Hardware-Accelerated Adaptive EWA Volume Splatting

Wei Chen[*] Zhejiang University, China
Liu Ren[†] Carnegie Mellon University, USA
Matthias Zwicker[‡] MIT, USA
Hanspeter Pfister[§] MERL, USA

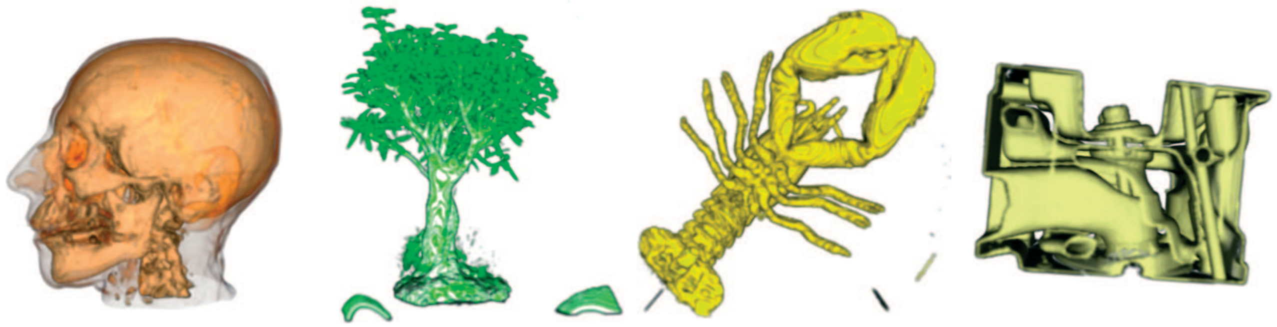


Figure 1: Adaptive EWA splatting for head (208× 256× 225, 2.86 fps), bonsai (256× 256× 128, 7.53 fps), lobster ($301 \times 324 \times 56$, 10.6 fps) and engine (256× 256× 110, 10.28 fps) data sets with 512 × 512 image resolution.

Abstract

We present a hardware-accelerated adaptive EWA (elliptical weighted average) volume splatting algorithm. EWA splatting combines a Gaussian reconstruction kernel with a low-pass image filter for high image quality without aliasing artifacts or excessive blurring. We introduce a novel adaptive filtering scheme to reduce the computational cost of EWA splatting. We show how this algorithm can be efficiently implemented on modern graphics processing units (GPUs). Our implementation includes interactive classification and fast lighting. To accelerate the rendering we store splat geometry and 3D volume data locally in GPU memory. We present results for several rectilinear volume datasets that demonstrate the high image quality and interactive rendering speed of our method.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processor; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: Direct volume rendering, volume splatting, EWA filter, hardware acceleration

1 Introduction

Splatting is a popular algorithm for direct volume rendering that was first proposed by Westover [30]. The splatting process reconstructs a continuous function from the sampled scalar field using 3D reconstruction kernels associated with each scalar value. For volume rendering, the continuous function is mapped to the screen as a superposition of pre-integrated 3D kernels, which are called 2D *footprints*. Recently, Zwicker and colleagues [35] proposed a high quality splatting algorithm called *EWA volume splatting* for aliasing-free splatting. However, achieving interactive high quality EWA splatting is still difficult due to the computational complexity of EWA filtering and insufficient commodity hardware support. These two issues limit the applicability of high quality EWA volume splatting.

In this paper, we present two major contributions addressing these issues: First, we introduce *adaptive EWA splatting*, an adaptive filtering scheme to reduce the cost of EWA computation that still achieves high quality rendering with antialiasing. The adaptive EWA splatting algorithm can be incorporated seamlessly into previous splatting systems. Second, we exploit programmable graphics hardware to achieve interactive EWA volume splatting. We present a hardware-accelerated EWA volume splatting framework that allows interactive high quality volume rendering, interactive transfer function design, and fast two-pass shading.

Our approach stores both the *proxy geometry* (i.e., the textured quads representing the splats) and the 3D volume data locally in graphics hardware for efficient access during interactive rendering. This leads to two advantages over previous approaches. First, parallel processing in graphics hardware can be fully exploited with retained-mode splatting. Second, the memory bandwidth bottleneck between CPU and GPU occurring in immediate-mode algorithms is completely avoided, facilitating interactive volume splatting. However, the memory requirements to store the proxy geometry can be very large due to the large number of voxels. We solve this problem by employing proxy geometry compression and a fast decompression procedure based on the regularity of regular or rectilinear volumes, which are commonly used in volume rendering.

The remainder of this paper is organized as follows: We first discuss related work in Section 2. We then briefly review the EWA volume splatting scheme and introduce adaptive EWA volume splatting in Section 3. Next, we present our hardware-accelerated adaptive EWA volume splatting framework in Section 4. In Section 5 we compare our approach with previous ones and present results for several rectilinear volume data sets that demonstrate the high image quality and interactive rendering speed of our method. Finally, we conclude our work in Section 6.

[*]e-mail:chenwei@cad.zju.edu.cn
[†]e-mail:liuren@cs.cmu.edu
[‡]e-mail:matthias@graphics.lcs.mit.edu
[§]e-mail:pfister@merl.com

2 Related Work

Hardware-accelerated volume rendering algorithms for rectilinear grids include ray casting [25], texture slicing [24, 3], shear-warp and shear-image rendering [21, 31], and splatting. For a detailed overview see [20]. In this paper we focus on volume splatting, which offers the most flexibility in terms of volume grids (including non-rectilinear [11]) and mixing with point-sampled geometry [35]. Splatting is also attractive because of its efficiency, which derives from the use of pre-integrated reconstruction kernels.

Since Westover's original work [29, 30], most volume splatting algorithms focus on improving the image quality, including ray-driven perspective splatting [17], edge preservation [6], eliminating popping and blur [13, 14], and image-aligned splatting [16].

The aliasing problem in volume splatting has first been addressed by Swan and colleagues [27] and Mueller and colleagues [15]. They used a distance-dependent stretch of the footprints to make them act as low-pass filters. Zwicker and colleagues [34] developed EWA splatting along similar lines to the work of Heckbert [5], who introduced EWA filtering to avoid aliasing of surface textures. They extended his framework to represent and render texture functions on irregularly point-sampled surfaces [33], and to volume splatting [35].

