Splatting With Shadows

Manjushree Nulkar and Klaus Mueller

Department of Computer Science, State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA
{manju,mueller}@cs.sunysb.edu

Abstract. In this paper we describe an efficient approach to add shadows to volumetric scenes. The light emitted by the lightsource is properly attenuated by the intervening volumetric structures before it is reflected towards the eye. Both parallel and perspective lightsources can be efficiently and accurately modeled. We use a two-stage splatting approach. In the first stage, a light volume is constructed in $O(N^3)$ time, which is about the same time it takes to render a regular image. This light volume stores the volumetric attenuated light arriving at each grid voxel and only needs to be recomputed if the light source is moved. If only diffuse shading is required, then the contribution of any number of lightsources can be stored in the same space. The second stage is formed by the usual rendering pipeline. The only difference is that the light contributions are interpolated from the light volume, instead of using the constant light source intensity. Once the light volume is computed, the actual rendering is only marginally more expensive than in the unshadowed case. The rendered images, however, convey three-dimensional relationships much better and look considerably more realistic, which is clearly needed if volume graphics is to become a mainstream technology.

1 Introduction

Volume rendering is a popular technique to visualize datasets that come on three-dimensional grids. Based on their structure, volumetric grids can be divided into two main groups: (i) cubic or uniform rectilinear, as typical for datasets acquired from medical scanners, such as MRI, CT, PET, SPECT, and 3D US, or (ii) curvilinear or irregular, as is most often the case for datasets output by computational methods, such as CFD or FE. This paper focuses on the former group. Volumetric datasets obtained via voxelization of analytically defined or polygonal data are most commonly members of the first group as well.

Native volumetric datasets can be visualized in two ways. (1) They can be converted into a polygonal mesh representing some iso-surface [12] in the level-set of the volume, which can then be rendered with fast and ubiquitous polygon rendering hardware, or (2) they can be viewed without conversion using direct volume rendering algorithms, which tends to be slower but allows the iso-surface to be changed on the fly, emphasizing different structures and aspects of the volume at will. Retaining the full volume also allows the rendering of soft surfaces and amorphous phenomena, such as clouds, gas, steam, which would be difficult to represent with polygonal models.

The spectrum of direct volume rendering algorithms can be divided into four major

families: raycasting [11][24], splatting [26], shear-warp [10], and texture-mapping hardware-accelerated [4]. The quality and computational overhead of these was recently evaluated by Meissner et. al. [14]. There it was found that the surveyed splatting algorithm, *i.e.*, image-aligned sheet-buffered splatting [18], produces images of comparable quality to raycasting, but renders faster when objects are sparse or moderately irregular.

Shadows play an important role in the perceived realism of a computer-generated scene. They also provide additional depth cues to the viewer. This was recognized quite early in the development of polygonal renderers. The shadow z-buffer algorithm by Williams [28] first renders a z-buffer image from the view of the lightsource. This shadow z-buffer image is then used to determine if an object point visible from the eye is also visible from the lightsource, and thus lit by it. The shadow volume algorithm by Crow [5] constructs polygonalized solids that model the volume of shadow cast into space by the silhouette of an occluder. During the rendering, a visible point is first verified that is does not fall inside such a shadow volume before it is being lit. Finally, fast shadow algorithms project the occluders onto a flat surface, producing a planar shadow polygon that can be projected as a texture using graphics hardware. Although this fake-shadow approach [3] is inaccurate for non-planar objects, it is frequently used in computer games where speed of image generation is of utmost importance.

Most of the shadow renderers for polygonal scenes capitalize on the fact that the cast shadows are independent of the viewing direction, as long the light source stays in a fixed place. This turns the generation of the shadow datastructure into a pre-processing task. Furthermore, the polygonal shadow renderers also divide the scene into fully lit (the points outside the shadow volume) and fully unlit (the points inside the shadow volume) portions. This gives rise to sharp shadows, although shadow volumes for soft shadows, as generated by extended lightsources, have also been implemented [19] (for a survey of shadow algorithms see also [29]).

Shadow volumes and the shadow z-buffer were defined for boundary representations and not for volumetric datasets where all space is filled with attenuating material. There, as light traverses the volume and is reflected towards the eye, it is continuously attenuated by the volumetric densities it traverses. This effect is not modeled by either shadow volumes nor shadow z-buffers, and hence these algorithms are not suitable for direct volumetric rendering.

However, an algorithm that works for both domains is raytracing. Although slow, it traditionally produces images of considerably higher quality than polygon projection methods, and this also holds true for the generation of shadows. Soft shadows with translucent effects have been generated using raytracing, for surface representations [27] as well as for volumetric datasets [21]. More recently, Behrens [2] utilized 3D texture mapping hardware for fast shadow generation to suggest depth relationships on unshaded volumetric objects. This approach first constructs a shadowed volume using the hardware, which can then be viewed from the eyepoint in place of the original, unshadowed volume. The performance drops less than 50% when shadows are included, however, only orthograpic (parallel) lightsources can be modeled without considerable overhead. Finally, global illumination methods and volumetric radiosity [6][13][20] can also produce realistic shadow effects but image generation tends to be slow.

Recently, Dobashi et.al. [7] proposed a method for the modeling of light attenuation

effects of clouds. Their method is probably the one most closely related to ours, as it, too, represents the volumetric objects (the clouds) as an aggregation of smoothly varying basis functions (here metaballs [30]). The metaballs are first rendered under parallel projection from the viewpoint of the sun to calculate the amount of light arriving at each metaball center, and then they are, as lit metaballs, projected to the screen. A basic splatting algorithm is employed for both rendering steps.

