. This is commonly known as α-acceleration [4]. The quality of the images obtained with lower opacity thresholds is not much degraded, since the colors accumulated at large ray opacities are highly attenuated and are not very noticeable in the final image. For lower thresholds or α-acceleration to be used in our splatting algorithm, we must restrict the maximal opacities obtainable during compositing to the desired cut-off value. Then, the convolved opacity buffer will yield the correct occlusion buffer for that opacity threshold.

Most graphics boards support convolution in hardware since it is a frequently used operation for multimedia image processing. Moreover, it is a relatively inexpensive operation, especially when the convolution filter is separable (which is the case for a box filter). However, it still puts a burden on the system and we only want to perform opacity buffer convolution operations in image areas that received actual splat contributions in the current sheet buffer. The same is true for the compositing operations. Since we have to transform each voxel into screen coordinates anyhow (to index the occlusion map), we can maintain a dynamic data structure that keeps track of the spatial splat projection statistics while splats are added to the sheet buffer. The simplest form of such a data structure is a bounding box which contains the centers of all voxels that have been projected into the current sheet-buffer. We then only convolve the image areas within a slightly larger bounding box (to account for the extent of the footprint bounding box). While a simple bounding box may work well for localized splats projection statistics, for more dispersed objects we may want to partition the screen into a number of small tiles, say of size 32×32 pixels. Then, whenever a voxel center is projected into a tile, a counter associated with the tile is incremented. If both tile and image sizes are a power of two, we can perform tile indexing very efficiently via simple bit-shifts. Compositing and convolution operations are then only performed within the (expanded) tiles, when the counter exceeds a necessary (but not sufficient) threshold for the entire tile to opaque. The pseudocode for our complete splatting algorithm is provided in Fig. 12.

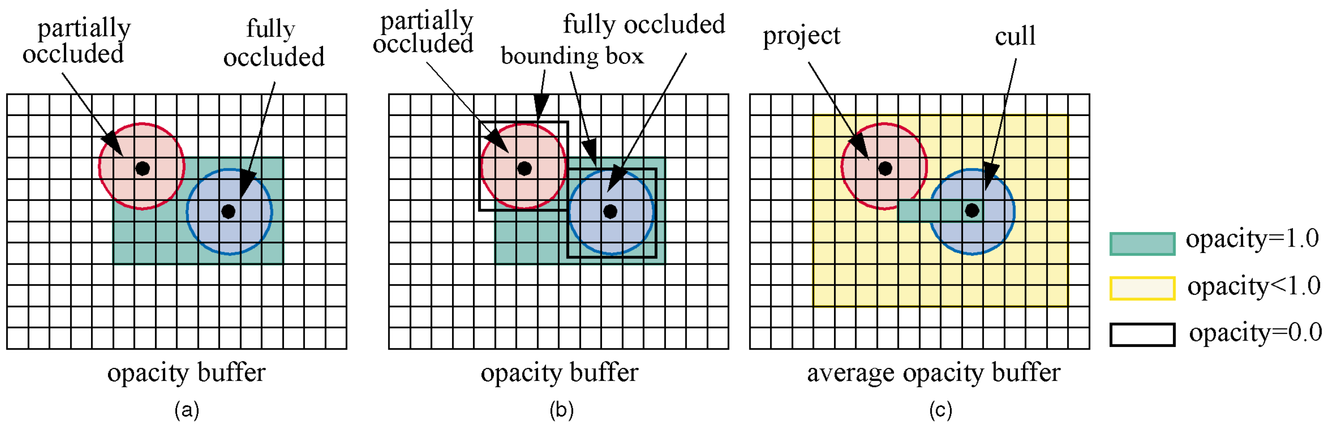


Fig. 11. (a) Using the projected voxel center to index the opacity buffer does not distinguish between the fully occluded and the partially occluded voxel. (b) Only when all pixels within the footprint bounding box have opacity = 1.0, the voxel is fully occluded. (c) If all pixels within a footprint's bounding box have opacity = 1.0, then the average opacity of these pixels is also 1.0. We can thus compute an average opacity buffer that stores the average opacity within a neighborhood of the size of the footprint's bounding box at each pixel location. This average opacity buffer is then indexed by the projected voxel center to determine full occlusion and henceforth decide culling.

Note that we can accelerate the axis-aligned splatting algorithm in a similar manner. Hence, our occlusion culling algorithm for splatting works for both image-aligned and axis-aligned splatting approaches.

The size of the box filter depends upon the volume zoom factor. Thus, the box filter must be redefined for each new viewpoint, which is easy to do. For volume grids with nonunit aspect ratios, the convolution filter has the shape of a parallelogram. We can either implement this filter as a nonseparable convolution, which is more expensive, or we can approximate the parallelogram by a (separable) rectangle (see Fig. 13). This introduces slight inaccuracies which, in practice, do not lead to visible artifacts. For perspective viewing, we need to redefine the convolution filter size for every slice, depending on the perspective magnification and kernel size used in the next slicing slab to be rendered. If we perform the perspective transformation before mapping the kernel footprints, then all voxels that are located beyond the distance at which the perspective grid rate falls below that of the volume grid have equal-sized footprints. Before this distance, the footprints become progressively smaller with decreasing distance from the screen [17].

