Eurographics/ IEEE-VGTC Symposium on Visualization (2006)
Thomas Ertl, Ken Joy, and Beatriz Santos (Editors)

GPU-Accelerated Volume Splatting With Elliptical RBFs

Neophytos Neophytou[1] Klaus Mueller[1] Kevin T. McDonnell[2] Wei Hong[1]
Xin Guan[1] Hong Qin[1] Arie Kaufman[1]

[1]Center for Visual Computing, Computer Science, Stony Brook University
[2]Department of Mathematics and Computer Science, Dowling College

Abstract

Radial Basis Functions (RBFs) have become a popular rendering primitive, both in surface and in volume rendering. This paper focuses on volume visualization, giving rise to 3D kernels. RBFs are especially convenient for the representation of scattered and irregularly distributed point samples, where the RBF kernel is used as a blending function for the space in between samples. Common representations employ radially symmetric RBFs, and various techniques have been introduced to render these, also with efficient implementations on programmable graphics hardware (GPUs). In this paper, we extend the existing work to more generalized, ellipsoidal RBF kernels, for the rendering of scattered volume data. We devise a post-shaded kernel-centric rendering approach, specifically designed to run efficiently on GPUs, and we demonstrate our renderer using datasets from subdivision volumes and computational science.

Categories and Subject Descriptors (according to ACM CSS): I.3.3 [Computer Graphics]: Display Algorithms

1. Introduction

The representation of a volume dataset as a collection of partially overlapping 3D radial basis functions (RBFs, simply called kernels in those days), centered at the grid positions, gave rise to one of the first volume rendering algorithms -- splatting [Wes90]. While early splatting approaches suffered from blurring as well as popping artifacts during view transitions, the introduction of post-rasterization classification and shading [MMC99], in conjunction with the image-aligned kernel-slicing approach [MSH*99], enabled volume rendering with splatting at high quality in both static and dynamic viewing modes. A hallmark of splatting is its object-centric (or better, voxel-centric) rendering paradigm, which enables the efficient and rendering of sparse and irregularly-shaped datasets [MHB*00][HHF*05]. Splatting is also a popular paradigm for surface rendering with point-based representations, giving rise to point-based rendering [LW85][PZvB*00] [RL00]. Here, surfaces are represented as collections of surface-aligned 2D RBFs which are rasterized to the screen to form the image. Just as in volume splatting, surface-splatting with points allows a more efficient representation and rendering of intricate objects with high, but possibly sparse detail. In terms of volume rendering, if all one desires are iso-surfaces, then a favorable approach is to convert a volume into a set of surface points on the fly and render these with surface splatting [LT04][vRLH*04]. This eliminates the need for reconstructing the surface in volume space by interpolating the local neighborhood of 3D RBFs. On the other hand, if the goal is to create a composited or summed integration of the volume data, say, along the viewing direction, then it is preferable to retain the volumetric RBF representation during the rendering. Our paper focuses on this latter task.

Point-based objects rendered with RBF surface splatting are commonly created from dense point clouds, which were acquired via range scanning. To find the RBF-based surface, some optimization method is used, such as least squares and others [DTS01][OBH04]. More recently, researchers have also used methods of this kind to reduce volumetric datasets originated from computational science, such as CFD or finite element simulation, into a smaller set of RBFs [JWH*04] [WBH*05], and a scattered volumetric point cloud results from this RBF-fitting process. On the other hand, volumetric scanning, using Lidar and others, also gives rise to scattered volumetric point clouds [JRS*02].

The original volume splatting algorithm was motivated by medical data, which come on cartesian grids. The rendering algorithms for these could rely on this regular structure and rasterize and composite the splats in very regular ways. Scattered data, on the other hand, requires more complicated space traversal, rasterization, and compositing strategies. Most methods simply order the RBFs and then rasterize them as single, pre-shaded blobs, projecting their screen space footprints (usually a Gaussian). However, pre-shading and -classification leads to blurring on zooms, and it also can cause the leakage of color. A better method, just as in the splat rendering of regular data, is to reconstruct a set of parallel, image-aligned density slices of the volume first, and then perform color lookup and shading, followed by a merging with a front-to-back or back-to-front compositing buffer. The method presented in this paper follows this high-quality paradigm.

In that respect, our work is most similar to that of Jang et al. [JWH*04], who use an octree for spatially organizing the RBFs and then pass a set of equi-distant slice planes across this decomposition. The octree helps to quickly locate the RBFs which intersect with the slice. Then, for each slice plane pixel -- forming a point P_{sl} in volume space -- they employ a fragment program on the GPU to evaluate the exponential Gaussian functions of the RBFs overlapping at P_{sl}, plugging in the volume-space distance of P_{sl} and each RBF center. They report frame-rates of 4 fps for grids with less than 1000 kernels. Although excellent images can be obtained, the explicit object-space interpolation approach suffers from high memory and computational overhead, since each point may have to be loaded and its kernel function be evaluated a number of times

©The Eurographics Association 2006.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

per slice, once for each slice pixel. This in turn limits the attainable rendering frame rate. We take the opposite strategy, more in line with point-based rendering, which spreads a volume point's contributions to all affected pixels in a slice with a single load and requires only a few kernel evaluations. More specifically, the contributions of our paper are:

- Instead of using expensive fragment programs at each slice pixel, we rasterize kernel sections, taking advantage of the much faster hard-wired floating-point polygonal rasterization facilities provided by the latest graphics cards, such as the Nvidia 6800 FX. This enables us to achieve splat rendering rates of two orders of magnitudes higher than with an explicit kernel function evaluation approach.
- In addition to spherical RBFs we also support generalized ellipsoidal kernels, using a novel GPU-based scalable kernel slicing method. Elliptical RBFs potentially enable a better, more data-aligned fitting, which can reduce the number of RBFs and also lower fitting errors resulting from the optimization. While we do not explore this potential in this paper, there have been a number of results in point-based surface rendering which substantiate this claim [DTS01].
- Finally, our method also uses a spatial decomposition (but a flat organization, not an octree), in conjunction with an active splat list, to quickly locate the kernels falling into a slice slab.

Our paper is organized as follows: In Section 2 we review some related previous works on the rendering of irregular grids, splatting, and GPU-accelerated rendering. Then, in Sections 3 and 4, we describe the theory and implementation of our framework. Section 5 presents some of our results, and in Section 6 we conclude and give directions for future work.