Point-based geometry has been successfully rendered on the GPU [26, 2, 4]. Ren and colleagues [23] derived an object space formulation of the EWA surface splats and described its efficient implementation on graphics hardware. For each point in object-space, quadrilateral that is texture-mapped with a Gaussian texture is deformed to result in the correct screen-space EWA splat after projection. The work presented in this paper builds on that algorithm and extends it to volume splatting.

Other techniques were proposed to improve splatting performance, such as opacity-based culling [16], fast splat rasterization [7], hierarchical splatting [9], object and image space coherence [8], shell splatting [1], 3D adjacency data structure [19] and post-convolved splatting [18]. Lippert and Gross [10] introduced a splatting algorithm that directly uses a wavelet representation of the volume data. Welsh and Mueller [28] used a hierarchical and frequency sensitive splatting algorithm based on wavelet transformations and pre-computed splat primitives, which accomplishes view-dependent and transfer function-dependent splatting. None of these methods have been implemented completely on the GPU.

Some GPU-accelerated splatting methods [22, 1] use texture mapping hardware for the projection and scan-conversion of footprints. In more recent work, Xue and Crawfis [32] compared several hardware-accelerated splatting algorithms, including an efficient point-convolution method for X-ray projections. They did not address anti-aliasing and reported lower performance numbers than our adaptive EWA splatting implementation.

3 Adaptive EWA Volume Splatting

Our adaptive splatting approach is based on EWA volume splatting introduced by Zwicker and colleagues [35], hence we briefly review this technique in Section 3.1 and refer the reader to the original publication for more details. We then present adaptive EWA volume splatting in Section 3.2.

3.1 EWA Volume Splatting

Volume splatting interprets volume data as a set of particles that are absorbing and emitting light. To render the data, line integrals are precomputed across each particle separately, resulting in 2D footprint functions or *splats* in the image plane. The splats are composited back-to-front to compute the final image. Particles are represented by 3D reconstruction kernels, and a common choice is 3D elliptical Gaussian kernels. We use the notation $\mathbf{G_V}(\mathbf{t}-\mathbf{p})$ to represent an elliptical Gaussian kernel centered at a 3D point $\mathbf{p}$ with a 3×3 variance matrix $\mathbf{V}$:

$$G_{\mathbf{V}}(\mathbf{t}-\mathbf{p}) = \frac{1}{(2\pi)^{3/2}|\mathbf{V}|^{1/2}} e^{-\frac{1}{2}(\mathbf{t}-\mathbf{p})^T\mathbf{V}^{-1}(\mathbf{t}-\mathbf{p})} \tag{1}$$

Although Gaussian kernels have infinite support in theory, they are truncated to a given *cutoff radius* r in practice. I.e., they are evaluated only in the range

$$(\mathbf{t}-\mathbf{p})^T\mathbf{V}^{-1}(\mathbf{t}-\mathbf{p}) \le r^2, \tag{2}$$

where usually $1 \le r \le 3$. Further, the choice of Gaussians as 3D kernels guarantees a closed-form footprint function after integration along viewing rays.

However, the change of sampling rate due to the perspective transformation in the splatting process usually results in aliasing artifacts. EWA volume splatting solves this problem by convolving the footprint function with a 2D low-pass filter, which yields an aliasing-free footprint function called the *EWA volume resampling filter*. Zwicker and colleagues [35] derived a closed-form representation of the EWA volume resampling filter that is based on the following two assumptions: First, the low-pass filter takes the form of a 2D Gaussian. Second, the nonlinear perspective transformation that maps reconstruction kernels to image space is linearly approximated using its Jacobian.

To summarize the derivation of the EWA volume resampling filter we introduce some notation. The rotational part of the viewing transformation that maps object space to camera space coordinates is given by a 3×3 matrix $\mathbf{W}$. We denote camera space coordinates by $\mathbf{u} = (u_0, u_1, u_2)$. The origin of camera space $\mathbf{u} = 0$ is at the center of projection and the image plane is the plane $u_2 = 1$. Camera space coordinates of a voxel k are given by $\mathbf{u}_k$. Image space coordinates are denoted by $\mathbf{x}$, the image space position of voxel k is $\mathbf{x}_k$. Further, the Jacobian of the perspective projection at a point $\mathbf{u}_k$ in camera space to image space is a 3×3 matrix $\mathbf{J}_{u_k}$ (see [35]):

$$\mathbf{J}_{\mathbf{u}_k} = \begin{pmatrix} \frac{1}{u_{k_2}} & 0 & -\frac{u_{k_0}}{u_{k_2}^2} \\ 0 & \frac{1}{u_{k_2}} & -\frac{u_{k_1}}{u_{k_2}^2} \\ \frac{u_{k_0}}{||(u_{k_0},u_{k_1},u_{k_2})||} & \frac{u_{k_1}}{||(u_{k_0},u_{k_1},u_{k_2})||} & \frac{u_{k_2}}{||(u_{k_0},u_{k_1},u_{k_2})||} \end{pmatrix}. \tag{3}$$

Given the 3×3 variance matrix $\mathbf{V}''_k$ of a reconstruction kernel k in object space, its transformation to image space is $\mathbf{V_k} = \mathbf{J}_{u_k}\mathbf{W}\mathbf{V}''_{\mathbf{k}}\mathbf{W}^T\mathbf{J}_{u_k}{}^T$. The EWA volume resampling filter $\rho_k(\mathbf{x})$ is now obtained by integrating the reconstruction kernel in image space along viewing rays and convolving it with the Gaussian low-pass filter. As derived by Zwicker and colleagues [35], this yields the 2D footprint function

$$\rho_k(\mathbf{x}) = \frac{1}{2\pi|\mathbf{J}_{\mathbf{u}_k}^{-1}||\mathbf{W}^{-1}||\hat{\mathbf{V}}_k+\mathbf{V}_h|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{x_k})^T\mathbf{M}_k(\mathbf{x}-\mathbf{x_k})}, \tag{4}$$

where we use the notation

$$\mathbf{M}_k = (\hat{\mathbf{V}}_k+\mathbf{V}_h)^{-1}. \tag{5}$$

Here, $\mathbf{V}_h$ is the 2×2 variance matrix of the Gaussian low-pass filter, which is usually chosen to be the identity matrix. The 2×2 variance matrix $\hat{\mathbf{V}}_k$ is obtained by skipping the third row and column of $\mathbf{V}_k$ [1].