7 Results

We have tested the various aspects of our algorithm as well as sample implementations of axis-aligned sheet-buffered splatting, both in back-to-front and front-to-back volume traversal, on three volumetric datasets:

- The UNC MRI head ($256 \times 256 \times 163$ voxels, aspect ratio $1 \times 1 \times 1$). We used the LEGION algorithm [22] to segment this dataset into skin, eyes, and brain matter.
- A ganglion nerve cell ($512 \times 512 \times 76$ voxels), acquired with a confocal microscope. This volume has a highly skewed aspect ratio of $1 \times 1 \times 5$, which is typical for this kind of dataset.
- The MRI tomato from Lawrence Berkeley Lab [12] ($256 \times 256 \times 64$ voxels and an aspect ratio of $1 \times 1 \times 3$). This dataset has been segmented by LBL into five components: the core, the endocarp, the locule, the placenta, and the seeds.

For all our experiments, unless stated otherwise, we have used a Gaussian splatting kernel with a radial extent of 2.0 for all splatting methods.

First, we demonstrate the effectiveness of the new image-aligned splatting algorithm with respect to eliminating the popping artifacts of traditional axis-aligned splatting. Consider Fig. 14, where we show a three-frame sequence of an animation around the nerve dataset. The orientation angles at which the frames were taken are: 41°, 44°, and 47°. A headlight is used as the only lighting source. The top row shows images obtained with the axis-aligned method. The sheet-buffer orientation switched between 44° and 47°, and the difference of object brightness in these two frames is severe. Recall that the nerve volume has an aspect ratio $(ar_x, ar_y, ar_z) = (1, 1, 5)$, giving rise to very elongated ellipsoidal kernels. In the first two frames, the compositing occurs along the z-axis. We can actually see the shape of the elongated kernels in these frames. The very high brightness results from compositing many highly opaque, long kernels onto a small image area all at once, and not in sections (see our comments at the end of Section 4). In the third frame, the kernels are composited along the x-axis (along their short axis), which reduces the image brightness. The sudden pop in brightness when this transition occurs is very noticeable in animated viewing. In contrast, the new image-aligned method (bottom row) with proper sectioning and compositing of the kernels avoids these problems and provides for a pop-free animation across the entire angular range.

We will now compare the various splatting algorithms in terms of their computational effort. In order to investigate the run-time behavior for a variety of scenarios, we have rendered our three test objects using different transfer functions. The image in row 1, column 1 of Fig. 16 shows the MRI head dataset, rendered as an opaque iso-surface, Fig. 15a shows the same dataset rendered semitransparent with glass-like specular reflections. Fig. 15b shows the

```
Clear(compositingBuffer);
for i=startSlice....endSlice
  Clear(sheetBuffer);
  Clear(tiles);
  do
     currentBucket=sliceBucket[i];
     for j=1...currentBucket.count
        indx=currentBucket.indxArr[j];
        x=X-transform(isoVoxelArr[indx].coordinates);
        y=Y-transform(isoVoxelArr[indx].coordinates);
        if occlusionBuffer(x,y)<opacityThreshold
           if isoVoxelColor[indx] not computed
              isoVoxelColorOpacity[indx] =
                       Shade(x, y, z-distance[indx], isoVoxelGradientArr[indx]);
           ProjectSplat(x, y, isoVoxelColorOpacity[indx], sheetBuffer);
           IncrementTileCount(x, y, tiles);
     if currentBucket.next not equal Nil
        currentBucket=currentBucket.next;
     else
        break;
  Composite(sheetBuffer, compositingBuffer, opacityThreshold, tiles);
  occlusionBuffer=
      Convolve(compositingBuffer.opacityImage, footprintBoxFilter, tiles);
Display(compositingBuffer);
```

Fig. 12. Pseudocode that demonstrates list-based splatting with slice buckets and splat visibility test.

tomato dataset, rendered semitransparent with the core, placenta, and seeds showing through. Finally, Fig. 15c shows the nerve dataset in a head-on view down the z-axis, while the image in row 3, colum 3 of Fig. 16 shows a side view of the nerve, as seen down the x-axis. Table 2 lists the rendering times for these images. To generate this table, we have chosen a slab width of 1.0 and an opacity threshold of 0.95, where applicable. The timings given were obtained on an SGI Octane with a R10000 processor and 640MB of memory. The resident 2D texture mapping hardware was used to rasterize the footprints and the graphics hardware was used to perform the convolution operations. However, we have also recently implemented a pure-software version of our algorithms to run on a 400MHz Pentium II with 256MB of memory. The timings obtained on this machine were roughly comparable with the ones obtained with SGI graphics hardware support.