In this paper, we combine the efficiency of the recently introduced image-aligned splatting algorithm [18] with the realism of volumetric shadows. The algorithm uses splatting for the rendering as well as for the shadow generation process. Both parallel and perspective lightsources, with or without hood, can be easily and accurately modeled. By performing all shadow-related calculations in a pre-processing step, the rendering of volumes with shadows is as efficient as the rendering without them. The preprocessing is only required if lightsources are moved, but since splatting is also used at that stage, it is not any costlier than the volume rendering itself. The paper is structured as follows: Section 2 will present some preliminaries, mainly on splatting. Section 3 will outline the theoretical concepts, while Section 4 will describe our implementation of these as well as a few extensions of the basic framework. Finally, Section 5 and Section 6 present results and conclusions, respectively.

2 Preliminaries

The splatting algorithm was proposed by Westover [26]. It works by representing the volume as an array of overlapping basis functions, commonly Gaussian kernels with their amplitudes scaled by the voxel values. An image is generated by projecting these basis functions to the screen. The screen projection of the radially symmetric 3D basis function can be efficiently achieved by the rasterization of a precomputed 2D footprint lookup table, where each footprint table entry stores the analytically integrated kernel function along a traversing ray. A major advantage of splatting is that only voxels relevant to the image must be projected and rasterized. This can tremendously reduce the size of the volume data that needs to be both processed and stored [17]. Another advantage of splatting is that filtering for the purpose of anti-aliasing can be efficiently achieved by stretching (and attenuating) the footprints before they are rasterized to the screen [23]. Stretching the footprint along one direction reduces the bandwidth of the underlying kernel in the same direction. This is useful to produce blurring effects as well as to prevent aliasing when the volume is viewed below grid resolution or in perspective [23]. In the latter case, the kernels are stretched for volume portions further away from the eye point, where the sampling density falls below the grid resolution [15].

The most basic splatting approach [25] simply composites all kernels on the screen in back-to-front order. Although this is the fastest method, it can cause color bleeding and also introduce sparkling artifacts in animated viewing due to the imperfect visibility ordering of the overlapping kernels. This may not be a problem if the objects are highly amorphous, as, for example the clouds in [7], but it can be highly noticeable for more opaque and structured objects. An improvement in these regards is Westover's sheet-buffer method [26] that sums the voxel kernels within volume slices most parallel to the image plane. Although doing so eliminates the color bleeding artifacts in still frames, it

introduces very noticeable brightness variations in animated viewing. A more recent method by Mueller et. al. [17][18] eliminates these drawbacks, processing the voxel kernels within slabs, or sheet-buffers, of width Δs, aligned parallel to the image plane — hence the approach was termed *image-aligned sheet-buffered splatting*: All voxel kernels that overlap a slab are clipped to the slab and summed into a sheet buffer, followed by compositing the sheet with the sheet in front of it. Efficient kernel slice projection can still be achieved by analytical pre-integration of an array of kernel slices and by using fast footprint rasterization methods to project these to the screen [9]. Fig. 1 illustrates the image-aligned splatting algorithm in closer detail. Although the shadow algorithm described here would work with both incarnations of splatting, we choose the image-aligned one for its better image quality.

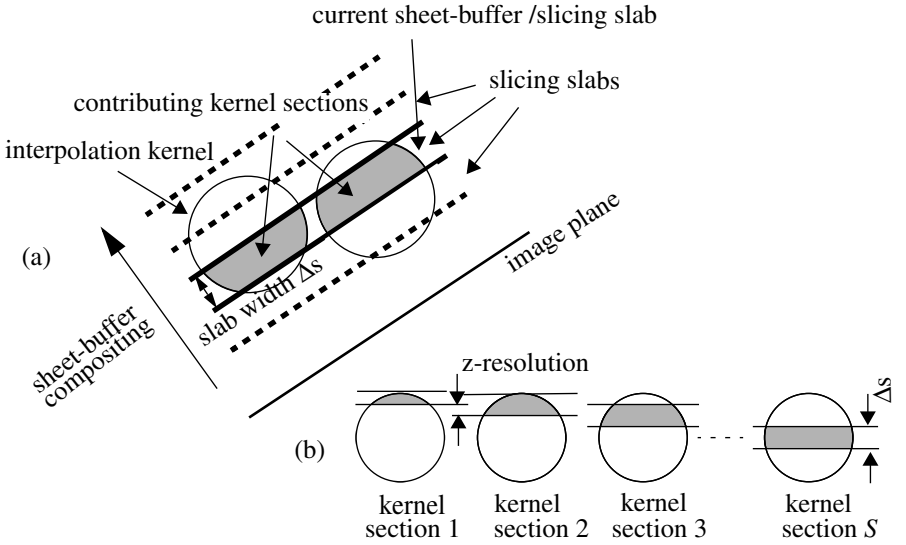


Fig 1. Image-aligned sheet-buffered splatting (from [17]). (a) All kernel sections that fall within the current slicing slab, formed by a pair of image-aligned planes spaced apart by the sampling interval Δs, are added to the current sheet buffer. The sheet-buffers are composited in front-to-back order. (b) Array of pre-integrated overlapping kernel sections (shaded areas). The integration width of the pre-integrated sections is determined by the slab width Δs, while the z-resolution is determined by the number of kernel sections.

3 Theory

3.1 Low-albedo light transport

The low-albedo volume rendering integral is written as:

$$C(\boldsymbol{x}, \boldsymbol{r}) = \int_0^L C(s)\mu(s)e^{\left(-\int_0^l \mu(t)dm\right)}dl \qquad \begin{array}{l} s = EYE + r \cdot l \\ t = EYE + r \cdot m \end{array} \tag{1}$$

where $C(\boldsymbol{x}, r)$ is the spectrum of light (usually divided into a red, green, and blue color component) that comes from a direction defined by the unit vector r and is received at location $\boldsymbol{x}$ on the image plane. *EYE* is the location of the eye point, L is the length of the ray, $\mu(\boldsymbol{s})$ is the extinction (or density) at a location $\boldsymbol{s}$ along the ray, and $C(\boldsymbol{s})$ is the spectrum of light at $\boldsymbol{s}$.