[1]Throughout the paper, a matrix with a hat symbol denotes the result of skipping its third row and column.

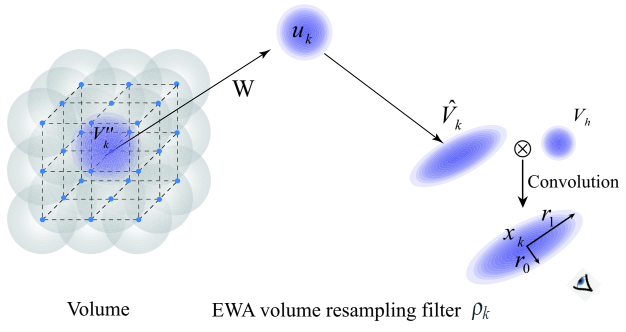


Figure 2: EWA volume splatting process.

3.2 Adaptive EWA Filtering

Even though the EWA volume resampling filter avoids aliasing artifacts because of its built-in low-pass filter, its evaluation is computationally quite expensive as is obvious from Equation 4. The motivation of *adaptive EWA volume splatting* is to simplify the evaluation in an adaptive way but still accomplish high quality, aliasing-free splatting.

Adaptive EWA volume splatting is based on the following observations: When the volume data is far away from the view point, the sampling rate of diverging viewing rays falls below the sampling rate of the volume grid. To avoid aliasing artifacts, the EWA volume resampling filter has to rely on strong prefiltering to get rid of high frequency components in the volume data. In this case, the shape and the size of the EWA resampling filter is dominated by the low-pass filter. Hence, approximating the EWA resampling filter with a low-pass filter alone will avoid the expensive EWA computation without degrading the rendering quality much. In the other extreme case when the volume data is very close to the view point, the sampling rate of diverging viewing rays is higher than that of the original volume grid. The low-pass filter, though not a dominant component in the resampling filter, can degrade the rendering quality with unnecessary blurring. Approximating the EWA resampling filter with the dominant reconstruction filter (i.e., without the convolution with the low-pass filter) not only reduces the computation cost but also yields better rendering quality. However, in the transition between the two extremes, neither approximation can avoid aliasing artifacts without time-consuming EWA computation. As a consequence, in our adaptive EWA splatting approach we classify each volume particle into one of the above three cases during rendering. This allows more efficient computation of footprint functions whereas preserving high image quality of EWA splatting (Figure 3).

We now present a distance-dependent classification criteria for adaptive EWA volume splatting based on a careful analysis of the EWA volume resampling filter (Equation 4). The 2×2 variance matrix $\mathbf{M}_k(\mathbf{x})$ (Equation 5) determines the final footprint's size and shape, which can be described mathematically as an ellipse. Because $\mathbf{W}$, $\mathbf{V}''_k$ and $\mathbf{V}_h$ are the same for all voxels in one view, the footprint of each voxel depends only on the Jacobian $\mathbf{J}_{\mathbf{u}_k}$. Suppose that $\mathbf{V}''_k$ is symmetric and the cutoff radius (see Equation 2) of the reconstruction and the low-pass kernels are r_k and r_h respectively, then $\hat{\mathbf{V}}_k$ is symmetric and the minor and major radius of the ellipse can be derived from Equation 4:

$$r_0 = \sqrt{\frac{r_k^2}{u_{k_2}^2} + r_h{}^2}, \quad r_1 = \sqrt{\frac{r_k{}^2(u_{k_0}^2+u_{k_1}^2+u_{k_2}^2)}{u_{k_2}^4} + r_h{}^2} \tag{6}$$

Not surprisingly, the depth of a voxel in camera space u_{k_2} (Figure 3) largely determines the ellipse radii as can be seen in Equation 6. Remember that the distance between the viewpoint and the image plane is 1.0 (see section 3). It can be shown that u_{k_0}/u_{k_2} and u_{k_1}/u_{k_2} range from $-\tan(fov/2)$ to $\tan(fov/2)$ given fov is the view angle. Hence the maximum value of $(u_{k_0}^2 + u_{k_1}^2 + u_{k_2}^2)/u_{k_2}^2$ is $(1.0 + 2.0 \times \tan(fov/2)^2)$. Therefore, a conservative distance dependent adaptive criteria can be determined by considering u_{k_2} only. To compute r_0, r_h can be discarded given that r_k/u_{k_2} is much larger than r_h. In this case, r_h can be skipped for the computation of r_1, too. On the other hand, if $r_k \times (1.0 + 2.0 \times \tan(fov/2)^2)/u_{k_2}$ is much smaller than r_h, r_1 can be approximated by r_h and so does r_0. From the above analysis we derive the following distance-dependent adaptive EWA filtering formula[2] with two controlling parameters c_{min} and c_{max}:

$$\begin{pmatrix} \mathbf{H}_k(\mathbf{x}) = \mathbf{x}^T \cdot \hat{\mathbf{V}}_k^{-1} \cdot \mathbf{x}, & if\ u_{k_2} < \frac{r_k}{r_h} \times c_{min} \\ \mathbf{H}_k(\mathbf{x}) = \mathbf{x}^T \cdot \mathbf{V}_h^{-1} \cdot \mathbf{x}, & if\ u_{k_2} > \frac{r_k}{r_h} \times c_{max} \\ \mathbf{H}_k(\mathbf{x}) = \mathbf{x} \cdot (\hat{\mathbf{V}}_k + \mathbf{V}_h)^{-1} \cdot \mathbf{x}, & otherwise \end{pmatrix} \tag{7}$$

Based on the above criteria, adaptive EWA volume splatting determines the appropriate resampling filter to be used for efficient interactive rendering as illustrated in Figure 3. Note that the parameters c_{min} and c_{max} can be adjusted to achieve the desired balance between efficiency and quality. For example, by slightly increasing c_{min} and decreasing c_{max}, adaptive EWA splatting becomes less conservative without affecting the image quality much.