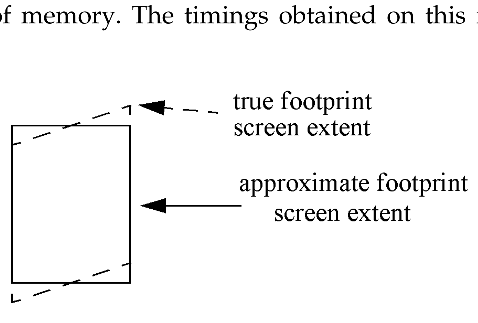

true footprint screen extent
approximate footprint screen extent

Fig. 13. True footprint screen extent when the grid aspect ratio is nonunity vs. approximate footprint screen extent. The former gives rise to a nonseparable convolution filter, while the latter can use a cheaper separable convolution filter.

The first columns of Table 2 list the results obtained with back-to-front traversal of the volume. We observe that, for both sheet-buffered splatting methods, the time for back-to-front rendering is highest, as here all voxels in the iso-interval must be splatted. However, back-to-front rendering with image-aligned splatting is generally at least twice as expensive than with axis-aligned splatting, as indicated by the ratio term in this section of Table 2. This is simply because each voxel gives rise to three to four footprints that need to be rasterized. Note that the solid head takes longer since it contains all voxels above the skin iso-value, while the semitransparent head contains only the voxels for the skin, the eyes, and the brain.

Let us now move to the front-to-back section of Table 2. We have advocated front-to-back splatting in this paper since, in this traversal order, we can take advantage of early splat elimination, i.e., visibility acceleration, as introduced in Section 6. Note that the rendering times for front-to-back splatting without visibility acceleration are identical with those of back-to-front splatting and are therefore not listed in Table 2. We observe in the first column of this table section that, in most cases, visibility acceleration indeed reduces the rendering times for the image-aligned splatting method, which is a direct result of the reduced number of voxels that need to be splatted. This is true even for the semitransparent views. We also see, however, that, for the side-view of the nerve dataset, the rendering time actually increased. Apparently, the overhead for convolution domi-