The sampled color $C(\boldsymbol{s})$ can be expanded into:

$$C(s) = C_e(s) + C_r(s) \tag{2}$$

where $C_e(\boldsymbol{s})$ is the color emitted at $\boldsymbol{s}$ and $C_r(\boldsymbol{s})$ is the color received at $\boldsymbol{s}$ and reflected towards the eye. $C_r(\boldsymbol{s})$ is usually obtained via the traditional illumination equation [8]:

$$C_r(s) = C_a k_a + C_l C_o(s) k_d \cdot N(s)L(s) + C_l k_s \cdot (N(s)H(s))^{ns} \tag{3}$$

where C_a is the ambient color, k_a is the ambient material coefficient, C_l is the color of the light source, C_o is the color of the object (determined by the transfer function), k_d is the diffuse material coefficient, N is the normal vector (determined by the gradient), L is the light direction vector, k_s is the specular material coefficient, H is the halfvector, and ns is the Phong exponent. The entities k_a, k_d, k_s, and C_a are not dependent on the sample location, while N, L, and C_o are. Further, for volume renderers that only model the attenuation of light after it has been reflected but not before, C_l is independent of $\boldsymbol{s}$ as well. Although this is physically correct for the emitted light, it is not so for the reflected light where the light also incurs attenuation on its path from the light source to the reflection site. In many applications, this deficiency is not really a disadvantage, since shading is only used to give depth cues and the light intensity is simply set to unity. However, if one desires a more physically correct illumination model which allows for both shadowing and self-shadowing, one should use this equation:

$$\begin{aligned} C_r(s) = {} & C_a k_a + C_l(s) C_o(s) k_d \cdot N(s)L(s) \\ & + C_l(s) k_s \cdot (N(s)H(s))^{ns} \end{aligned} \tag{4}$$

where C_l is a function of $\boldsymbol{s}$. C_l is then computed as:

$$C_l(s) = C_L \cdot e^{\left(-\int_s^T \mu(t)\,dm\right)} \qquad t = s + r_l \cdot m \tag{5}$$

where C_L is the color of the lightsource, r_l is the unit direction vector that points from s to the light source location *LGT*, and T is the distance $\boldsymbol{s}$ - *LGT*.

In a practical implementation, the integrals are replaced by Riemann sums which puts equation (1) as follows (see e.g. [11]:

$$C(\boldsymbol{x}, \boldsymbol{r}) = \sum_{i=0}^{L/\Delta s} C(s_i)\alpha(s_i) \cdot \prod_{j=0}^{i-1} (1 - \alpha(s_j)) \qquad \begin{aligned} s_i &= EYE + r \cdot i \cdot \Delta s \\ s_j &= EYE + r \cdot j \cdot \Delta s \end{aligned} \tag{6}$$

where α is known as opacity = (1 - transparency), and Δs is the ray sampling interval.

The weighting of the colors with α-terms is called compositing.

Likewise, equation (5) is written as follows:

$$C_l(s_i) = C_L \prod_{k=1}^{T/(\Delta s)} (1 - \alpha(s_k)) \qquad s_j = s_i + r_l \cdot k \cdot \Delta s \tag{7}$$

3.2 The effect of imperfect interpolation filters

The rendering rays must sample the volume grid at discrete positions along their paths. This sampling process can be thought of as a two-step process, performed at each ray sample location: First, the discrete grid signal is reconstructed into a continuous signal, which is subsequently resampled to yield the sample value. Of course, placing the interpolation filter at the sample location and integrating the sample neighborhood by the kernel function performs these two steps simultaneously, in place. However, for the sake of quantifying the errors of the various shadow algorithms presented later, it is helpful to imagine that these two steps are completely disjoint, *i.e.*, the grid is first completely reconstructed into a continuous representation (by ways of convolution with the interpolation filter kernel), and then the rays traverse this continuous function, sampling it at discrete locations. It is the reconstruction step that introduces the well known lowpassing artifacts — unless an (impractical) *sinc* filter is used for this purpose. Hence, the number of times a volume must be reconstructed before an image is generated determines the amount of lowpassing that will occur. Specifically, each such lowpassing operation will yield a continuous signal that is equivalent to the original discrete signal convolved with a wider and wider interpolation filter, one that suppresses an increasing amount of the signal's upper passband, causing an increasing amount of smoothing. We shall soon see that different approaches used for shadow generation vary in the number of reconstructions that need to be performed, which is crucial for the image quality that can be obtained. In the following, we will discuss three such algorithms, trading complexity with the number of lowpassing operations that they perform. We will, for now, assume that there is only one lightsource in the scene, and outline a solution for multiple lightsources in Section 4.

3.3 Computing the light attenuation during rendering

This is the most straightforward approach and a direct implementation of equation (6): At each sample location $\boldsymbol{s}$, a ray r_l is cast from $\boldsymbol{s}$ to the light source and the amount of attenuation that the light source spectrum undergoes by the intervening volume densities is computed. This process is illustrated in Fig. 2a. To compute the complexity associated with this, let us assume that the volume is cubic with N^3 voxels, Δs=1, and the resolution of the rendered image is $M \times M$, *i.e.*, there are M^2 rendering rays. In that case the complexity of the rendering is $O(M^2 \cdot N + M^2 \cdot N \cdot N) = O(M^2 \cdot N^2)$, where the first term is the cost for interpolating M^2 rays at O(N) sample locations, while the second term is the complexity to cast shadow rays from each of these $O(M^2 \cdot N)$ sample locations and sampling those at O(N) locations. Hence, the complexity is $O(M^2 \cdot N^2)$, one magnitude of N higher than rendering without light attenuation effects.