2. Related Work

Our paper deals with the point-based rendering of irregularly distributed RBFs in volumes. For a survey of point-based rendering methods for surfaces refer, for example, to [KB04]. In the scope of regular-grid splatting of pre-classified points, the EWA framework has been devised to separate the object-space scaling and filtering of 3D kernels from the image-space filtering of footprints [ZPvB*01]. Our framework could be enhanced in similar ways, and we will discuss ideas to this end in the future work section of this paper. The EWA algorithm has recently also been accelerated on the GPU [CRZ*04].

In the following, we shall concentrate on related work in volumetric RBF splatting of scattered and irregularly-gridded data. There have been various algorithms that use the splatting paradigm for these purposes. Meredith and Ma [MM01] use spherical kernels that fit into a cube and are mapped to textured squares for projection. Jang et al. [JRS*02] fill a cell with one or more ellipsoidal kernels, which they render with elliptical splats. A similar approach was also taken by Mao [Mao96]. Hopf et al. [HLE04] apply splatting to very large time-varying datasets and render the data as anti-aliased GL points. Common to all these approaches is that shading precedes splat projection because the overlap of the kernels in volume space makes it difficult to interpolate the local information, such as gradient and density, needed for the shading information. Doing so would be equivalent to a local reconstruction of the field function. The image-aligned splatting method [MSH*99], on the other hand, enables this by interpolating a set of parallel, image-aligned density slices, splatting the slices of the kernels they intersect (see also Section 3). This allows the mapping of pixel densities into color by ways of the transfer function, as well as shading by calculating the gradients within-slice (x, y) and across-slice (z). This, in turn, enables crisp edges on zooms and reduces the color leakage created by mixing pre-colored RBF "balls" on the image plane. It also resolves the ordering paradox resulting from the partial overlap of the kernels, which is even more pronounced in the irregular, scattered data case, where possibly large and small RBFs mix together.

As mentioned in Section 1, a slice-interpolating approach bearing these advantages was recently taken by Jang et al. [JWH04] for scattered data, but they use a GPU fragment program to evaluate the Gaussian function of all kernels that overlap at a given point. Their post-shaded approach leads to imagery with much better lighting effects, but it suffers from the overhead associated with the explicit kernel evaluation. Since they run an optimization algorithm to encode their volumes into a minimal set of spherical RBFs, the number of kernels they must render is not very large (not more than 1000), and therefore they can obtain interactive frame-rates of up to 4 fps. However, the reduction is highly dependent on the accuracy threshold set and the complexity of the original, unoptimized volume. It is therefore worthwhile to explore methods that can process points at a faster rate. In fact, by performing the local function reconstruction via analytic means in the fragment shaders, the explicit approach does not exploit the much faster hard-wired interpolation facilities that exist in GPUs, and which have been exploited by even early splatting approaches run on SGI hardware [CM93]. It turns out that by using the new floating blending capabilities of the latest generation of graphics cards, all splatting operations necessary can be executed within the hard-wired parallel rasterization and pixel processing units on these cards. We shall see that by using these facilities a significantly higher point rendering rate can be obtained, while still maintaining the high visual quality enabled by the slice-based post-shaded rendering.

3. Method

First, we briefly review the image-aligned splatting approach [MSH*99] and show the modifications required for the splatting of ellipsoids. We also summarize how it is accelerated on the GPU (see [NM05] for more detail). Then we move to its extension to elliptical RBF kernels.

3.1 The modified image-aligned splatting algorithm

Fig. 1 illustrates the algorithm in its modified form. For reasons which will become apparent at the end of this section, the slicing planes only extract slices of the kernel, and not pre-integrated kernel slabs, as was done in [MSH*99]. The algorithm only requires a single 2D generic Gaussian footprint texture (and not an array as in [MSH*99]) and a 1D texture with a Gaussian scaling function. This decomposition is enabled by the separability of higher-dimensional Gaussian functions into a set of products of lower-dimensional Gaussians.

$$G = Ae^{-k(x^2+y^2+z^2)} = Ae^{-kz^2}e^{-k(x^2+y^2)} \quad (1)$$

Here, the first part is the scaling function and the second is the splat. When a slicing plane cuts across a kernel, the 1D Gaussian scaling texture is indexed by the slice plane's dis-

©The Eurographics Association 2006.

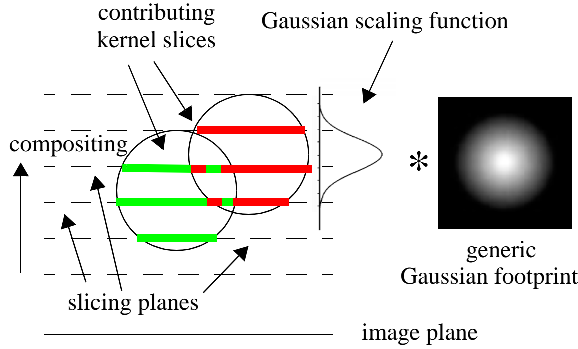

Figure 1: Modified image-aligned sheet-buffered splatting, as seen from the top. Kernels and planes extend to 3D. A viewing angle of 0° is shown, but any viewing angle is possible since the kernels are radially symmetric.

tance from the center of the kernel. The result is then used as a factor to scale (using the OpenGL functions) the footprint texture when it is rasterized. This yields a properly scaled rasterized kernel slice. Extracting kernel slices instead of slabs has not resulted in lower quality rendering for the cartesian-grid case. In essence, while the present method is a point-sampling of the slice function, the previous slab method pre-filters (box-averages) the kernel along the viewing direction (using the slab width as the box-filter support). Thus, the present method is similar to point-sampling in ray casting. To overcome possible staircase artifacts, etc., one could easily add pre-integrated rendering [EKE01], using the interpolated densities of a front and back slice as indices.

In [NM05], we have described a number of techniques that promote the use of built-in hardware facilities, such as early z-culling, (i) to eliminate kernels projecting onto opaque image areas at an early stage and (ii) to focus slice-based shading and compositing operation onto areas that have received kernel contributions. We extend the standard algorithm by generalizing the basic primitive of the spherical kernel, used to represent a point, to a general ellipsoid. An ellipsoid can be perceived and processed as a deformed (warped) sphere, since it can be represented by a series of 3D scaling and rotation operations applied to a unit sphere. By transforming all the slicing/compositing operations of the 3D ellipsoid to operations on a unit sphere, we can exploit the symmetry of the radial kernel and apply an efficient algorithm which is also easy to accelerate using current generation GPU hardware. Once transformed we use the method shown in Fig. 1 to extract the slice. Fig. 2 justifies our approach of splatting only the kernel slices and not the pre-integrated slabs.