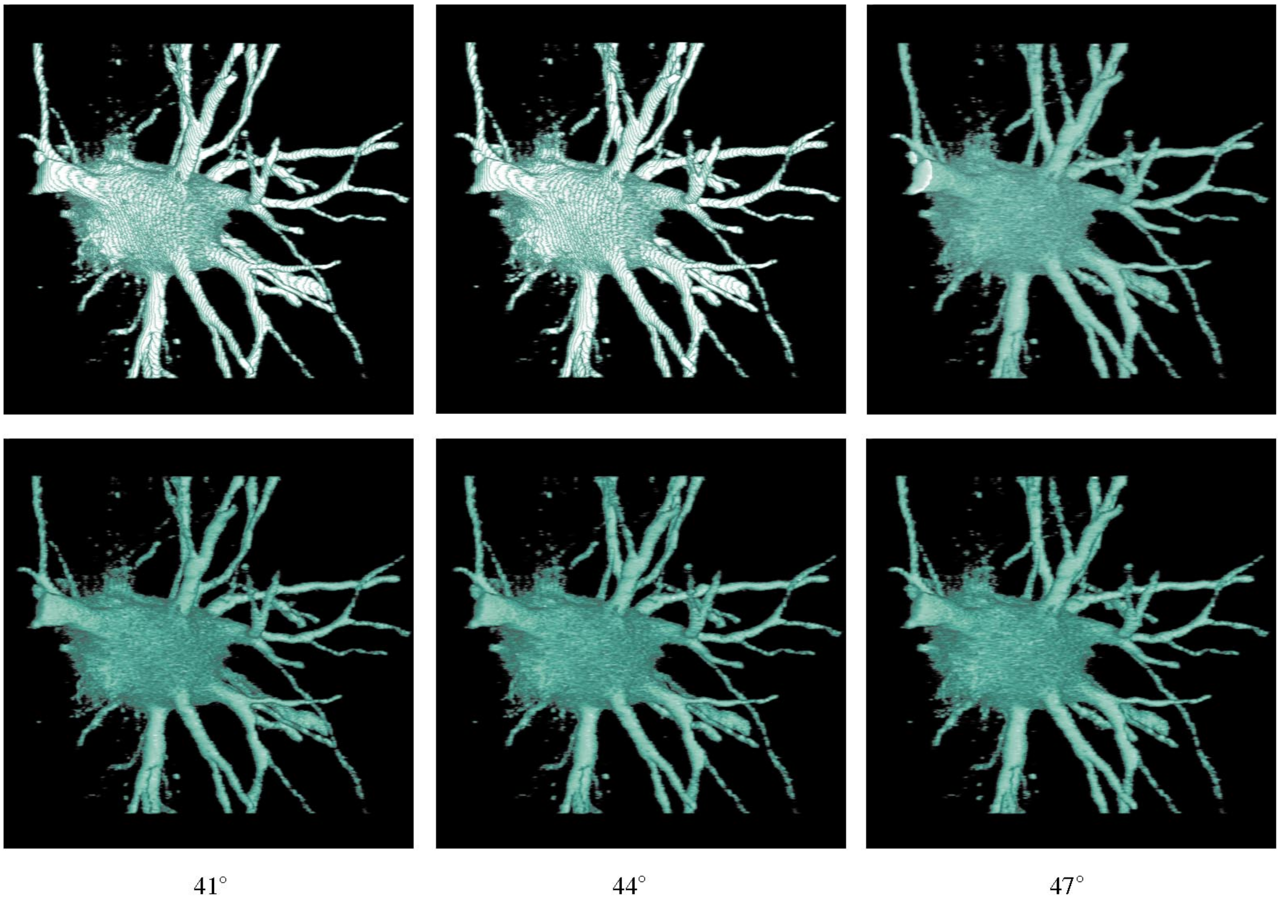


Fig. 14. Top row: nerve dataset rendered with axis-aligned sheet buffer-based splatting. Bottom row: nerve dataset rendered with image-aligned sheet buffer-based splatting. Noticeable popping artifacts are observable with the axis-aligned method, while the image-aligned method provides for a smooth animation.

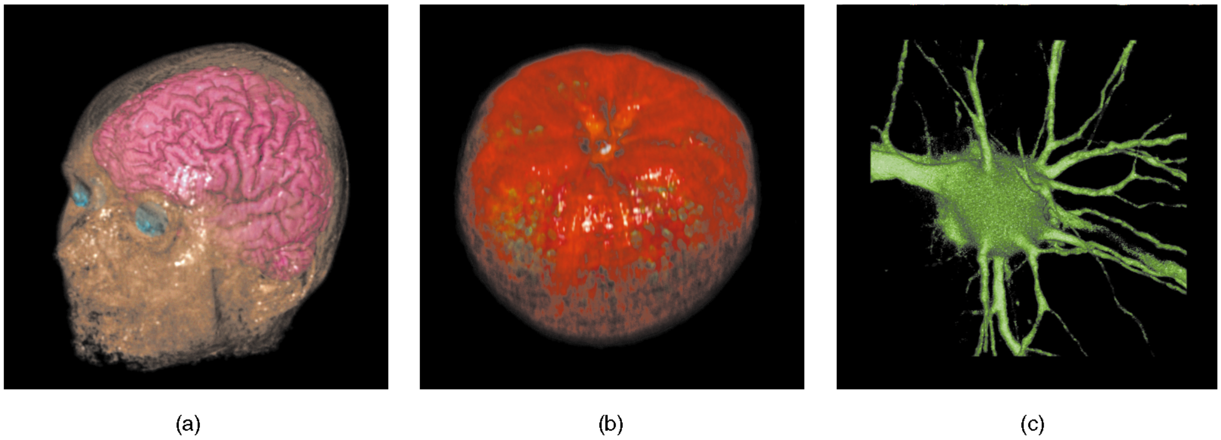


Fig. 15. (a) MRI dataset, rendered semi-transparent with glass-like specular reflections, (b) MRI dataset, rendered semitransparent with eyes and brain showing through, (c) tomato, rendered semitransparent with seeds, core, and placenta showing through, (d) head-on view of the nerve dataset.

nated the savings obtained through voxel culling. This is mainly due to the fact that the iso-voxels in the nerve dataset are relatively sparse and dispersed, but nevertheless occupy a large volume, and the convolution is mostly wasted on empty opacity buffer areas. By tiling the opacity buffer, we can limit the convolution (and compositing) efforts to the relevant buffer regions and reduce the rendering times significantly. This is documented in the third column of this section of Table 2. We can apply the tiled visibility acceleration to axis-aligned sheet-buffered splatting as well and speed-ups between 1.4 and 2.2 are obtained. Finally, comparing the rendering times for fully accelerated axis-aligned and image-aligned front-to-back splatting, we observe that the runtime ratios have dropped significantly. Image-aligned splatting is now only about