Let us now assess the degree of lowpassing of this approach. Following the theoret-

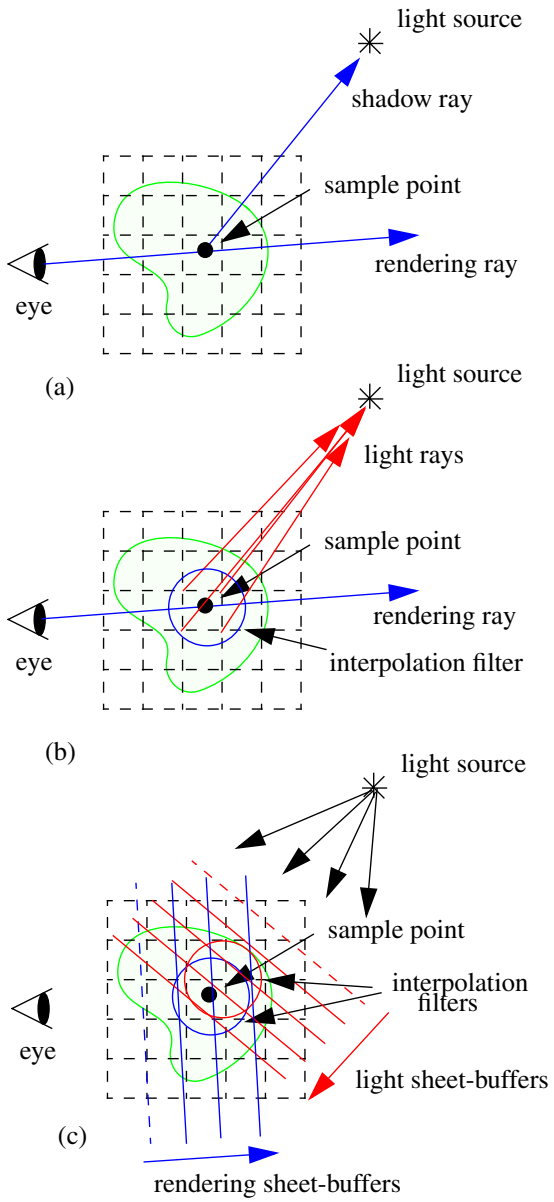


Fig 2. Different ways to compute light pre-reflective light attenuation. Items drawn in red denote operations done in a pre-processing step that do not have to be repeated for every rendered frame, while items drawn in blue denote operations that are required for each rendered frame. (a) Casting a light ray for every interpolated and shaded sample. (b) Casting light rays from every voxel to compute a light volume, whose voxels are then interpolated during rendering to obtain a sample's lighting. (c) Constructing the light volume via splatting, which requires alternating forward and backward projection steps, and then using this light volume for the rendering, done also using splatting.

ical framework presented in Section 3.2, we observe that once the volume has been reconstructed into a continuous form, all rays can be cast in their entirety from light-source (emission) → sample (reflection) → image (reception). Hence we conclude that this approach will lowpass the discrete volume only once during image generation, which is optimal.

3.4 Pre-computing a light-volume using raycasting

We can return to a rendering complexity of $O(M^2 \cdot N)$ by computing the light attenuations beforehand and storing these in a *light volum*e, in which each voxel holds the amount of light received. An alternative to this is a *shadow volume* which would store the attenuations. Note that this shadow volume is not binary as in the polygonal approaches mentioned earlier, but is continuous where each voxel can assume values in the range [0, 1]. Constructing the light volume requires a ray to be sent from each voxel to the light source, measuring the attenuation of the light on its way to the voxel. Equation (7) becomes:

$$C_l(v) = C_L \prod_{j=1}^{T/(\Delta s)} (1 - \alpha(s_j)) \qquad s_j = v + r_l \cdot j \cdot \Delta s \tag{8}$$

where v is the location of the illuminated grid voxel and r_L is now the unit direction vector that points from v towards *LGT*. The complexity of this process is $O(N^3 \cdot N) = O(N^4)$. Once the light volume is constructed, it can be interpolated at the sample locations $\boldsymbol{s}$ during rendering to retrieve the properly attenuated amount of light received from the light source at $\boldsymbol{s}$. The rendering complexity is the same than that of a standard raycasting algorithms, $O(M^2 \cdot N)$, with one extra interpolation per sample location to interpolate the light. If the image has about the same resolution than the volume, then M=N and the rendering complexity is $O(N^3)$. The amount of lowpassing to obtain the shadow effects, however, has increased by one: First we reconstructed the volume to get the attenuated light to the voxels (the pre-processing step), and then we reconstructed the light volume during the rendering to interpolate the attenuated light at the sample points (Fig. 2b).

3.5 Pre-computing a light-volume using splatting

Splatting can help to reduce the complexity of the light-volume generation. We will now describe a framework that uses splatting at all stages of the volume rendering: (1) for light-volume generation and (2) for image generation.

