CAR-TR-1022
CS-TR-4869
UMIACS-TR-2007-22

April 25, 2007

Modelling and Rendering Large Volume Data with Gaussian Radial Basis Functions

Derek Juba
Amitabh Varshney

University of Maryland
College Park, MD
{*juba, varshney*}@*cs.umd.edu*

Abstract

Implicit representations have the potential to represent large volumes succinctly. In this paper we present a multiresolution and progressive implicit representation of scalar volumetric data using anisotropic Gaussian radial basis functions (RBFs) defined over an octree. Our representation lends itself well to progressive level-of-detail representations. Our RBF encoding algorithm based on a Maximum Likelihood Estimation (MLE) calculation is non-iterative, scales in a $O(n \log n)$ manner, and operates in a memory-friendly manner on very large datasets by processing small blocks at a time. We also present a GPU-based ray-casting algorithm for direct rendering from implicit volumes. Our GPU-based implicit volume rendering algorithm is accelerated by early-ray termination and empty-space skipping for implicit volumes and can render volumes encoded with 16 million RBFs at 1 to 3 frames/second. The octree hierarchy enables the GPU-based ray-casting algorithm to efficiently traverse using location codes and is also suitable for view-dependent level-of-detail-based rendering.

1 Introduction

Scientific visualization is currently facing a grand challenge in coping with vast quantities of data arising from high-fidelity acquisitions and large-scale scientific simulations. Such datasets range from a few gigabytes to several terabytes and are often characterized by an imperative need for interactive exploratory visualization capabilities. For instance, the Richtmyer-Meshkov instability simulation performed at the Lawrence Livermore National Laboratory [23] has produced a $2048 \times 2048 \times 1920$ data over 273 time steps. The sheer size of this data makes it a challenge to visualize it on commodity graphics hardware.

We believe that implicit representations offer a powerful model for facilitating interactive visual exploration of large volumetric datasets. Implicit functions have been used for almost two decades in visual computing. Introduced as *blobby models* [1] and *metaballs* [27], they have grown to be widely used in games and movies. A nice overview of some of the early work in implicit surfaces can be found

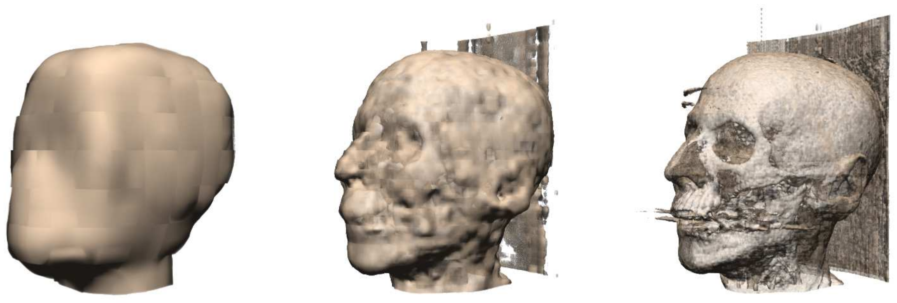

Figure 1: Renderings from three different levels of an RBF hierarchy for the UNC Head data set. From left to right, the renderings were generated using 4, 6, and 8 levels from the octree and consist of 561, 20.6 K, and 485 K RBFs.

in [4]. The use of implicit representations for volume modeling and rendering is a recent phenomenon. Implicit representations offer several advantages for volumes. First, their expressive power makes them well-suited for trading off memory accesses with computation. This is proving to be a powerful technique to hide memory latency for modern multi-core architectures. Second, the analytical formulation of implicit representations is highly amenable to local geometry processing, such as computing first and higher-order derivatives. Indeed, recent research has established the relationship of such geometric operators to features such as vortices and shocks by Weiler *et al.* [33] and the principled design of transfer functions by Kniss *et al.* [20, 21]. Third, volumetric implicit representations offer a multi-scale, and view-dependent capability to control visual detail in an intuitive manner. This is likely to have a direct implication in devising techniques for facilitating comprehension and reducing clutter in visual depictions.

In this paper we present an algorithm for succinctly representing and efficiently rendering large scalar volumetric data using a hierarchy of implicit functions based on anisotropic radial basis functions. The novel contributions of our work are:

1. A multiresolution representation using anisotropic radial basis functions that can encode a given volumetric dataset with progressively greater detail. Our representation is well-suited for time-critical rendering, progressive refinement, view-dependent level-of-detail, and progressive transmission.

2. An $O(n \log n)$-scalable fitting algorithm based on Maximum Likelihood Estimation that can rapidly fit large datasets in a memory-friendly manner. Specifically, we can fit 512^3 dataset in around 20 minutes with a 1.6% RMS error.