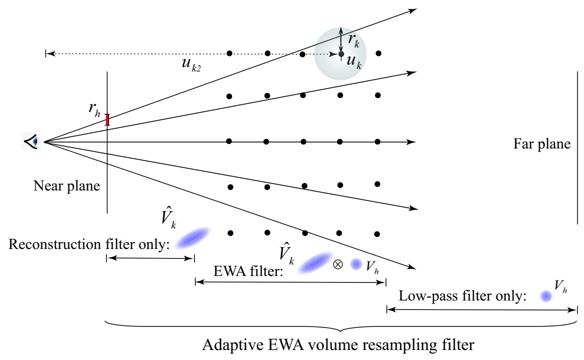


Figure 3: The distance-dependent adaptive EWA volume splatting scheme. The low-pass filter or the reconstruction filter can be chosen to replace EWA resampling filter in the extreme cases.

Adaptive EWA volume splatting can work with regular, rectilinear and irregular volume datasets. It can also be incorporated seamlessly in previous splatting systems. However, the downside of the approach is the additional computation required for the classification criteria of each footprint function. To reduce this cost, the volume data can be organized in patches or blocks in a spatial data structure. The filter criteria (Equation 7) is then conservatively evaluated on a per block basis and the same filter is applied to all voxels in a patch or block. Heuristic and cheap metrics can be used to speedup the calculation on-the-fly.

4 Hardware-Accelerated Framework

In this section we describe how we apply our adaptive EWA volume splatting approach in a hardware-accelerated splatting framework for the rendering of regular or rectilinear volume datasets. Our framework is based on an axis-aligned volume splatting scheme [30] with three traversal orders along the three major axes (Figure 4). During rendering, the voxels are processed in slices perpendicular to the major axis that is most parallel to the viewing

[2]For the nonsymmetric case, similar adaptive criteria can be derived.

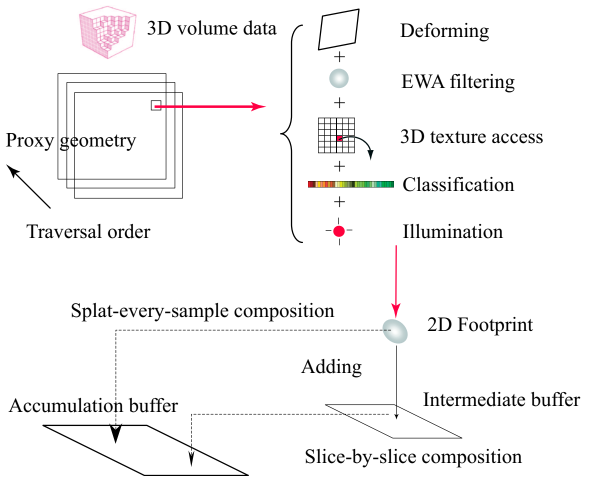


Figure 4: Hardware-accelerated EWA volume splatting. The process is based on axis-aligned scheme with three traversal orders. For simplicity, one traversal order is shown.

direction. A textured quad (*proxy geometry*) representing a splat is attached at each voxel. Splats are processed in two stages (Figure 4): In the first stage, the contributions from splats in one slice are added in an *intermediate buffer*. Then back-to-front or front-to-back composition of each slice in an *accumulation buffer* yields the final results. We call this composition method *slice-by-slice composition*. Alternatively, composition can be done directly without using intermediate buffer as shown in Figure 4. We call this simple composition method *splat-every-sample composition*.

Our hardware accelerated splatting pipeline features several innovations: First, it includes a hardware implementation of the adaptive EWA scheme, which is described in Section 4.1. This provides high image quality with better rendering performance than full EWA splatting. Further, the pipeline employs a retained-mode scheme that relies on proxy geometry compression as described in Section 4.2. This avoids the memory bandwidth bottleneck between CPU and GPU, which occurs in immediate mode algorithms. Finally, our pipeline includes a technique for interactive classification of voxels, which we describe in Section 4.3, and a fast two-pass shading method introduced in Section 4.4.

4.1 Adaptive Splat Computation

We embed adaptive EWA volume splatting in our framework using a patch-based classification scheme. In this scheme, the quads (i.e., voxels) of each slice are grouped into uniform rectangular patches. During splatting, we compute the camera space coordinate u_{k2} of each of the four corner voxels of the patch on-the-fly and evaluate the criteria given in Equation 7. If all four vertices meet the magnification criterion, the reconstruction filter is used as the footprint function. If all four vertices meet the minification criterion, the low-pass filter is used as the footprint function. Otherwise, the full EWA resampling filter is applied. Following the analysis in Section 3, we choose c_{min} and c_{max} in Equation 7 as 0.3 and $2.0 \times (1.0 + 2.0 \times \tan(fov/2)^2)$ respectively for all examples shown in the paper.

Our splatting process relies on proxy geometry, i.e., textured quads, as rendering primitives that represent the footprint functions. The texture on each quad encodes a 2D unit Gaussian function. Note that the geometry of the quad has to be deformed to stretch and scale the unit Gaussian texture, such that its projection to the image plane matches the footprint function. This is achieved using programmable vertex shaders as described by Ren and colleagues [23]. In particular, they explained how to derive the geometry of the quad by analyzing the EWA resampling filter (Equation 4). In our adaptive scheme, we implemented three different vertex shaders for each of the cases in Equation 7 and chose the appropriate one based on our per-patch evaluation described above. During the rasterization of the proxy quads, the filter weight, the color and the illumination components of each pixel are computed based on voxel attributes in the volume texture.

4.2 Proxy Geometry Compression

In our approach, volume data and its proxy geometry are stored locally in graphics hardware. This configuration avoids heavy bandwidth consumption between CPU and GPU and allows interactive rendering in programmable graphics hardware. However, a naive implementation has huge memory requirements. Let us take a 256^3 volume data set as an example. The scalar density value of each voxel usually takes one byte and the gradient vector takes three bytes with each of its three components quantized to 8 bits. Hence, we pack the attributes of each voxel into four bytes and save the 256^3 volume data as a volume texture of 64M bytes.

On the other hand, we use a quad (4 vertices) as proxy geometry for each voxel, so the whole volume data requires 64 million vertices. Each vertex contains its position (3 floating-point numbers, 12 bytes), volume texture coordinates (3 floating-point numbers, 12 bytes) and texture coordinates for splatting with the Gaussian texture (2 floating-point numbers, 8 bytes), resulting in a total of 32 bytes. Moreover, to specify the connectivity of a quad (i.e., 2 triangles) from 4 vertices, additional 6 vertex indices are needed. Each index is stored as a two or four byte integer, depending on the total number of vertices. With the three traversal orders for axis-aligned splatting, we store the indices for each quad three times. Using two bytes for each index, the proxy geometry of a voxel takes 164 bytes, hence the whole dataset requires as much as 2240M bytes of memory. Unfortunately, commodity graphics hardware currently provides a maximum of 256M local memory.