Let us begin with re-stating equation (6) in the context of splatting:

$$C(\boldsymbol{x}, \boldsymbol{r}) = \sum_{i=0}^{L/\Delta s} \bar{C}(s_i)\bar{\alpha}(s_i) \cdot \prod_{j=0}^{i-1} (1 - \bar{\alpha}(s_j)) \qquad \begin{aligned} s_i &= EYE + r \cdot i \cdot \Delta s \\ s_j &= EYE + r \cdot j \cdot \Delta s \end{aligned} \tag{9}$$

where $\bar{C}(s_i)$ and $\bar{\alpha}(s_i)$ are the averaged (over Δs) colors and opacities, respectively, as computed from the footprint tables [18]. Image-aligned splatting can be thought of as a raycaster that casts all rays simultaneously in lock-step as a rayfront (see Fig. 2c). Thus, for some fixed i, all s_i for all rays are available in one sheetbuffer, and are composited from front to back. Due to this simultaneous raycasting approach, equation (8) changes

quite significantly:

$$C_l(v) = C_L \cdot \frac{\sum_{l,m} \left(\prod_{k=0}^{l-1} (1 - \bar{\alpha}(s_k)) \right) \bar{h}(s_{l,m} - v)}{\sum_{l,k} \bar{h}(s_{l,m} - v)} \qquad \begin{array}{l} s_l = LGT - r_m \cdot l \cdot \Delta s \\ m = 0, 1, ..., res \cdot N^2 \end{array} \tag{10}$$

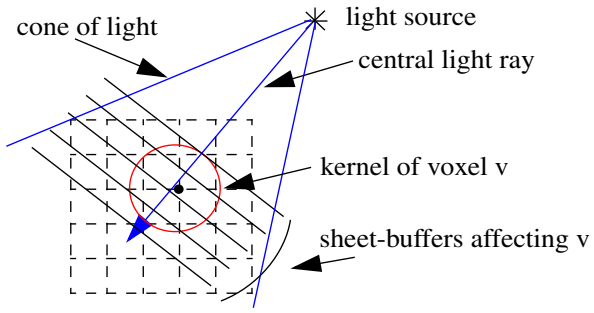
where $s_{l,m}$ is the m-th sample point on the sheetbuffer $l \cdot \Delta s$ away from the light source, and h is the interpolation kernel. Note that since splatting stores the averaged (over a line of length Δs) kernel values, we need to write $\bar{h}$ instead of h. Note also that we process the voxels in front-to-back order as seen from the lightsource. To understand equation (10) better, let us think of the task of constructing the light volume as one in which the grid voxels must sample the continuous light field that is formed by light rays emanating from the light source and being attenuated by the volume densities. This light field is available in discrete form in the composited sheet buffers as they progress from the light source (the product term in equation (10)). Sampling this light field consists of splatting the sheetbuffer content back into the grid (the sum term in equation (10)). Dividing by the sum of kernel weights affecting a voxel v provides the necessary normalization.

The complexity is $O(N^3)$ for the rendering (similar to the raycasting approach in Section 3.4), but also $O(N^3)$ for the generation of the light volume, a magnitude lower than for the raycasting approach. However, there are now three reconstruction and lowpassing steps associated with the shadow data. The first two occur during the computation of the light volume, i.e., (1) when splatting the volume opacities into the sheetbuffers (*i.e.*, reconstructing the volume, which is sampled by the sheet buffers), and (2) when splatting the composited sheet-buffer opacities into the voxels of the light volume (*i.e.*, reconstructing the sheet buffers, which is sampled by the volume). The third reconstruction and lowpassing operation occurs during rendering, when splatting the attenuated light into the sheet-buffers (*i.e.*, reconstructing the volume, which is sampled by the sheet-buffers). Fig. 2c illustrates all three operations.

Note that a raycasting approach that starts at the lightsource (instead of at the voxels) would achieve the same effects. We chose splatting since we wanted to have a common algorithm for both tasks: light volume construction and rendering. In addition, splatting will reduce the complexity of the light volume generation for all those volume datasets that also benefit from splatting being used in the rendering phase, *i.e.*, sparse or irregular volumetric objects. Please refer to [14] for a detailed comparison of raycasting vs. splatting in terms of their rendering costs. If we assume the standard volume rendering configuration in which the image resolution is the same than the volume resolution, *i.e.*, M=N, then the theoretical rendering complexity of the raycasting approach outlined in Section 3.4 matches that of the splatting approach.

4 Implementation

In this section we will describe the practical aspects of our new enhancement for splatting. Consider Fig. 3 for an illustration of the basic algorithm for one lightsource.




```
ComputeLightVolume(Volume *v)
    // initialize data structures
    Init (lightVolume, 0);
    Init (weightVolume, 0);
    Init (attenuationBuffer, 0);
    Init (opacityBuffer, 0);
    place sheetbuffer perpendicular to the central light ray and to the first voxel with opacity > 0
    // start generating the light volume
    while sheetbuffer is inside the volume {
      for all voxels vj within sheetBuffer width {
          if OpacityTransferFunction[density(vj)] = = 0;   // no need to process invisible voxels
              continue;
          pick appropriate footprintTable for vj;
          v' ← project vj onto sheetbuffer;
          for all sheetBuffer pixels pi that fall inside footprint(v')  {  // rasterize the footprint
              weight = footprintTable[|v' - pi|];
              // forward splat: the light reaching the voxel is prop. to (1 - attenuation integral)
              opacityBuffer[pi] += weight · OpacityTransferFunction[density(vj)];
              // reverse-splat the light and the interpolation  weights
              lightVolume[vj] += weight · (1 - attenuationBuffer[pi]);
              weightVolume[vj] += weight;
      }     }
      // composite opacities for next reverse splat
      attenuationBuffer = opacityBuffer · (1 - attenuationBuffer) + attenuationBuffer;
      advance sheetBuffer by Δs;
    }
    // normalize
    for all voxels vj with OpacityTransferFunction[density(vj)] > 0
      lightVolume[vj] = LightSourceIntensity · (lightVolume[vj] / weightVolume[vj]);
```

Fig 3. Pseudocode for the computation of the light volume in splatting.