3.2 Ellipsoid slicing

We shall now derive our warping/slicing procedure for ellipsoidal kernels more formally. Modeling the 3D elliptical Gaussian kernel using the implicit equation of a general ellipsoid (quadric surface) yields the following equation:

$$ax^2 + by^2 + cz^2 + 2dxy + 2zey + 2fxz + 2gx + 2hy + 2jz + k = 0 \quad (2)$$

Let us assume that the ellipsoid is centered at the origin *(0,0,0)* and its size is one. We can then let *g=h=j=0* and *k=-1*. In the matrix form of equation (3), the ellipsoid can now be expressed as a 4x4 quadric matrix Q, which is equivalent to applying an affine transformation to a unit sphere.

$$(x, y, z, 1)^T \cdot Q \cdot (x, y, z, 1) = 0 \quad (3)$$

$$Q = \begin{bmatrix} a & d & f & 0 \\ d & b & e & 0 \\ f & e & c & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = (M^{-1})^T \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \cdot M^{-1} \quad (4)$$

In the above notation, the quadric matrix of a unit sphere is represented as an identity matrix with the last element set to be *-1*, and it will be referred to as *I'*. An arbitrary ellipsoid Q can therefore be composed by scaling and then rotating a sphere around its center. Thus, given a scaling matrix S and a rotation matrix R, the resulting ellipsoid will be:

$$\begin{aligned} Q &= (M^{-1})^T \cdot I' \cdot M^{-1} \\ &= (R^{-1})^T \cdot (S^{-1})^T \cdot I' \cdot (S^{-1}) \cdot (R^{-1}) \end{aligned} \quad (5)$$

Since R is orthogonal, and S is diagonal we have $R^{-1} = R^T$ and $S^T = S$, and therefore:

$$Q = R \cdot S^{-1} \cdot I' \cdot S^{-1} \cdot R^T \quad (6)$$

For rendering, each ellipsoid needs to be transformed into screen space by first positioning it into volume space (applying a translation T to its center), and then applying the viewing transformation matrix V to the matrix Q. Here, $V = S_V \cdot R_V$, where the matrices S_V, R_V are the zoom (scale) and rotation matrices, respectively. The screen-space scaling S_V will be applied as a 2D scaling transformation to the ellipsoid slices just before the rasterization takes place. In the remainder of the text, we define the term *screen space* to be the coordinate system aligned with the screen before the final viewport transformation is applied, sometimes also referred to as *eye-space*.

Using V and T to transform the ellipsoid into screen space, we are now ready to slice it for rasterization into the corresponding slicing plane buffers. For this, we would intersect it

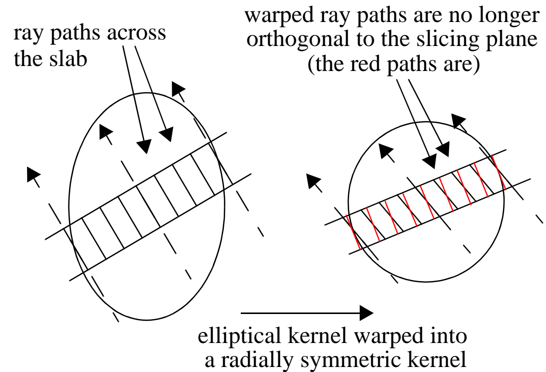

Figure 2: Illustration of the kernel warping process in 2D. In the original image-aligned splatting algorithm, the kernel slice tables would store the pre-integration of the rays across the slab, traversing it orthogonal to the slab's two bounding planes. However, the warping undoes this orthogonality. The integrals due to the red rays would be looked up and not those due to the warped black rays. This could lead to slight errors, which are different for each slab position. Therefore we do not store the slab integrals, but only use a cross-sectional texture of the radially symmetric RBF (we use Gaussians), which is then scaled by the orthogonal kernel value.

©The Eurographics Association 2006.

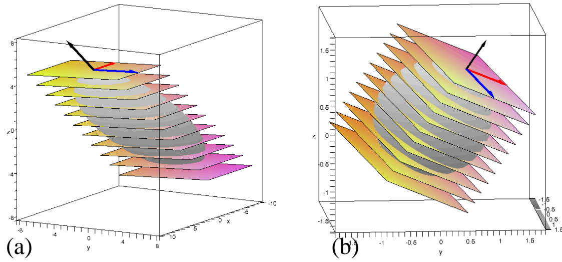

Figure 3: Ellipsoidal kernel sliced (a) in screen space, (b) in unit texture space (*unit sphere space*). The *u,v,n* vectors are shown in red, blue, and black respectively. These 3 vectors, along with the size of the direction vector *n* provide enough information to create all the intersecting slices of the ellipsoid in screen space (progressing along the viewing direction) and evaluate the corresponding Gaussian kernel by texture-mapping the 2D slices from the unit sphere space.

with a group of equidistant planes perpendicular to the viewing direction. The kernel slices would then be added to the corresponding slicing buffers. Fig. 3a illustrates an arbitrary ellipsoid with a set of slicing planes along the viewing direction. However, since the kernel slices and the orthogonal scaling are orientation-dependent, we would require a separate generic footprint and scaling function for each possible screen space orientation of this individual ellipsoid, and for each orientation of the collection of ellipsoids in a given view.

We can overcome these obstacles by warping the slicing planes from the space of a given ellipsoidal kernel into the space of a standard spherical kernel of unit size. By expressing an ellipsoidal kernel as a transformation applied to a spherical kernel, we can take also advantage of the hard-wired GPU texture-mapping and blending functionality to achieve a very fast mapping. While an ellipsoid will be sliced in screen space, resulting in a set of stacked 2D ellipses, the kernel evaluation will be done by applying the corresponding texture coordinates of the slicing polygons to a 3D spherical kernel in unit texture space. Thus these ellipses will be treated as deformed disks, or, more specifically, as slices of a spherical kernel. Fig. 3b illustrates the corresponding texture coordinate space, where the intersecting polygons are now slicing a unit sphere. Further optimizations are then applied to take advantage of the fact that the Gaussian kernel being used is spherically symmetric. We will now describe our slicing technique in further detail.