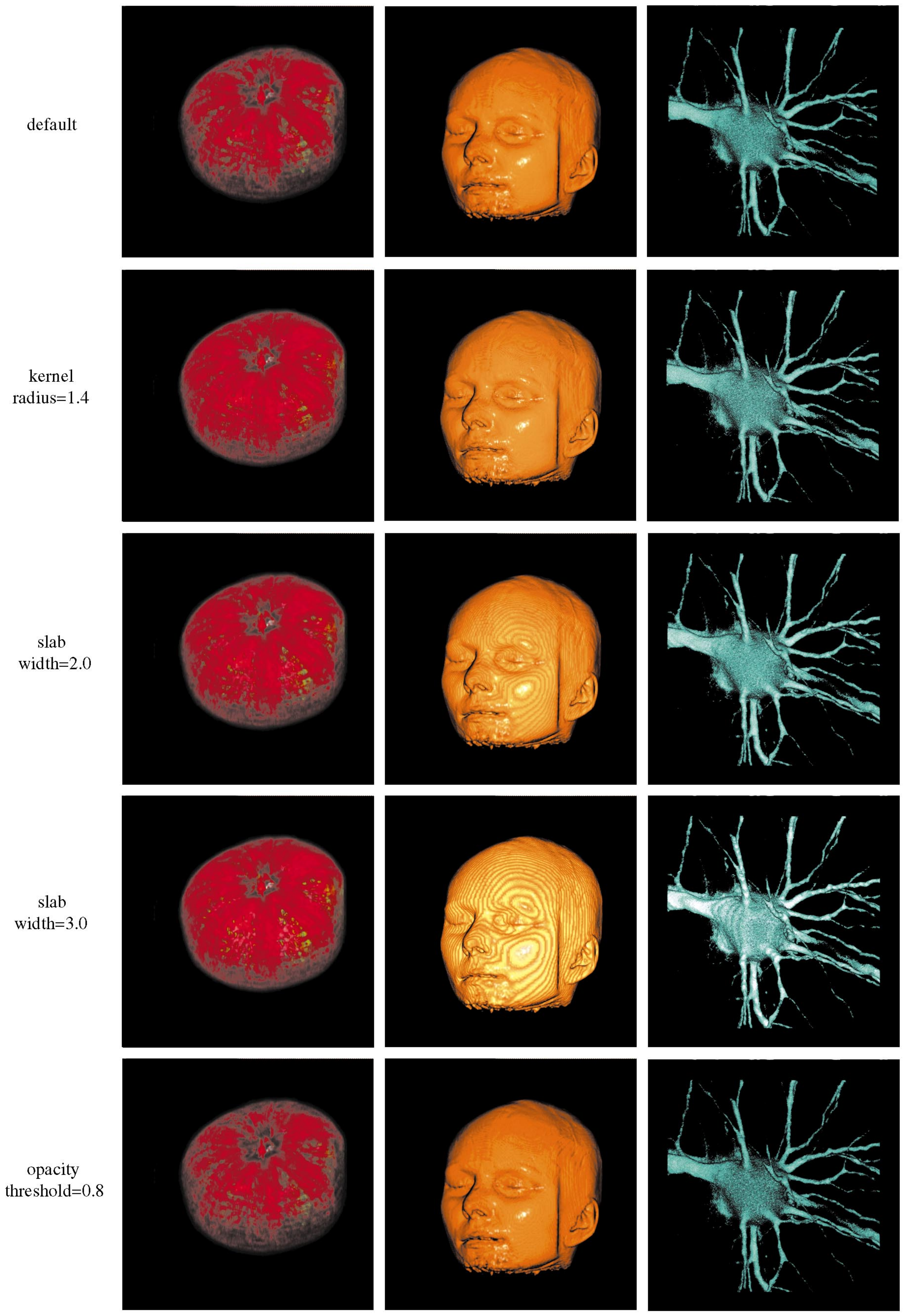


Fig. 16. An array of images which show the effects acceleration methods have on image quality.

TABLE 2
Rendering Times (in secs) for Various Incarnations of Sheet Buffer-Based Splatting Algorithms

rendering order	back-to-front			front-to-back			
sheet-buffer	img.-align.	axis-align.	ratio	img.-align.	img.-align.	axis-align.	ratio
acceleration	-	-		+ vis. test	+ vis. test + tiles	+ vis. test + tiles	
opaque head	54.6	22.5	2.4	21.2	12.7	10.2	1.2
transp. head	35.8	15.8	2.2	22.5	14.8	9.7	1.5
nerve side-view	26.0	14.3	1.8	33.1	12.5	9.8	1.3
nerve head-on	17.6	9.0	1.9	11.3	8.4	4.7	1.8
tomato	36.4	14.9	2.4	23.8	15.5	10.3	1.5

Each ratio term refers to the two columns immediately preceding it. The timings were obtained on an SGI Octane with R10000 CPU and 640MB of memory, using 2D texture mapping hardware for footprint rasterization and opacity buffer convolution. However, similar timings were also obtained with a pure software implementation on a 400MHz Pentium II with 256MB of memory.