Depending on the size of the interpolation kernel a light volume voxel will receive contributions from a number of consecutive attenuation sheet-buffers. It will also contribute to a number of consecutive opacity sheet-buffers (see Fig. 3a). This manifests two of the three interpolation operations that are necessary for this algorithm (the third is the interpolation during rendering).

Fig. 3a shows the (perspective) light source as a single point which irradiates the volume in a divergent fashion. We have mentioned before that in order to prevent aliasing artifacts in perspective rendering, the splatting kernels must be progressively stretched as a function of distance from the lightsource. At the same time the kernel amplitude must be progressively lowered to be in conformance with the Fourier scaling

theorem. The same needs to be done during light volume construction, both in the forward splatting and in the reverse splatting. On the other hand, far-away lightsources can be modeled as well by simply running the algorithm in parallel-projection mode. Then we only need to supply a light direction vector and orient the sheet buffer orthogonal to it.

The light volume generated in ComputeLightVolume() can be used repeatedly as long the light source position and intensity does not change. Rendering is performed as usual for splatting [17], only now the light volume must be splatted as well to obtain the $C_l(\boldsymbol{s_i})$ in equation (6). The code in Fig. 3 uses one white light source (just one intensity channel), which gives rise to a single-channel light volume. Thus, during rendering, if we perform post-classification on the sheet-buffer [16], there will be one extra sheet-buffer required to sample the light volume. Then, during shading, the pixels in the light sheet buffer are used to get the light intensity instead of the light source directly. Thus there are no extra operations beyond the light sheet buffer rasterizations.

There is a variant to the approach outlined above: Instead of a light volume one can use an attenuation volume, in which the terms (lightVolume[$\mathbf{v_j}$] / weightVolume[$\mathbf{v_j}$]) are stored. This allows one to change the color and intensity of the light source during rendering.

In addition to the memory needed to store the density volume, our approach requires additional memory to hold the light volume. If there is only one light source, then two extra unsigned shorts, or floats for extra precision, must be provided for each voxel: one for the weights used for normalization and one for the light intensity. The former goes unused after the normalization has been performed. In fact, once the light volume has been normalized, only one extra byte per voxel is necessary. However, one can easily preserve memory by performing the normalization on-the-fly by ways of a rolling normalization buffer, where a sheet of voxels is normalized immediately after their kernels have been traversed by all their relevant sheet-buffers.

If m lightsources are used and view-dependent specular lighting effects are desired, then m bytes will be required per voxel to hold the attenuated light for each source. The weight volume for normalization can be reused for each lightsource. If view-dependency effects are not to be considered *(i.e.*, only diffuse shading is implemented), then shading may be performed during light volume computation and all light contributions can be collapsed into one byte. If colored lightsources are used, then three bytes are required. If the shading is performed during normalization of each attenuated light, then one would need only two unsigned shorts or floats for the unnormalized light and the normalization weights.

Due to the repeated lowpassing (*i.e.*, blurring), the surface voxels of very opaque objects may end up receiving insufficient light due to self-shadowing. This causes the images to appear somewhat dark. We can fix this by adding a constant term to all voxels in the light volume, reducing the opacities in the transfer function during light volume construction, or using a light source intensity greater than unity (but clamping the light volume to 1.0). Further, using smaller Gaussians that attenuate higher frequencies less also helps to reduce these effects. Another remedy is to simply add only light to an individual light volume voxel when it is first encountered, *i.e.*, when it is touched by the attenuation buffer for the first time. In that case a weight volume is not needed.

Although this procedure is only approximate from a signal processing point of view, it does produce brighter images since the voxel self-shadowing is reduced.

Finally, it is sometimes advisable to use two sets of opacity transfer functions, one for the light volume generation and one the rendering. If the opacities for the light volume generation are chosen lower then some light is able to penetrate across thin but opaque structures and so faintly illuminate structures on the other side. Although not physically correct, this trick tends to generate nicer images at times.

5 Results

We ran our algorithm on a few volumetric datasets stemming from a number of different domains and applications. The first page of this paper shows a volume graphics scene: a voxelized chair with parallel and perspective shadows, respectively. The shadows add a considerable amount of realism and mood to this minimalistic scene. Fig. 4 (colorplate) shows the bottom of the lobster dataset in full view and zoomed-up, with and without perspective shadows. Note the shadows cast by the legs onto the lobster's body: Only in the shadowed image one can truly discern the legs sticking out of the lobster's shell. However, we also note that the images are somewhat darker, even on surfaces that are not in shadow. This is a manifestation of the attenuation leakage due to the lowpassing. Fig. 5 (colorplate) shows a physical simulation of diesel being injected into a cylinder of an engine filled with air. Note the soft shadows cast by the semitransparent gas. Fig. 6 (colorplate) shows a number of renderings of a voxelized chair.

The image-aligned sheet-buffered splatting engine is identical to that described in [9]. All we have added here is the ability to splat (sheet-buffer) energy into the grid during light volume calculation, to run the splatting algorithm from the viewpoint of the light source, and to splat one extra volume, *i.e.*, the light volume, during image generation. Thus, in the general case, the rendering performance is bound to be very similar to that already reported in [14] for a wide variety of datasets and image sizes. We have observed that the construction of the light volume took about 25% longer than the generation of an image, due to the required backprojection. This light volume rendering time grows proportional to the number of light sources. The images themselves took virtually the same time to generate, with or without shadows.