3.3 Efficient ellipsoid slicing on the GPU

The obvious approach to ellipsoid slicing would deal directly with equation (2) of the general ellipsoid, and its quadric representation in equation (4). To create a 2D ellipse one only needs to drop the last two rows and columns of the matrix Q. This approach was used for EWA splatting [ZPvB*01], which, however, splats entire kernels without slicing. Incorporating slicing is possible, but we discovered that it requires many operations, including square roots, that are not well accelerated on the GPU hardware. Although they could be issued in a fragment program, we prefer the faster hard-wired vector and matrix operations the hardware provides. Further, such a method requires conditional statements, which also pose challenges and are currently not recommended for wide use by the GPU manufacturers. In the following paragraphs, we describe an approach which performs the slicing task with fewer computations and uses more vector and matrix operations which are better accelerated on the GPU. For our own research, we have used the results of the other, analytical approach to evaluate the correctness of our method. Unfortunately, the space restrictions do not permit us to include both solutions in this paper.

For the following discussion, consider the transformation pipeline $E = T_V \cdot S_V \cdot R_V \cdot T \cdot R \cdot S$, where the final screen space ellipsoid is expressed as the result of a series of transformations to be applied to a unit sphere, positioned at the origin. Reading from right to left (multiplication order), S and R are the scaling and orientation matrixes, which define the shape and orientation of each ellipsoid, while T positions the ellipsoid in volume space, by applying a translation from the origin. The matrix *TRS* defines the ellipsoid returned from an optimization [JWH*04][WBH*05], local fitting, or modeling procedure [GM04][MCQ04]. Matrices R_V, S_V and T_V are the decomposition of the viewing transformation and they encode a rotation, scaling and translation of the volume. The scaling and translation are applied during the rasterization stage, so our slicing pipeline uses $M = R_V \cdot T \cdot R \cdot S$, which encodes all the transformations that lead to a unscaled view transformed ellipsoid. Therefore, by multiplying the view transformed ellipsoid quadric Q with M^{-1} we would obtain a unit sphere:

$$
\begin{aligned}
Q' &= \left((M^{-1})^{-1}\right)^T \cdot Q \cdot (M^{-1})^{-1} \\
&= M^T \cdot \left((M^{-1})^T \cdot I' \cdot M^{-1}\right) \cdot M = I'
\end{aligned}
\tag{7}
$$

On the other hand, when the same transformation, M^{-1}, is applied to the set of intersecting polygons which slice the ellipsoid, we end up with a set of parallel polygons that slice a unit sphere. Note that these slicing planes in unit sphere space are not necessarily axis-aligned, as one can observe from Fig. 3b. But this difference in slice orientation is not relevant, since the kernel is radially symmetric. The notable and crucial difference, however, is that in unit sphere space the number and distance of slice planes has changed, due to the warp. Since our plan is to set up an incremental procedure for: (i) the retrieval of consecutive kernel slices, and (ii) their rasterization to the screen, we need to know four sets of vectors:

- the vector n_{us} connecting the centers of adjacent slices in unit sphere space (the black vector in Fig. 3b): this will allow us to compute the increment in the 1D texture storing the Gaussian scaling function.
- the vector n_{ss} connecting the centers of adjacent slices in screen space (the black vector in Fig. 3a): this will allow us to incrementally position the extracted kernel slice texture polygon in the sheet buffer for rasterization.
- the slice texture polygon orientation vectors u_{ss}, v_{ss} in screen space (the blue and red vectors in Fig. 3a): these are also needed to position the kernel slice texture polygons -- they define $slice_{ss}$, the polygon that maps the slice texture onto the sheet buffer, as defined in equations (13) and (15).
- the slice texture orientation vectors u_{us}, v_{us} in unit sphere space (blue and red vectors in Fig. 3b): they define $slice_{us}$, the polygon that maps the slice texture in the unit sphere.

Multiplying all the vertices of $slice_{us}$ and n_{us} by matrix M yields $slice_{ss}$ and n_{ss}.

©The Eurographics Association 2006.

Inside the GPU, each ellipsoid is expressed using 13 float values. These include the scaling and rotation matrices, S and R, and the position and scalar value of the point (density, $X, Y, Z, S_x, S_y, S_z, R_1, R_2, R_3, R_4, R_5, R_6$). We only store 6 elements to represent the 3x3 rotation matrix R, since it is orthogonal and the last 3 elements can be derived using a cross-product operation of the first two lines of the matrix. We use the translation, rotation and scaling matrices to compute M and M^{-1}.

$$M = R_V \cdot \begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

The key element to our approach is that all slicing operations on the ellipsoid are computed in the ellipsoid's coordinate system, with the ellipsoid centered at (0,0,0). We define matrix *M'*, which brings the screen space ellipsoid to the ellipsoid's coordinate system by cancelling the translation as shown in equation (9). The inverse of matrix *M'* is also very easy to compute, since R and R_V are both orthogonal.

$$M' = R_V \cdot T^{-1} \cdot R_V^{-1} \cdot M = R_V \cdot T^{-1} \cdot R_V^{-1} \cdot R_V \cdot T \cdot R \cdot S \tag{9}$$

$$M' = R_V \cdot R \cdot S$$

$$M'^{-1} = S^{-1} \cdot R^{-1} \cdot R_V^{-1}$$

Using *M'*, we first compute the slicing planes in unit texture space by applying the inverse transform M'^{-1} to the coordinate system vectors of the screen aligned slicing planes. The resulting vectors, *u,v,n* in unit sphere space are the first three columns of M'^{-1}:

$$[u, v, n] = M'^{-1} \tag{10}$$

These vectors are no longer orthogonal, since S is not orthonormal in general, and therefore we must re-orthogonalize them, resulting in *u',v',n'*.