TABLE 3
Runtimes (in secs) and Speedups (in Parentheses) for Various Methods Used to Accelerate the Rendering Time of the Sheet Buffered Splatting Methods

method / dataset	default	kernel radius=1.4	slab width=2.0	slab width=3.0	opacity threshold=0.8
opaque head	12.7	10.5 (1.2)	8.1 (1.5)	6.4 (2.0)	11.8 (1.1)
nerve head-on	8.4	6.9 (1.2)	5.3 (1.6)	4.5 (1.9)	7.2 (1.2)
tomato	15.5	12.6 (1.2)	10.1 (1.5)	8.2 (1.9)	13.1 (1.2)

Varying the slab width is unique to the image-aligned method. The default case is kernel radius = 2.0, slab width = 1.0, and opacity threshold = 0.95.

50 percent more expensive than axis-aligned splatting with the benefit of much improved image quality.

Our framework offers a number of easy acceleration techniques, partially unique to the image-aligned sheet-buffer approach, that reduce rendering times even more. For instance, using smaller kernels, such as a Gaussian with radial extent of 1.4, reduces the number of section footprints to be rasterized per voxel. Making the kernel sections wider, i.e., increasing the width of the sheet-buffer slabs, cuts the number of kernel sections as well. It, however, also increases the volume sampling interval which potentially leads to aliasing-related artifacts. Finally, we may also lower the opacity threshold for early splat elimination, which is equivalent to α-acceleration in conjunction with early ray termination [4].

Fig. 16 presents an array of images which show the effects these acceleration methods have on image quality, while Table 3 presents the savings in rendering time. To demonstrate our findings, we have selected three representative datasets and transfer functions: the semitransparent rendering of the tomato, the opaque head, and the head-on view at the nerve cell. The images reveal that the quality of the semitransparent rendering of the tomato does not suffer much from any of the acceleration methods, even when a slab width of 3.0 is used. Thus, it appears that we can significantly speed-up the rendering, especially when the acceleration methods are combined. On the other hand, an opaque dataset, such as the head, appears to be very sensitive to slab width—strong aliasing artifacts appear, even for a moderate slab width of 2.0. Lowering the opacity threshold or the kernel radius has only minor or no adverse

TABLE 4
Various Splat Statistics

	total # voxels	# voxels in iso-range		# shaded voxels		total # kernel sections	# rasterized kernel sections	
opaque head	10.6 M	2.2 M	20 %	118 k	18%	11.1 M	321 k	3%
transp. head	10.6 M	1.4 M	13%	413 k	29%	7.1 M	1.5 M	21%
nerve side-view	19.9 M	1.0 M	5%	114 k	11%	4.0 M	331 k	8%
nerve head-on	19.9 M	1.0 M	5%	301 k	30%	4.0 M	923 k	23%
tomato	4.2 M	1.3 M	30%	672 k	51%	6.7 M	2.5 M	37%

The percentages given in the second column of one category refer to the quantities given in the category immediately to the left.

effects on image quality for these datasets and views. The nerve dataset is opaque as well, but more irregular and dispersed in structure than the head. An increase in slab width to 2.0 is well tolerated in this dataset, as well as using a smaller kernel or lowering the opacity threshold. Hence, we recognize that the utility of these speed-promoting enhancements are dependent on the underlying dataset and transfer function. Lowering the opacity threshold and using smaller kernels is generally well-tolerated, while varying the slab width represents a more sensitive issue.

The time spent on rasterization is directly proportional to the number of voxel sections surviving the early splat elimination test. The time spent on shading is somewhat proportional to this number, but could be slightly higher since not all kernel sections may be eliminated and thus the kernel must be shaded for the surviving (front-most) sections of the kernel. (Note that shading is done only once per kernel.) To account for this dependency, and also to illustrate the considerable savings that come with early splat elimination, we have listed some splat statistics in Table 4. We see that, for the opaque objects, the number of splats to be rasterized can be culled by more than 90 percent in some cases. For the semitransparent objects, more than two thirds of the voxels are culled. Similar relationships hold for the number of shaded voxels in relation to the number of voxels in the iso-range.

For the image-aligned splatting approach, extracting a new iso-range from the iso-voxel list usually takes less than 0.2 seconds using binary search. In the axis-aligned approach, we use the technique proposed by Ihm and Lee [8], sorting each volume slice by voxel value. In this way, we find the iso-voxels quickly by using an inexpensive binary search in each slice.

8 Conclusions

We have presented a new splatting technique that rectifies one of the most notorious disadvantages of splatting: the inseparability of volume integration and compositing. In contrast to traditional sheet-buffered splatting, the new method does not add voxel kernels as a whole to a volume axis-aligned sheet-buffer. Rather, it sections the voxel kernels by pairs of image-parallel slicing planes and adds all kernel slices that fall between such a pair of slicing planes to the associated image-aligned sheet-buffer. The sheet-buffers themselves are composited as usual, but the compositing now occurs orthogonal to the image plane and is separated from the kernel integration. This brings the splatting approach more in line with raycasting, but retains its original advantages: the sparse volume representation and the efficient volume projection via footprint lookup tables. The new method does not suffer from the disturbing popping artifacts that occur in animated viewing with traditional sheet-buffered splatting and also generates images of better quality. In addition, by varying the distance between slicing planes, we are given an easy means for controlling the trade-off of rendering speed vs. rendering quality.