Finally, we also implemented all three shadow generation variants within a simple, unoptimized raycaster. The timings obtained for the lobster dataset are given in the following table:

	variant 1 (section 3.3)	variant 2 (section 3.4)	variant 3 (section 3.5)
light volume generation time	-	132 s	35 s
rendering time	44 s	29 s	29 s

We observe that it takes roughly 1/4 of the time to generate the light volume with the reverse splatting method of variant 3 than with the raycasting method of variant 2. We also observe that rendering the shadows on the fly, without precomputing a light volume, adds about 50% to the rendering time. Finally, we observe that the combined time

for light volume generation and rendering for variant 3 is 64s, which is 45% greater than rendering the shadows without precomputing the light volume (variant 1). Note, however, that one can easily reuse the light volume for a number of close-by frames without noticeable error, and in this regard, the light volume generation is already amortized after 3 frames.

Fig. 7 shows renderings of the lobster with all 3 variants using the raycaster. We notice that variant 3 produces only slightly darker images than the other two variants.

variant 1 variant 2 variant 3

Fig 7. Renderings of the lobster dataset for the three shadow generation variants using a simple raycaster (timings are given in the table above).

6 Conclusions

We have described an efficient method to add shadows and attenuated light effects to volumetric scenes. In contrast to most volume renderers, the light is attenuated by the volumetric densities on its entire path across the volume: (1) from the source to the reflecting surface, and (2) from the reflecting surface to the eye. Our algorithm uses splatting for the entire two-stage process, *i.e.*, light volume generation and rendering. Existing software or hardware for splatting only requires minor modifications to enable the rendering of attenuated light and shadows. What needs to be added is the ability to modify a voxel value (a volume write-back mechanism) and a divider unit (for the normalization step).

An advantage of using splatting, instead of raycasting, for the generation of the light volume is that the computational complexity can be reduced by one order of magnitude. This is in addition to the other advantages that this object-centric, point-based method may be able to deliver [14]. However, a downside is that the described splatting approach lowpasses the volume data related to the light attenuation more than the alternative raycasting methods. This may cause the attenuations to "leak out" of the object interior, which can lead to globally darkened surfaces at rendering time. Our experiments indicate, however, that these effects may not be as severe as initially anticipated.

In the future, we plan to extend the existing framework to model extended light-sources by convolving the interpolation kernel with the shape of the light source, and using this interpolation kernel in place of the usual one. Doing so will generate softer shadows with penumbras. A similar technique was used by Soler and Sillion [22] to generate smooth shadow maps. Our framework would also allow the modeling of global

illumination by making more than one sheetbuffer pass across the volume, from different directions. In that case the sheetbuffers would sample the volume along their path and distribute this energy among voxels further down the trajectory. Conceptually, this approach is similar to the one presented in [6], but can benefit from the efficiency of the splatting algorithm. Finally, we plan to model the spectral filtering of the light by the traversed medium, which would result in colored shadows. We can easily achieve this by specifying spectrally dependent transmission transfer functions to control the calculation of light attenuations during light volume generation. This would scale the present light volume memory overhead by a factor of 3.

Acknowledgments

We would like to thank Jian Huang and Roger Crawfis for the basic splatting software [1] that we extended in this work. Special thanks go to Poojan Prabhu for implementing the three shadow algorithms into his raycaster, allowing the qualitative and quantiative comparison of the three variants. We would also like to thank SFB 382 of the DFG (German Research Foundation) for the fuel dataset. Finally, we would like to thank the anonymous reviewers for their thoughtful suggestions.

References

[1] http://www.cis.ohio-state.edu/graphics/research/FastSplats/

[2] U. Behrens and R. Ratering, “Adding shadows to a texture-based volume renderer,” *1998 Symposium on Volume Visualization*, pp. 39-46,1998.

[3] J. Blinn, “Me and my (fake) shadow,” *IEEE CG&A*, vol. 9, no. 1, pp. 82-86, 1988.

[4] B. Cabral, N. Cam, and J. Foran, “Accelerated volume rendering and tomographic reconstruction using texture mapping hardware”, *1994 Symposium on Volume Visualization*, pp. 91-98, 1994.

[5] F. Crow, “Shadow algorithms for computer graphics,” *Proc. SIGGRAPH’77*, pp. 242-247, 1977.

[6] F. Dachille, K. Mueller, Ari Kaufman, "Volumetric backprojection," *2000 Symposium on Volume Rendering*, pp. 109-117, 2000.

[7] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, T. Nashita, “A simple, efficient method for realistic animation of clouds,” *Proc. SIGGRAPH 2000*, pp. 19-29, 2000.

[8] F. Foley, A. Van Dam, S. Feiner, J. Huges, *Computer Graphics: Principles and Practice*, Addison Wesley, 1996.

[9] J. Huang, K. Mueller, N. Shareef, R. Crawfis, "FastSplats: Optimized Splatting on rectilinear grids," *Visualization’2000*, pp. 219-227, 2000.

[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] M. Levoy, “Efficient ray tracing of volume data”, *ACM Trans. Comp. Graph.*, vol. 9, no. 3, pp. 245-261, 1990.

[12] W. E. Lorensen and H. E. Cline, “Marching cubes: a high resolution 3D surface construction algorithm”, *Proc. SIGGRAPH’87*, pp. 163-169, 1987.

[13] N. Max, “Optical models for direct volume rendering”, *IEEE Trans. Vis. and Comp. Graph.,* vol. 1, no. 2, pp. 99-108, 1995.

[14] M. Meissner, J. Huang, D. Bartz, K. Mueller, R. Crawfis, "A practical comparison of popular volume rendering algorithms," *2000 Symposium on Volume Rendering*,

pp. 81-90, Salt-Lake City, October 2000.

[15] K. Mueller, T. Möller, J.E. Swan, R. Crawfis, N. Shareef, and R. Yagel, "Splatting errors and antialiasing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 2, pp. 178-191, 1998.

[16] K. Mueller, T. Möller, and R. Crawfis, "Splatting without the blur", *Proc. Visualization'99*, pp. 363-371, 1999.