$$n' = norm(u \times v), u' = norm(n' \times v), v' = (n' \times u') \tag{11}$$

Here, the corrected-normalized vector *n'* is the direction of the axis passing through the center of all slicing planes and through the center of the sphere in unit texture space (along the black vector in Fig. 3b), while the corrected normalized vectors *u'* and *v'* define the actual orientation of the slicing planes. The real length of *n'* (which has been normalized to a unit vector) can be recovered by computing its dot product with the original vector *n*. The resulting vector will become:

$$n'' = \|n \bullet n'\| \cdot n' \tag{12}$$

Thus, *n"* is the normal vector for all slicing planes in the unit sphere space and its length is the distance between any two slicing planes. This vector may be used to advance the current slicing plane for the kernel to the next slicing plane by adding *n"* to each of the corner vertices. Therefore, *u'* and *v'* can be used to define the slicing polygon as the set of vertices:

$$slice_{us} = \{-u' - v', -u' + v', u' + v', u' - v'\} \tag{13}$$

and the normal vector along the slicing axis is defined as:

$$n_{us} = n'' \tag{14}$$

Now, to obtain the corresponding slicing polygons and normal vector in screen space, we need to multiply the above set with the original transformation matrix M. Thus,

$$slice_{ss} = M \cdot slice_{us} \tag{15}$$

and the corresponding normal vector is

$$n_{ss} = M \cdot n_{us} \tag{16}$$

Note that the sampling distance in screen space is usually defined as $\Delta z = 1.0$ (but any value can be chosen to achieve tighter or sparser sampling along the viewing axis). Thus,

$$n_{ss} = (dx, dy, 1.0) \tag{17}$$

where *(dx, dy)* is the screen-space vector that can be used to incrementally position consecutive kernel slice polygons on the sheet buffer screen. Using (12), the unit sphere space slice distance vector is:

$$n_{us} = (0, 0, \|n''\|) \tag{18}$$

where $\|n''\|$ can be used to index the Gaussian scaling function. Since the Gaussian in kernel space is a radially symmetric function, we can choose to slice it along the z direction, which also reduces $slice_{us}$ to:

$$slice_{us} = \{(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 0, 0)\} \tag{19}$$

We can now set up an incremental kernel slicing procedure which only needs to perform the (screen space - unit sphere space) transformation once for the initial kernel mapping and can use simple vector additions to obtain the remaining mappings. The setup cost for each ellipsoid is not large, requiring two matrix-matrix multiplications to obtain $M'^{-1} = S^{-1} \cdot R^{-1} \cdot R_V^{-1}$ three cross-products, two normalizations, one dot product, four matrix-vector multiplications, and a few scalar-vector multiplications. The incremental slicing just involves four vector additions and a scalar addition. More on this next.

4. Implementation

Our current implementation extends the existing GPU-accelerated framework for image-aligned post-shaded splatting of regular volumetric datasets [NM05] to scattered data points, represented either as ellipsoidal or as spherical kernels. This existing GPU-accelerated image-aligned splatter achieves very high quality images even for sparse datasets at high magnifications and interactive frame-rates.

Extending this GPU accelerated algorithm, however, to render unstructured data sets that consist of arbitrarily oriented general ellipsoids of various sizes poses several challenges. Splatting is an object order approach which employs front-to-back compositing (for occlusion culling). Either a sorting/bucketing operation is required whenever the viewpoint changes, or one may use a spatial decomposition data structure which will facilitate a front-to-back traversal of the dataset during rendering. Both of these solutions are not trivial to implement on the GPU and they impose several trade-offs. These trade-offs involve the amount of data that has to be transferred from the CPU during every frame, versus the amount of data that will be replicated and processed multiple times directly on the GPU. We have addressed the visibility and occlusion issue using a flat cubic data-structure (yet the extension to an octree would be possible) in combination with a set of vertex and fragment programs that are used to emulate dynamic behaviors which would otherwise be taken for granted on the random access memory model of the CPU. The variability of the data is also addressed by this collection of vertex and fragment programs, which define a set of multiple-attribute arrays. The data points are first processed as textures taking advantage of the parallelism employed by the GPU at the fragment level to screen-align and slice all of the relevant ellipsoids (those that have non-zero opacity and color after transfer function indexing, and their immediate neighbors that

©The Eurographics Association 2006.

usually have slightly lower densities and were only barely rejected by this test). This is an operation that consumes only little overhead, but we also note that it is conceivable, for larger volumes, to divide the space into sub-volumes and perform the entire process in front-to-back order, but only on sub-volumes that are still visible after compositing the ones ahead of them. These textures are then passed on to the vertex processor as vertex attribute arrays, where they are iteratively processed (rasterized) as the slicing operation proceeds through the volume.

Similar to [NM05], we employ an elaborate mechanism which exploits the early z-culling hardware optimization in order to eliminate extraneous splatting onto already opaque regions (early splat elimination). A similar mechanism, which also takes advantage of early z-culling, is used to restrict the expensive shading and compositing operations only to fragments of the current sheet-buffer that have been updated by splatting. This efficient hardware-accelerated fragment level method provides an alternative to more complex CPU based methods that use tiles or quadtree based structures.

Finally, in order to achieve correct results, we have adjusted the shading/compositing process to normalize the contents of the sheet-buffer during reconstruction, and we also take special considerations at the volume boundaries. In the following subsections we will describe our implementation in further detail.

4.1 Ellipsoid slicing and rendering on the GPU

The slicing of all the input points in the system is done by a fragment program which processes all the initial input data as a set of textures. We have seen in Section 3 that 13 floating point values are required for an ellipsoid. These are packed in a set of 3 textures and are permanently resident in GPU memory. To fit all values, we "pack" the Z and Y elements of the translation matrix T together. The output of the slicing calculation also returns 12 values for each ellipsoidal point. These include the starting position for the first intersecting slice in screen-space, (p_x, p_y, p_z) which is the center of the slice, the 2D u and v vectors $(u_x,u_y),(v_x,v_y)$, which define the axis-aligned slicing polygon, as explained in Section 3, and the $n_{ss} = (dx, dy, 1.0)$ vector which defines the difference of each slicing polygon to the next. To slice the kernel in unit texture space, the vector $n_{us} = (0, 0, \|n''\|)$ is provided, and the starting position is given by $(0, 0, StartZ_{us})$. Finally, the scalar value of the data point is provided as a single float. The range for the scalar density value is [0...1], which gives some room to encode more data into the density input value, as we shall see in the remainder of this section. These 12 values are encoded in a set of 3 textures. The processing of all the data points is performed in one pass and the results are simultaneously assigned to the 3 destination textures using the multiple render targets (MRT) extension, which is available on both the latest NVidia as well as ATI boards. The measured timings for this step were around 30-90 msec, for datasets of sizes varying from 35K-500K points.