We then extended the algorithm to volumes with anisotropic grid spacing, which often occur in medical datasets. We proposed a new grid warping technique that does not require the usual ellipsoidal kernels, and thus avoids the artifacts obtained with those at no extra cost. Our method can account for the view-dependent ray sampling rate that results from splatting an anisotropic grid without interpolating additional slices. We can adjust for this circumstance by slicing the kernels into thinner sections, using a 2D array of summed-area tables from which any kernel section integral can be efficiently retrieved.

Finally, we have extended the principles of raycasting's early ray termination to splatting. After each sheet-buffer compositing, an occlusion buffer is computed from the opacity buffer via a convolution operation. The occlusion buffer can then be efficiently indexed by a projected voxel center to determine the voxel's (partial) visibility. A pre-set opacity threshold determines how early splats are culled from the rasterization pipeline, which is a scheme similar to α-acceleration used in raycasting. Early splat elimination culls up to 97 percent of voxels from rasterization. By limiting the convolution and compositing operations to those image tiles that have received contributions in the current sheet-buffer, we can limit the computational overhead of the method significantly.

Future work will investigate speedups obtained when projecting all voxels with low opacities as full kernels, since these voxels do not contribute much to the image anyhow. We will also investigate schemes that use the value of the opacity image to determine how much the splat will potentially contribute to the final image. If the opacity is already large, we could simply project the entire voxel at once instead of splatting it multiple times as kernel slices (this is equivalent to the β-acceleration in [4]). Furthermore, if a voxel opacity is low, we could decrease the relevant splat size to cull opacity-multiplied splat contributions that are below the resolution of the sheet and accumulation buffer.

Acknowledgments

We would like to thank Nelson Max for helpful hints and comments. Further, we would also like to thank Don Stredney and Dennis Sessanna from the Ohio Supercomputer Center for hardware support and maintenance. This work was supported in part by the U.S. Department of Engery and the National Library of Medicine.

References

[1] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *Proc. 1994 Symp. Volume Visualization,* pp. 91-98, 1994.

[2] R. Crawfis and N. Max, "Textured Splats for 3D Scalar and Vector Field Visualization," *Proc. Visualization '93,* pp. 261-266, 1993.

[3] R. Crawfis, "Real-Time Slicing of Data-Space," *Proc. Visualization '96,* pp. 271-279, 1996.

[4] J. Danskin and P. Hanrahan, "Fast Algorithms for Volume Raytracing," *Proc. 1992 Workshop Volume Visualization,* pp. 91-98, 1992.

[5] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Proc. SIGGRAPH '88,* pp. 51-58, 1988.

[6] T. He, S. Wang, and A. Kaufman, "Wavelet-Based Volume Morphing," *Proc. Visualization '94,* pp. 85-92, 1994.

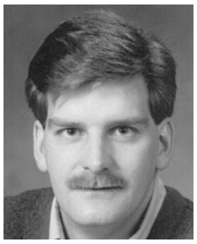
[7] G.T. Herman, J. Zheng, and C.A. Bucholtz, "Shape-Based Interpolation," *IEEE Computer Graphics and Applications,* vol. 12, no. 3, pp. 69-79, 1992.