[17] K. Mueller, N. Shareef, J. Huang, and R. Crawfis, "High-quality splatting on rectilinear grids with efficient culling of occluded voxels", *IEEE Trans. Vis. and Comp. Graph.*, vol. 5, no. 2, pp. 116-134, 1999.

[18] K. Mueller and R. Crawfis, "Eliminating popping artifacts in sheet buffer-based splatting", *Proc. Visualization'98*, pp. 239-245, 1998.

[19] T. Nishita and E. Nakamae, "Continuous tone representation of three-dimensional objects taking account of shadows and interreflections, " *Proc. SIGGRAPH'85*, pp. 23-30, 1985.

[20] H. E. Rushmeier and K. E. Torrance, "The zonal method for calculating light intensities in the presence of a participating medium," *Proc. SIGGRAPH 87*, pp. 293-302, 1987.

[21] L. Sobierajski, and A. Kaufman, "Volumetric raytracing," 1*994 Symposium on Volume Visualization*, pp. 11-18, 1994.

[22] C. Soler and F.X. Sillion, "Fast calculation of soft shadow textures using convolution," *Proc. SIGGRAPH'98*, pp. 321 - 332, 1998.

[23] J.E. Swan, K. Mueller, T. Moeller, N. Shareef, R. Crawfis, and R. Yagel, "An antialiasing technique for splatting," *Proc. Visualization'97*, pp. 197-204, 1997.

[24] H. Tuy and L. Tuy, "Direct 2D display of 3D objects", *IEEE Comp. Graph. & Appl*., vol. 4 no. 10, pp. 29-33, 1984.

[25] L. Westover, Splatting, "A Parallel, Feed-Forward Volume Rendering Algorithm," Ph. D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, TR91-029, 1991

[26] L. Westover, "Footprint evaluation for volume rendering", *Proc. SIGGRAPH'90*, pp. 367-376, 1990.

[27] T. Whitted, "An improved illumination model for shaded display," *Comm. ACM*, vol. 23, no. 6, pp. 343-349, 1980.

[28] L. Williams, "Casting curved shadows on curved surfaces," *Proc. SIGGRAPH'78*, pp. 270-274, 1978.

[29] A. Woo, P. Poulin and A. Fournier, "A Survey of Shadow Algorithms," *IEEE CG & A*, vol. 10, no. 6, 1990.

[30] G. Wyvill, A. Trotman, "Ray-Tracing Soft Objects," *Proc. of CG Internationa*l, pp. 439-475, 1990.

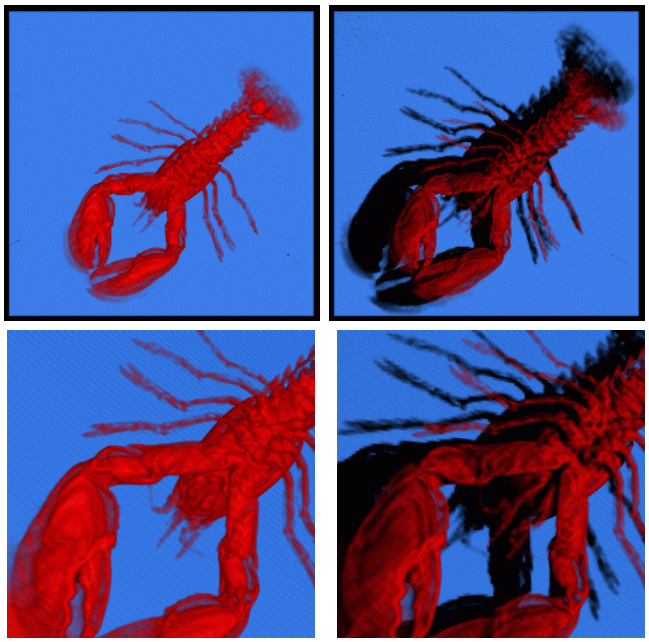


Fig 4. Lobster dataset in full view and zoomed-up, with and without perspective shadows. Note the shadows cast by the legs onto the lobster's body: Only in the shadowed image one can truly discern the legs sticking out of the lobster's shell.

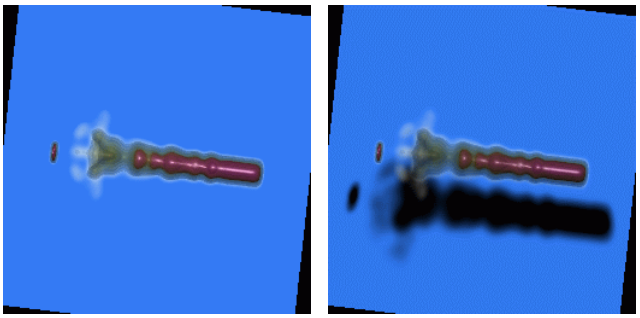


Fig 5. A physical simulation of diesel being injected into a cylinder of an engine filled with air. Note the soft shadows cast by the semitransparent gas.

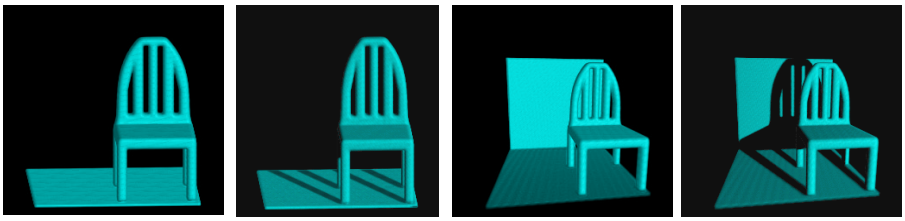


Fig 6. A voxelized chair rendered with and without parellel and perspective shadows, respectively. The shadows add significantly to the realism of the scene.