The slicing step has to take place once every time the view point changes. When all points are processed they are converted into vertex arrays within the graphics board memory, and they are then assigned as vertex attribute arrays and passed to the splatting vertex program. The vertex program maintains the slicing polygon for each data point via the use of a uniform status variable that is passed to the program for all vertices. The currentSlicingZ parameter, which is passed to all the vertices, holds the z value of the current image aligned slicing plane. Thus, for every point, the vertex program first decides whether it does intersect this slice, and if so, it computes the position of each of its corners and the corresponding texture coordinates. Otherwise it forces the point to be culled by setting the coordinates outside of the view frustum.

The polygons slicing general ellipsoids have to be rendered using the GL_QUAD primitive, since point sprites are not applicable in this case. Our system follows the standard method used for rendering billboards with vertex programs. This method requires that the point data is replicated once for every destination vertex, and an additional piece of information is provided in the vertex data to decide which one of the polygon corners it represents. We employ this approach, and encode the polygon identity information using the density value. During the construction of the textures, we have *density = (vertexCount%4) * 1000 + inputScalarDensity.* The vertex program decides the texture coordinates as well as the corner positioning for a given vertex.

The third set of parameters define the slicing of the 3D spherical kernel in 3D unit texture space. This operation is implemented using the texture mapping of a 2D texture encoded Gaussian kernel, modulated by a 1D Gaussian along the z-direction. The 1D Gaussian value is currently computed inside the vertex shader, since vertex textures are still slower.

After all of the kernel contributions for the corresponding slice have been accumulated, the reconstructed density slice is passed to a fragment program which performs classification, shading and composites the results to the final frame-buffer image. In order to achieve correct reconstruction of unstructured data, a normalization step needs to be performed on the reconstructed densities. This is done by maintaining the splatted weights contributed by all sliced points in a separate channel of the slicing buffer. During normalization, the fragment program divides the densities by their corresponding weight. Special care has to be taken at the boundary regions of the volume, where coverage is sparse, and therefore the accumulated weights are smaller than one. Dividing by the small weight values would give rise to unnaturally high density values at the boundaries. In this implementation, a threshold is set before the division is performed. Thus, the resulting densities are:

$$normalizedDensity = (density)/(max(1, weight)) \quad (20)$$

An alternative solution to the boundary problem when splatting irregular grids would be to add a set of "ghost points" around the boundaries, which will have zero density, but normal weight values. This follows the fact that regular volumes implicitly have infinite coverage throughout the volume. That is, there exist empty points which have weights, but zero values wherever there is empty space in the regular volume, including the boundary regions. Normalization, however, is not required in regular volumes because everything would have to be divided by the same weight which can be integrated in the kernel texture.

4.2 GPU-support for spatial decomposition

In this work we address the spatial decomposition requirement imposed by the image aligned splatting algorithm by using a flat decomposition data structure in combination with additional functionality in the vertex programs that process each point. The volume is decomposed into a low-resolution

©The Eurographics Association 2006.

cubic block data structure, which can then be traversed in a front-to-back order in the exact same way as a full octree.

The spatial decomposition structure is first created on the CPU in a preprocessing step just after the volume is loaded. Every point is sliced and rasterized onto a very low resolution 3D grid, where each grid point maintains a list of all the points that intersect it. The result is a 3D regular cubic structure, which when traversed in a front-to-back order it will return a list of data points for each of its non-empty cells. The structure is then flattened onto a set of four 2D textures by copying the contents of every cell, in the format described in Section 4.1. The resulting texture will be processed by the ellipsoid slicing fragment program, resulting in a set of attribute vertex arrays for rendering, every time the viewpoint changes.

During rendering, each active cell's associated vertex list will be called using the *glDrawArrays* OpenGL call. The cells are placed in the active list in a front-to-back traversal of the decomposition data structure, and they are taken out of the active list when they stop producing slicing polygons, which is when the *currentSliceZ* has passed even the largest point, and which can be determined by a comparison with the maximum point diagonal for the cell.

The main challenge with maintaining such a structure on the GPU during rendering is the handling of splats that span to more than one cells, and are therefore being called for rendering in multiple lists. In the CPU scenario, a flag could have easily been set in the data-structure to signify that a point has already been called by another list and should not be considered active. In our implementation we emulate the tagging behavior by using uniform variables that pass the current state to the vertex program. Along with the current slicing plane, we pass the current cell ID as a uniform variable. During the slicing step, for each data point, we also compute the ID of the starting cell. This is the cell where the first slicing polygon for the point will be encountered, and it is easily computed by projecting $(start_x, start_y, start_z)$ to the 3D cubic decomposition structure. The point is then assigned that ID in all cells that it participates in. Then, in the rendering stage, we also compute the cell ID for each cell that is sliced by the current sheet buffer, and a point/vertex is only allowed to render if its starting cell ID is the same than that of the current cell, and if it also intersects the current slicing buffer. If it has a different ID, then it has already been activated before for rendering, and this new instantiation of the point can be is culled from the viewing frustum.

5. Results

We have tested our system using datasets from subdivision volumes [MCQ04] and computational science, both encoded as a field of irregularly distributed ellipsoidal RBFs. The RBF fields for the former set of volumes was obtained by fitting Gaussian kernels to a mid-level of the subdivision hierarchy, while the RBF field for the latter was obtained by fitting basis functions to a local Delauney triangulation [HNM*06]. This paper does not seek to determine error bounds on these -- it is merely concerned with achieving their fast rendering. The hardware configuration consists of a Pentium 4 running at 3GHz and 1 GB RAM, and the graphics board is an NVidia Quadro FX 3400 with 256MB RAM, which is equivalent to a GeForce 6800 GT board.

In order to test throughput, all the datasets were rendered at an image resolution of 400x400 pixels. Table 1 summarizes the results for each dataset, along with some description. The second column lists the number of ellipsoidal or spherical data points included in the dataset (the number in parentheses lists the number of non-occluded, splatted voxels, which are between 70-90% of the total number of voxels). As we render our volumes in semi-transparent mode, occlusion culling effects are less pronounced. The third column gives the range of the minimum and maximum diagonals of the enclosing cube for the ellipsoidal or spherical kernels, which is a measure of the maximum point sizes. This is actually the diagonal of the enclosing cube for an ellipsoidal or spherical kernel, and it gives a measure of the maximum size of the point. The fourth column shows our measured frame rate for these datasets at an image resolution of 400x400 and the fifth column directs to Fig. 4 (color-plate).