Facing these huge memory requirements, previous splatting approaches resorted to immediate-mode rendering, sending each quad separately to the rendering pipeline. This solves the memory problem at the cost of huge bandwidth consumption between CPU and GPU. In contrast, our approach employs a proxy geometry compression scheme that allows to store the volume data locally in graphics memory. Fast decompression is performed on-the-fly in the vertex shader.

4.2.1 Efficient Compression

We exploit the regularity of rectilinear or regular volumes to reduce the size of proxy geometry. First, the position of each vertex can be omitted because it can be calculated from the volume texture coordinates. Second, one slice of proxy geometry can be shared by all slices of the volume because the difference of the volume texture

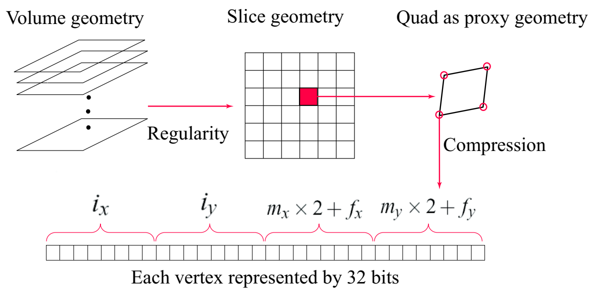


Figure 5: Proxy geometry compression.

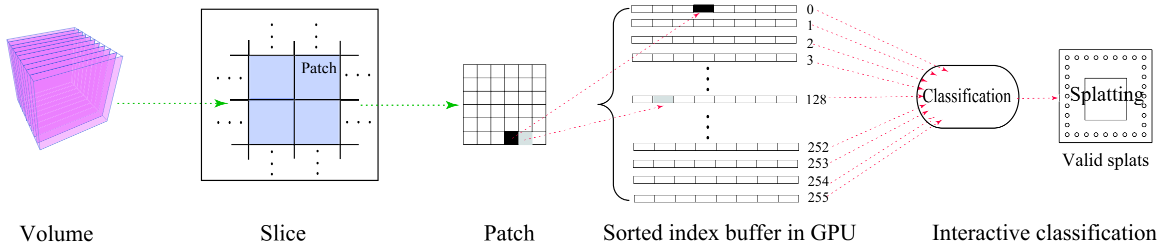


Figure 6: Auxiliary data structure for interactive classification. To reduce the size of the index buffers, the size of each patch is chosen to contain no more than 65,535 vertices so that two instead of four bytes can be used as an index.

coordinates between consecutive slices is constant. This one slice of proxy geometry is called the *proxy geometry template*. Third, within a slice the volume texture coordinate along the traversal direction does not change. Hence, only two texture coordinates are needed. Moving from one slice to the next along the traversal direction simply requires to update the constant texture coordinate. We maintain three proxy geometry templates for the three traversal orders, hence for the 256^3 volume dataset only 768k instead of 192M vertices are needed. However, for each vertex we still need to store two components of the volume texture coordinate and the Gaussian texture coordinates, totally 16 bytes per vertex.

Efficient encoding of vertex attributes is performed as follows: The two volume texture coordinates are denoted by t_x and t_y. They are stored as integers of the form $t_x = m_x \times 256 + i_x$, where $m_x = t_x/256$ and $i_x = t_x \; mod \; 256$, and analogous for t_y. Although i_x and i_y require 8 bits, m_x and m_y are stored using 7 bits only. This allows slices as large as $2^{15} \times 2^{15}$ voxels. Two additional bits f_x and f_y per vertex are used to specify the coordinates of the 2D Gaussian texture applied to the quad. As described in detail in [23], f_x and f_y are either zero or one. Hence, each vertex of the proxy geometry can be packed into 32 bits (Figure 5).

With these compression techniques, a 256^3 volume dataset requires only 12M $(256 \times 256 \times 4 \times 4 \times 3)$ instead of 2048M memory for the vertices, which corresponds to a compression ratio of 171:1 compared to the naive approach. Further, because for most volume datasets only $5\% - 20\%$ non-transparent voxels need to be rendered, the vertex indices will not require more than 115.2M bytes in most cases. As a result, the volume texture data and the packed proxy geometry information can be pre-loaded in the local memory of graphics hardware entirely for interactive rendering.

4.2.2 Fast Decompression

Recovering the volume texture coordinates and Gaussian texture coordinates from the packed representation is the first step in the processing of each vertex. We extract m_x, m_y, f_x and f_y from the packed representation using a small lookup table (our Direct3D implementation stores this lookup table in *constant registers* and accesses it through an *address register* in the vertex shader[12]). The decompression, the recovery of different kinds of texture coordinates, and the further calculation of voxel center positions require only a few operations that can be performed efficiently with programmable vertex shaders in graphics hardware.

4.3 Interactive Classification

Interactive classification is critical for interactive volume exploration. *Post-classification* schemes, where all the voxels are rendered and classification is performed on-the-fly, are not efficient because no culling of transparent voxels can be performed. Unnecessary rendering of transparent voxels can be avoided using *pre-classification*. Here, only non-transparent voxels are rendered, which typically account only for $5\% - 20\%$ of all voxels. However, in our retained-mode hardware implementation pre-classification requires collecting the indices of the vertices for those non-transparent voxels and loading the index buffer to the graphics hardware whenever the transfer function is changed. Because the construction of the index buffer is time-consuming and loading it to graphics memory involves a significant amount of data transfer between CPU and GPU, changes of the transfer function cannot be visualized at interactive rates.

We solve this problem by constructing an auxiliary data structure as illustrated in Figure 6. The basic idea follows the list-based splatting algorithm proposed by Mueller and colleagues [16]. We first bucket sort the voxels based on their density values. In contrast to [16], who built iso-value lists for the whole volume, we compute them for each slice of each traversal order. The indices of the corresponding proxy geometry are sorted accordingly and rearranged into iso-value index buffers. The index buffers, which are pre-loaded in graphics hardware before interactive rendering, span 256 intervals of iso-value voxels. Practically, the index buffers are placed in video memory or AGP memory, depending on their sizes. Putting them in AGP memory does not affect the performance as shown in our video demonstration. Pointers to the start of each iso-value array are maintained in main memory. When the transfer function is interactively changed, appropriate pointers are collected and merged quickly to send visible voxels to the rendering pipeline.