[8] I. Ihm and R.K. Lee, "On Enhancing the Speed of Splatting," *Proc. Visualization '95,* pp. 69-76, 1995.
[9] J.T. Kajiya and B.P. Von Herzen, "Ray Tracing Volume Densities," *Proc. SIGGRAPH '84,* pp. 165-174, 1984.
[10] P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation," *Proc. SIGGRAPH '94,* pp. 451-458, 1994.
[11] D. Laur and P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *Computer Graphics,* vol. 25, no. 4, pp. 285-288, 1991.
[12] Lawrence Berkely Lab, "Whole Frog Project," http://www.itg.lbl.gov/, 1994.
[13] R. Machiraju and R. Yagel, "Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors," *Proc. SUPERCOMPUTING '93,* pp. 699-708, 1993.
[14] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics,* vol. 1, no. 2, 1995.
[15] D. Meagher, "Efficient Synthetic Image Generation of Arbitrary 3D Objects," *Proc. IEEE CS Conf. Pattern Recognition and Image Processing,* pp. 473-478, June 1982.
[16] K. Mueller and R. Yagel, "Fast Perspective Volume Rendering with Splatting by Using a Ray-Driven Approach," *Proc. Visualization '96,* pp. 65-72, 1996.
[17] K. Mueller, T. Moeller, J.E. Swan, R. Crawfis, N. Shareef, and R. Yagel, "Splatting Errors and Anti-Aliasing," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 2, pp. 178-191, 1998.
[18] K. Mueller and R. Crawfis, "Eliminating Popping Artifacts in Sheet Buffer-Based Splatting," *Proc. Visualization '98,* pp. 239-245, 1998.
[19] T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics (SIGGRAPH '84 Proc.),* pp. 253-259, 1984.
[20] R.A. Reynolds, D. Gordon, and L. Chen, "A Dynamic Screen Technique for Shaded Graphics Display of Slice-Represented Objects," *Computer Vision, Graphics, and Image Processing,* no. 38, pp. 275-298, 1987.
[21] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Proc. SIGGRAPH '88,* pp. 51-61, 1988.
[22] N. Shareef, D.L. Wang, and R. Yagel, "Segmentation of Medical Images Using LEGION," *IEEE Trans. Medical Imaging,* to appear, 1999.
[23] L.M. Sobierajski and R.S. Avila, "Hardware Acceleration for Volumetric Ray Tracing," *Proc. Visualization '95,* pp. 27-34, 1995.
[24] M. Teschner, C. Henn, and M.E. Mueller, "Texture Mapping in Technical, Scientific and Engineering Visualization," SGI report available at www.sgi.com/chembio/resources/texture/, 1997.
[25] A. Van Gelder and K. Kim, "Direct Volume Rendering via 3D Texture Mapping Hardware," *Proc. 1996 Volume Rendering Symp.,* pp. 23-30, 1996.
[26] S. Wang and A. Kaufman, "Volume Sculpting," *Proc. 1995 Symp. Interactive 3D Graphics,* pp. 151-159, 1995.
[27] L. Westover, "Interactive Volume Rendering," *Proc. 1989 Chapel Hill Volume Visualization Workshop,* pp. 9-16, 1989.
[28] L. Westover, "Footprint Evaluation for Volume Rendering," *Computer Graphics SIGGRAPH '90,* vol. 24, no. 4, pp. 367-376, 1990.
[29] L. Westover, "SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm," PhD Dissertation, Univ. of North Carolina-Chapel Hill, 1991.
[30] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics (SIGGRAPH '91 Proc.),* pp. 275-284, 1991.
[31] R. Yagel and Z. Shi, "Accelerating Volume Animation by Space-Leaping," *Proc. Visualization '93,* pp. 63-69, 1993.

Klaus Mueller received a BS degree in electrical engineering from the Polytechnic University of Ulm, Germany, in 1987, then joined The Ohio State University, where he received an MS degree in biomedical engineering in 1990 and a PhD in computer and information science in 1998. He currently holds a post-doctoral research position at the same institution. Dr. Mueller's current research interests reside in computer graphics, in particular, high quality and interactive volume visualization, as well as in cone-beam computed tomography, where he works on the advancement of algebraic methods, such as ART, for cardiac imaging. Another avenue of Dr. Mueller's research is the design of effective PACS systems for medical image data management. For more information, see http://www.cis.ohio-state.edu/~mueller.

Naeem Shareef is a graduate student pursuing a PhD in computer and information science at The Ohio State University. He received a BS in applied mathematics and computer science from Carnegie Mellon University in 1990, and his MS in computer and information science from The Ohio State University in 1992. His research interests include volume graphics, scientific visualization, image segmentation using neural networks, visualization of large datasets, medical applications for volumes, and voxelization. He has held a graduate research appointment at the Ohio Supercomputer Center since 1993 and has received funding from The Ohio State University Cancer Hospital Research Institute and Lockheed Martin. Current funding is provided by a grant from the National Library of Medicine. His publications include work in voxelization using CSG, image and volume segmentation using a neural network, anti-aliasing issues in splatting, and volume visualization in the medical area. For more information, see: http://www.cis.ohio-state.edu/~shareef.

Jian Huang is a PhD student in the Department of Computer and Information Sciences at The Ohio State University, Columbus, where he also holds a graduate research associateship in the Volume Graphics Research Group. His research interests are computer graphics and visualization, networking and multimedia, high performance parallel systems. His research has been funded by grants from ASCI, NLM, and Ford. He obtained his BEng from the Nanjing University of Posts and Telecom, Peoples Republic of China in June 1996. In June 1998, he was awarded an MS degree in computer science and an MS degree in biomedical engineering, both from The Ohio State University. In the summer of 1998, he interned at Ford Motor Company. For more information, see http://www.cis.ohio-state.edu/~huangj.



Roger Crawfis received his BS in computer science and applied mathematics from Purdue University in 1984 and his MS and PhD in computer science from the University of California, Davis, in 1989 and 1995, respectively. He is an assistant professor at The Ohio State University. His research interests include scientific visualization, computer graphics, and volume rendering. Prior to joining OSU, Dr. Crawfis was the Graphics Technology group leader at the Lawrence Livermore Laboratory, where he was in charge of coordinating visualization projects for the past 12 years. He has published many papers on scientific visualization and the volume rendering of scalar and vector fields. For more information, see http://www.cis.ohio-state.edu/~crawfis.