The first two datasets, “Blunt Fin” and “Combustion”, were created using the Delauney RBF fitting method, and they were composed using ellipsoidal points. The last four datasets were created from subdivision volumes, and they all encode deformed objects. All are rendered in semi-transparent compositing mode. Looking at the results, we find that the rendering of irregular data using the image aligned splatting algorithm can give a points/sec. throughput rate of up to two orders of magnitude higher than existing methods that use fragment-shader based kernel evaluations [JWH*04] [WBH*05] on the slice plane instead, with the same high-quality visual results that slice-based post-shaded methods produce (using the same GPU platform or of one generation past, respectively). Here, we only compare the frame rates for pure volume rendering (the system of [WBH*05] is much more versatile, also rendering and mixing in results from flow field analysis).

Our next observation is that although the elliptical datasets have a significantly smaller point count, they also are significantly slower to rasterize. This can be explained by the fact that the size (diagonal) of the ellipsoids used to compose these datasets affects a much larger range. After a closer look into the rasterization process, we have found that these datasets also have much larger overlaps between the neighboring points, resulting in significant overdraw.

Dataset	# Points	Diagonal	FPS	Fig.4
Blunt Fin	34k (31k)	0.9-120	1.6	a,b
Combust.	39k (35k)	8.3-27.4	1.9	c,d
Chess Pc.	84k (63k))	1.9-2.5	3.5	e
Monster	91k (60k)	2.5-3.0	4.0	f
Toy Car	173k (117k)	2.0-4.3	2.2	g
Toy Train	199k (139k)	2.0-4.0	2.1	h

Table 1: Results.

5.1 Comparison with fragment-centric approaches

The existing volume RBF splatting method [JWH*04] [WBH*05] uses a fragment-centric approach, evaluating all RBFs on a single fragment position before moving to the next, making it a gather-algorithm. Ours, on the other hand, is an

©The Eurographics Association 2006.

RBF-centric approach, that is, we process (rasterize) the slices of the entire RBF before moving to the next, making it a scatter-algorithm. Similar to other researchers, we have also found that it is often advantageous to rather evaluate a (modest) function in the fragment shader than incurring the overhead associated with accessing a lookup texture to retrieve the pre-computed result. We do this to obtain the footprint scaling values from the 1D Gaussian scaling function, and so does the fragment-centric approach of [JWH*04][WBH*05] which, however, needs to evaluate the entire 3D RBF function at each fragment, in addition to the overhead associated with computing the 3D index. Looking at the fragment-centric and RBF-centric approaches, we observe that for a given dataset both will perform the same number of sheet-buffer writes and will also need the same number of RBF function values. However, they differ greatly in terms of the efficiency at which these operations are performed, First, the RBF-centric approach only requires an evaluation of one 1D Gaussian per RBF-slice, using the inexpensive kernel rasterization to spread the function values across the sheet buffer fragments. The fragment-based approach, on the other hand, needs to evaluate the 3D Gaussian for each fragment, which is much more costly than a simple scaled texture mapping. Second, the texture rasterization of the RBF-centric approach is likely to be significantly more coherent, light-weight, and optimized on the GPU hardware than the heavier (in terms of data) texture fetches of the fragment-centric scheme. Third, each data fetch of the RBF-centric approach will amount to a rasterization, unless the fragment is killed early due to early fragment elimination. On the other hand, the fragment-centric approach, may sometimes pull in RBFs that are out of reach of the fragment's location, although this number may be low for good tree encoding. All of these considerations may explain the large difference in performance of the two approaches, especially when considering that our approach uses the considerably more data-intensive ellipsoidal RBFs.

6. Conclusions and Future Work

We have described a framework which extends the high-quality image-aligned sheet-buffer splatting method to irregular grids, and we have shown how to successfully accelerate it on the GPU. The performance of our implementation, in terms of splats/sec., is quite high, and this suggests that it may be preferable to use splatting for the slice-based post-shaded rendering of scattered point datasets and RBFs, instead of reconstructing the slice plane fragment values via explicit kernel-evaluations in fragment programs.

A possible future extension of our work could be the integration of EWA-type filtering in perspective viewing. For this, one may simply stretch the kernel slice textures as a function of slice distance to achieve the desired low-passing effect. This would use the zoom (scale) matrix S_V introduced in Section 3.2. Finally, future work will also be geared towards improving the fitting procedure for elliptical RBFs.

Acknowledgements

This research was supported, in part, by NSF CAREER grant ACI-0093157 and NIH grant 5R21EB004099-02.

References

[CM93] R. Crawfis and N. Max, "Texture splats for 3D scalar and field visualization," *IEEE Visualization 1993*.

[CRZ*04] W. Chen, L. Ren, M. Zwicker, H. Pfister, "Hardware-accelerated adaptive EWA volume splatting," *IEEE Visualization 2004*.

[DTS01] Q. Dinh, G. Turk, G. G. Slabaugh, "Reconstructing surfaces using anisotropic basis functions," ICCV 2001: 606-613.

[EKE01] K. Engel, M. Krauss and T. Ertl. "High-quality preintegrated volume rendering using hardware-accelerated pixel shading," *Proc. of Graphics Hardware Workshop 2001*.

[GM04] X. Guan, K. Mueller, "Point-based surface rendering with motion blur," *Symp. on Point-Based Graphics 2004*.

[HHF*05] F. Vega Higuera, P. Hastreiter, R. Fahlbusch, and G. Greiner, "High Performance Volume Splatting for Visualization of Neurovascular Data," *Proc. IEEE Visualization 2005.*

[HLE04] M. Hopf, M. Luttenberger, T. Ertl, "Hierarchical splatting of scattered 4D data," *IEEE Computer Graphics & Applications*, 24(4): 64-72, 2004.

[HNM*06] W. Hong, N. Neophytou, K. Mueller, and A. Kaufman "Constructing 3D Elliptical Gaussians for Irregular Data", *Mathematical Foundations of Scientific Visualization, Comp. Graphics, and Massive Data Exploration*, Springer-Verlag, Germany 2006.