4.4 Fast Two-Pass Shading

Per-fragment lighting has to be applied for splat illumination because current graphics hardware allows volume texture access only in the fragment, but not in the vertex processing stage. However, this makes per-fragment processing expensive, since it involves volume texture access, voxel classification, and lighting computation for each pixel per splat. Each splat may cover between 2×2 and as many as 40×40 pixels, depending on the viewpoint.

We observe that pixels covered by one splat can share intermediate results, such as access to the volume texture, classification (via lookup table), and illumination computations. We propose a two-pass shading scheme to avoid redundant computation. In the first pass, voxels of a slice are projected as single-pixel points as shown in Figure 7a. The results of volume texture access, voxel classification, and illumination for each (point) splat are stored in a render target (also known as P-buffer). These intermediate results are then reused during lookup in the second pass, in which the full splat with per-fragment lighting and EWA filtering is projected (see Figure 7b). The final composition of all splats from each slice is shown in Figure 7c. In practice, this two-pass shading improves rendering performance by $5\% - 10\%$ though the additional rendering pass increases the number of context switches.

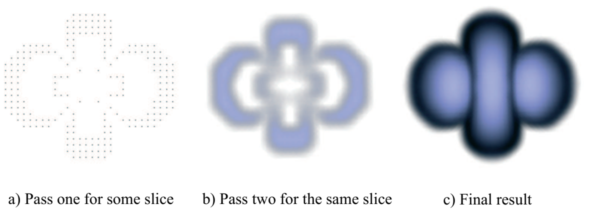


Figure 7: Two-pass shading for a test dataset (32^3).

5 Results

We have implemented our algorithm with DirectX 9.0 SDK [12] under Windows XP. Performance has been measured on a 2.4 GHz P4 system with 2 GB memory and an ATI 9800 Pro video card (256 MB local memory).

Data Type	EWA	Reconstruction	Low-pass
Regular	70	45	26
Rectilinear	74	45	26

Table 1: Numbers of vertex shader instructions for different filters.

The adaptive EWA volume resampling filter is implemented using three different vertex shaders. The main efficiency improvement of adaptive EWA filtering over full EWA resampling arises from the simplified computation in case of minification or magnification. Table 1 reports the numbers of vertex shader instructions required to implement the different filters for regular and rectilinear datasets. Note that we also maintain two different implementations for regular and rectilinear datasets, respectively.

We first demonstrate the efficiency and the quality of adaptive EWA volume splatting by rendering a $512 \times 512 \times 3$ checkerboard dataset with all voxels classified as non-transparent. In Figure 8a and Figure 8b we compare the image quality of splatting with the reconstruction filter only with that of splatting with the EWA volume resampling filter. We also compare the image quality of splatting with the low-pass filter only with that of splatting with the EWA volume resampling filter in Figure 8c and Figure 8d.

These comparisons show that splatting with improper filters, though more efficient as shown in Table 2, can result in aliasing artifacts and holes. Splatting with the EWA volume resampling filter corrects those errors, however at a high computational effort. Adaptive EWA splatting yields an image quality comparable to that of full EWA filtering, as shown in Figure 8e-h, but at reduced computational cost, as reported in Table 2.

Reconstruction (Figure8a) 6.25 fps	EWA (Figure8b) 4.97 fps
Low-pass (Figure8c) 6.14 fps	EWA (Figure8d) 3.79 fps
Adaptive EWA (Figure8e) 1.84 fps	EWA (Figure8f) 1.75 fps
Adaptive EWA (Figure8g) 6.88 fps	EWA (Figure8h) 4.83 fps

Table 2: Performance comparison in fps for checkerboard data set.

Figure 1 shows adaptive EWA splatting of a number of volume datasets. Based on our hardware-accelerated splatting framework, we compare the performance of the adaptive EWA splatting scheme with that of the previous EWA volume splatting method in Table 3. The performance improvement achieved by adaptive EWA filtering is about 10% to 20%.

We compare our retained-mode rendering approach with proxy geometry compression to a naive immediate mode implementation, where each splat is sent to the graphics pipeline separately. The results of the comparison are reported in Table 4 for various data sets using EWA volume splatting. The results clearly indicate that the CPU-to-GPU memory bandwidth is a bottleneck in this scenario, and our retained mode rendering approach leads to significant performance improvements.

Data	Rendered splat	Immediate	Retained
Bonsai	274866	0.53 fps	7.53 fps
Engine	247577	1.40 fps	10.28 fps
Checkerboard	786432	0.47 fps	6.88 fps
Lobster	555976	1.19 fps	10.60 fps
UNC Head	2955242	0.12 fps	2.86 fps

Table 4: Performance comparison in fps between immediate and retained rendering modes for adaptive EWA volume splatting.

We also compare the efficiency of pre-classification with list-based pre-classification. Table 5 shows the average classification time for various data sets. When the transfer function is unchanged, the rendering speed of list-based pre-classification is a little slower than pre-classification. However, our list-based pre-classification achieves much better performance during interactive transfer function changes as shown in Table 5.

Data	List-based Pre-classification	Pre-classification
UNC Head	2.80 fps	0.3 fps
Engine	10.18 fps	0.8 fps
Bonsai	7.23 fps	0.8 fps
Lobster	10.30 fps	1.1 fps

Table 5: Performance comparison in fps between list-based pre-classification and traditional pre-classification modes.

In Figure 9, we show a series of splatting results of the UNC head dataset with different transfer functions. We compare splat-every-sample composition (Figure 9a) with our slice-by-slice (or *sheet buffer*) splatting (Figure 9b) [30]. The images generated by splat-every-sample composition are darker than those of slice-based composition because of incorrect visibility determination. On the other hand, the splat-every-sample composition achieves better performance because it avoids per-slice additions.

It is well known that slice-based compositing introduces popping artifacts when the slice direction suddenly changes. Mueller and Crawfis [13] proposed to use an image-aligned kernel-slicing and traversal order. But their method has high computational cost and its GPU implementation needs to be investigated in the future. Note that our algorithm and almost all of the GPU implementation is independent of the compositing method that is used.