[JWH*04] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. Ebert, K. Gaither, T. Ertl, "Interactively visualizing procedurally encoded scalar fields," *IEEE/EG Visualization Symp. 2004*.

[JRS*02] J. Jang, W. Ribarsky, C. D. Shaw, N. Faust, "View-dependent multiresolution splatting of non-uniform data," *Proc. IEEE/ EG Visualization Symp. 2002*.

[KB04] L. Kobbelt, M. Botsch, "A survey of point-based techniques in computer graphics," *Computers & Graphics*, 28(6):801-814, 2004.

[LT04] Y. Livnat, X. Tricoche, "Interactive point-based isosurface extraction," *IEEE Visualization 2004*.

[LW85] M. Levoy, T. Whitted "The use of points as a display primitive," *TR 85-022*, U. North Carolina, Chappel Hill, 1985.

[Mao96] X. Mao, "Splatting of curvilinear volumes," *IEEE Trans. Visualization and Computer Graphics*, 2(2): 156-170, 1996.

[MCQ04] K. McDonnell, Y. Chang, H. Qin, "Interpolatory, solid subdivision of unstructured hexahedral meshes," *The Visual Computer*, 20(6):418–436, 2004.

[MM01] J. Meredith, K.-L. Ma, "Multiresolution view-dependend splat based volume rendering of large irregular data," *Symp. Parallel and Large-Data Visualization and Graphics 2001*.

[MHB*00] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A practical comparison of popular volume rendering algorithms," *2000 Symp. on Volume Rendering*.

[MMC99] K. Mueller, T. Möller, and R. Crawfis, "Splatting without the blur," *Proc. IEEE Visualization*, 1999.

[MSH*99] K. Mueller, N. Shareef, J. Huang, R. Crawfis, "High-quality splatting on rectilinear grids with efficient culling of occluded voxels," *IEEE Trans. on Visualization and Computer Graphics*, 5(2): 116-134, 1999.

[NM05] N. Neophytou and K. Mueller, "GPU accelerated image aligned splatting," *IEEE-TCVG/EG Workshop on Volume Graphics 2005*.

[OBH04] Y. Ohtake, A. Belyaev, H. Seidel, "3D scattered data approximation with adaptive compactly supported radial basis functions," *Conf. Shape Modeling and Applications 2004*.

[PZvB*00] H. Pfister, M. Zwicker, J. van Baar, M. Gross, "Surfels: surface elements as rendering primitives," *Proc. Siggraph 2000*.

[RL00] S. Rusinkiewicz, M. Levoy, "Qsplat: a multiresolution point rendering system for large meshes," *Proc. Siggraph 2000*.

[vRLH*04] B. von Rymon-Lipinski, N. Hanssen, T. Jansen, L. Ritter, E. Keeve, "Efficient point-based isosurface exploration using the span-triangle," *Proc. IEEE Visualization* 2004.

[WBH*05] M. Weiler, R. P. Botchen, J. Huang and Y. Jang, "Hardware-assisted feature analysis and visualization of procedurally encoded multifield volumetric data," *IEEE Computer Graphics & Applications,* 25(5): 72-81, 2005.

[Wes90] L. Westover, "Footprint evaluation for volume rendering," *Proc. SIGGRAPH* 1990.

[ZPvB*01] M. Zwicker, H. Pfister, J. van Baar, M. Gross, "EWA Volume Splatting," *Proc. IEEE Visualization 2001*.

©The Eurographics Association 2006.

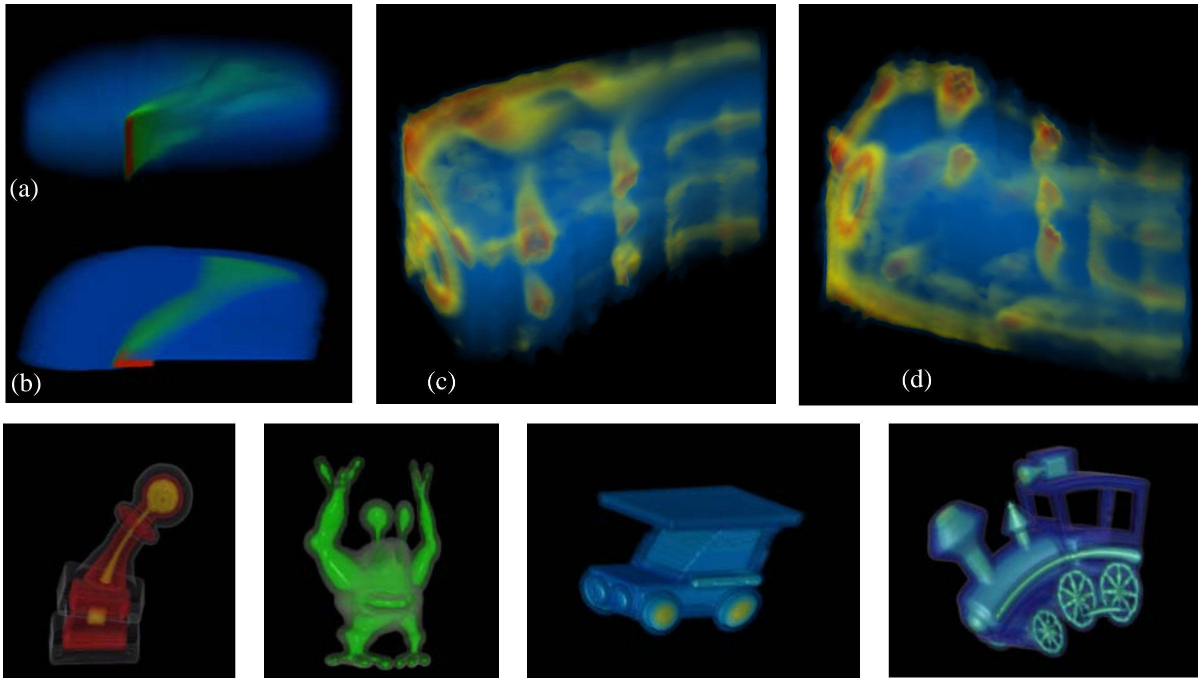

Figure 4: Example renderings of irregular datasets using our image aligned splattr: (a,b) two views of Blunt Fin, (c,d) two different views of the Combustion dataset, (e) deformed chess piece, (f) deformed green monster, (g) Deformed toy car, (h) deformed toy train.