6 Conclusions and Future Work

We have presented a hardware-accelerated adaptive EWA splatting approach for direct volume rendering. Adaptive EWA volume splatting yields high quality aliasing-free images at a smaller computation cost than full EWA splatting. We embedded the adaptive EWA splatting scheme in a hardware-accelerated volume splatting framework whose key features include efficient proxy geometry compression and fast decompression, support for interactive transfer function design, and fast two-pass shading.

In the future, we want to make our current hardware-accelerated framework more efficient. Several researchers demonstrate that EWA point splatting with OpenGL point primitives can significantly improve the performance by reducing the size of the proxy geometry and processing time for each splat [2, 4]. We plan to use a similar technique for adaptive EWA volume splatting. We also want to apply our adaptive EWA splatting scheme to irregular volume datasets in a hardware-accelerated framework. Finally, we are interested in developing a hardware-accelerated adaptive EWA volume splatting framework with image-aligned traversal order to avoid popping artifacts and implement post-classification.

Data	Resolution	Slice thickness	Rendered splats	Adaptive	EWA	Improvement
Engine	$256 \times 256 \times 110$	$1.0\times1.0\times1.0$	594430 (8.25%)	3.25 fps	2.90 fps	12%
Bonsai	$256\times 256 \times 128$	$0.586\times0.586\times 1.0$	274866 (3.28%)	7.53 fps	6.90 fps	9%
Lobster	$301\times 324\times 56$	$1.0\times1.0\times1.4$	177720 (31.7%)	10.60 fps	8.75 fps	21%
UNC Head	$208\times 256\times 225$	$1.0\times1.0\times1.0$	693032 (5.78%)	3.00 fps	2.70 fps	11%

Table 3: Performance comparison in fps between adaptive EWA splatting and EWA splatting.

Acknowledgments

The first author is partially supported by 973 program of China (No.2002CB312100), NSF of China for Innovative Research Groups (Grant No.60021201), NSF of China (Grant No.60103017,60303028) and ZPNSSFYSC (No.R603046). The second author is supported by School of Computer Science Alumni Fellowship from Carnegie Mellon University.

References

[1] C. P. Botha and F. H. Post. Shellsplatting: Interactive rendering of anisotropic volumes. In *Proceedings of 2003 Joint Eurographics - IEEE TCVG Symposium on Visualization*, May 2003.

[2] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *Proceedings of the 2003 Pacific Graphics Conference*, pages 335–343, 2003.

[3] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the 2001 ACM SIGGRAPH/Eurographics Workshop on Graphics hardware*, pages 9–16. ACM Press, 2001.

[4] G. Guennebaud and M. Paulin. Efficient screen space approach for hardware accelerated surfel rendering. In *Proceedings of the 2003 Vision, Modeling and Visualization Conference*, pages 1–10, Munich, Germany, 19-21 November 2003.

[5] P. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, June 1989.

[6] J. Huang, R. Crawfis, and D. Stredney. Edge preservation in volume rendering using splatting. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 63–69, NC, USA, 1998.

[7] J. Huang, K. Mueller, N. Shareef, and R. Crawfis. Fastsplats: Optimized splatting on rectilinear grids. In *Proceedings of the 2000 IEEE Visualization Conference*, pages 219–227, USA, October 2000.

[8] I. Ihm and R. K. Lee. On enhancing the speed of splatting with indexing. In *Proceedings of the 1995 IEEE Visualization Conference*, pages 69–76, 1995.

[9] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proceedings of ACM SIGGRAPH 1991*, pages 285–288, Las Vegas, NV, USA, August 1991.

[10] L. Lippert and M. Gross. Fast wavelet based volume rendering by accumulation of transparent texture maps. In *Proceedings of Eurographics 1995*, pages 431–443, September 1995.

[11] X. Mao. Splatting of non rectilinear volumes through stochastic resampling. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):156–170, 1996.

[12] Microsoft Corporation. *DirectX 9.0 SDK*, December 2002.

[13] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *Proceedings of the 1998 IEEE Visualization Conference*, pages 239–246, Ottawa, Canada, October 1998.

[14] K. Mueller, T. Moeller, and R. Crawfis. Splatting without the blur. In *Proceedings of the 1999 IEEE Visualization Conference*, pages 363–370, San Francisco, CA, USA, October 1999.

[15] K. Mueller, T. Moeller, J.E. Swan, R. Crawfis, N. Shareef, and R. Yagel. Splatting errors and antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):178–191, April-June 1998.

[16] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[17] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by utilizing a ray driven approach. In *Proceedings of the 1996 IEEE Visualization Conference*, pages 65–72, Ottawa, Canada, October 1996.

[18] N. Neophytou and K. Mueller. Post-convolved splatting. In *Proceedings of the symposium on Data visualisation 2003*, pages 223–230. Eurographics Association, 2003.

[19] J. Orchard and T. Mueller. Accelerated splatting using a 3d adjacency data structure. In *Proceedings of Graphics Interface 2001*, pages 191–200, Ottawa, Canada, September 2001.

[20] H. Pfister. *The Visualization Handbook*, chapter Hardware-Accelerated Volume Rendering. Chris Johnson and Chuck Hansen (Editors),Academic Press, 2004.

[21] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Proceedings of ACM SIGGRAPH 1999*, pages 251–260. ACM Press, 1999.

[22] R.Crawfis and N.Max. Texture splats for 3d scalar and vector field visualization. In *Proceedings of the 1993 IEEE Visualization Conference*, pages 261–266, 1993.

[23] L. Ren, H. Pfister, and M. Zwicker. Object-space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics 2002*, pages 461–470, September 2002.

[24] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM Press, 2000.

[25] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Eurographics/IEEE TCVG Symposium on Visualization 2003*, 2003.

[26] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, Phoenix, AZ, USA, July 2000.

[27] J. E. Swan, K. Mueller, T. Möller, N. Shareef, R. Crawfis, and R. Yagel. An anti-aliasing technique for splatting. In *Proceedings of the 1997 IEEE Visualization Conference*, pages 197–204, Phoenix, AZ, October 1997.

[28] T. Welsh and K. Mueller. A frequency-sensitive point hierarchy for images and volumes. In *Proceedings of the 2003 IEEE Visualization Conference*, Seattle, USA, October 2003.

[29] L. Westover. Interactive volume rendering. In C. Upson, editor, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 9–16, Chapel Hill, NC, May 1989.

[30] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of ACM SIGGRAPH 1990*, pages 367–376, August 1990.

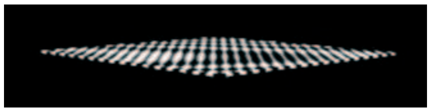
[31] Y. Wu, V. Bhatia, H. C. Lauer, and L. Seiler. Shear-image order ray casting volume rendering. In *ACM Symposium on Interactive 3D Graphics*, pages 152–162, Monterey, CA, June 2003.

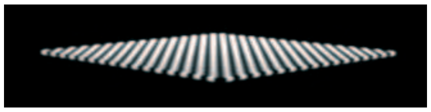
[32] D. Xue and R. Crawfis. Efficient splatting using modern graphics hardware. *Journal of Graphics Tools*, 8(3):1–21, 2004.

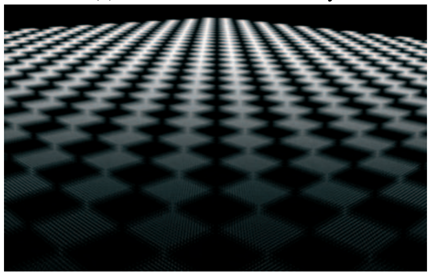
[33] M. Zwicker, H. Pfister., J. Van Baar, and M. Gross. Surface splatting. In *Proceedings of ACM SIGGRAPH 2001*, pages 371–378, Los Angeles, CA, July 2001.

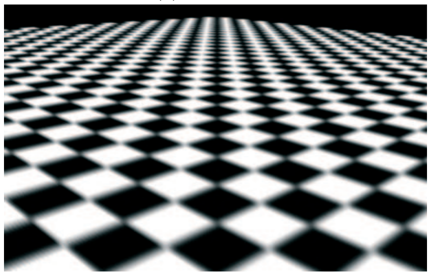
[34] M. Zwicker, H. Pfister, J. Van Baar, and M. Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

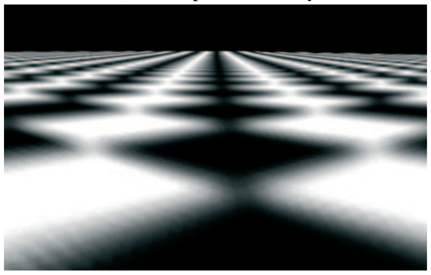
[35] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa volume splatting. *IEEE Visualization 2001*, pages 29–36, 2001.

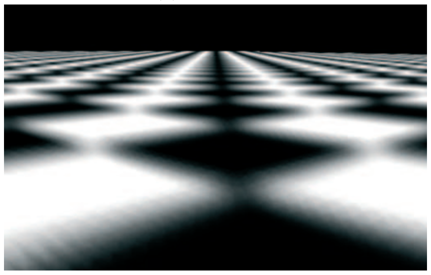

(a) Reconstruction filter only

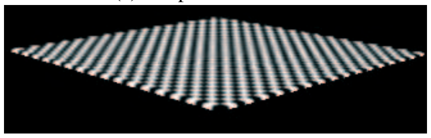

(b) EWA filter


(c) Low-pass filter only


(d) EWA filter


(e) Adaptive EWA filter


(f) EWA filter


(g) Adaptive EWA filter

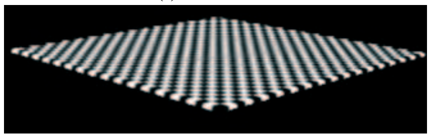
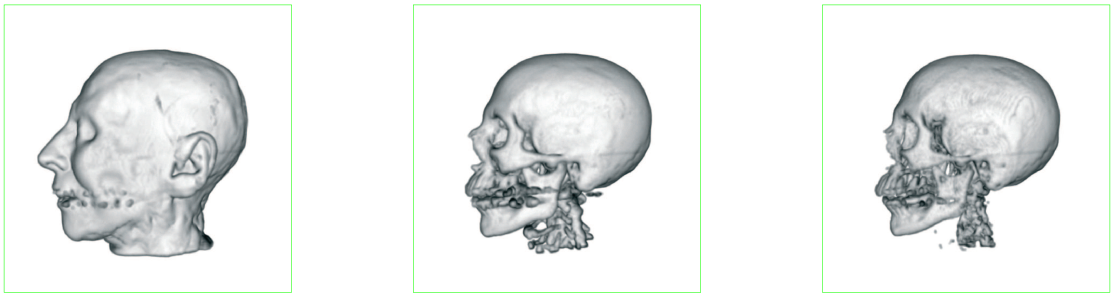

(h) EWA filter

Figure 8: Adaptive EWA splatting for checkerboard dataset with resolution of $512 \times 512 \times 3$. Figure (a-d) show that EWA filter is necessary. Figure (e-h) show adaptive EWA splatting leads to visually indistinguishable results.


(a) Splat-every-sample compositing mode. From left to right: 0.94 fps, 3.34 fps, 4.04 fps.

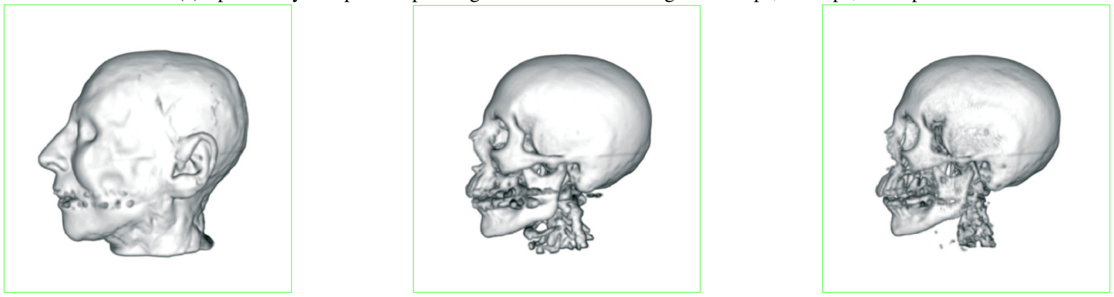

(b) List-based pre-classification / pre-classification modes. From left to right: 0.80 / 0.81 fps, 3.00 / 3.08 fps, 3.45 / 3.53 fps.

Figure 9: Comparison among splat-every-sample compositing, list-based pre-classification and pre-classification modes. From left to right: 2905251, 702768, 585682 splats.