3. A GPU-based ray-casting algorithm that can efficiently and directly render from the hierarchical implicit representation of volumes. Our GPU-based ray-casting algorithm supports acceleration techniques such as empty-space skipping and early-ray termination with implicit volumes.

4. A multiresolution octree hierarchy over implicit RBFs that can leverage prior work on octree location codes for efficient ray-casting, obviates the need to store cell boundaries, and enables view-dependent level-of-detail rendering.

We review related work in section 2. We present our fitting algorithm in section 3 and our direct volume rendering algorithm in section 4. Finally, we conclude with some suggestions for future work in section 5.

2 Related Work

The basic goal of implicit function fitting is to fit a representation $f(x_i) = d_i, \quad i = 1, \ldots, n$, to surface or volumetric data, where d_i are appropriately chosen scalars, such that the level sets of the function f, given by the values d_i have some meaning associated either with the geometry or the properties of the object. When fitting an implicit function to the input data, a local form of the fitting expression is desirable, since the fit must adapt to local geometrical features. Radial basis functions (RBFs) have been shown to be a versatile fitting tool in various fields. A strong mathematical basis for their theory has been established and theorems showing accuracy in various normed spaces have been proven for both surface and volumetric data [6, 7, 10, 24, 30, 31, 32, 36]. These representations have several nice properties associated with sensitivity to local features, ability to support different levels of detail, ability to mitigate noise, ability to incorporate various degrees of smoothness and other priors via regularization and choice of particular RBFs (e.g., thin plate splines [9]). While these methods are considered to be accurate, their naive implementations are both expensive to fit, and subsequently evaluate, making them not very popular methods for fitting large datasets of the type we are dealing with in this paper.

Research by Beatson *et al.* [5] shows how RBF function fitting and evaluation of the fitted function can be sped up by an order of magnitude using the fast multipole method (FMM) [12]. They used non-compactly supported RBFs due to their interpolation and extrapolation properties. Morse *et al.* [24] were the first to fit surfaces using compactly-supported RBFs. Due to their local region of influence, compactly-supported RBFs can permit efficient evaluation and the ability to incorporate local changes to the fitted function without the need for the FMM data-structures. Co *et al.* [8] and Jang *et al.* [16] were the first to represent scattered and irregular volumetric scalar fields using RBFs. They used Principal Component Analysis (PCA) [17] to cluster and determine centers for the Gaussian RBFs and used the Levenberg-Marquardt optimization method to determine the Gaussian RBF variances. Weiler *et al.* [33] presented a k-d-tree-based method to fit Gaussian RBFs to an unstructured volumetric vector field. In addition to using PCA clustering, they select some of their RBF centers to be at peaks and troughs of low frequencies in the data. They also use an approximate iterative method to quickly solve the system of equations for the RBF weights. Hong *et al.* [14] use arbitrarily-oriented elliptical RBFs to fit data on an irregular grid. The RBF variances and orientations are chosen to match the Voronoi cells of the data points. PCA is used to create an initial guess for the RBF parameters, which is then refined using an iterative optimization algorithm. Jang *et al.* [15] give a method of fitting arbitrarily-oriented elliptical RBFs using non-linear optimization, and extend it to support vector data.

Rendering of implicit functions has a rich history in visual computing. Ray tracing of implicit surfaces has been reasonably well-studied [1, 13, 18, 29, 35]. An alternative to ray-tracing is to sample the implicit surface with a collection of well-distributed particles and then render such particles. Methods for carefully sampling the implicit surfaces have been discussed by Witkin and Heckbert [34] as well as by Turk and O'Brien [32]. Another approach involves converting the implicit functions to polygons by using marching cubes [22] or continuation methods [2, 3]. The polygons can then be rendered as in traditional graphics. Direct volume rendering of implicit functions is a relatively new endeavor. Jang *et al.* [16] and Weiler *et al.* [33] have developed GPU-based volume rendering algorithms that proceed in a slice-by-slice fashion. For each fragment in a slice, a fragment program iterates over RBF parameters stored in a texture, computes the scalar value at that location, and looks up the corresponding color from a 1D texture. Neophytou *et al.* [26] present a splatting-based GPU-accelerated volume rendering method for arbitrarily-oriented elliptical RBFs. Their method splats each RBF onto each intersecting slice as a textured polygon and accumulates the splats in a texture buffer. The slice can then be rendered after classification with a fragment program.

In this paper we present the first GPU-accelerated ray-casting algorithm for direct rendering of implicit volumes and the first octree-structured RBF representation of regular volume data that lends itself well to adaptive and progressive levels of detail.

3 Modelling

We model volume data as a sum of Radial Basis Functions (RBFs). Symbolically, this can be represented as:

$$f(x) = w_0 + \sum_{i=1}^{M} w_i\,\phi_i(x) \tag{1}$$

where	x	is the position vector of a point in the volume
	$f(x)$	is the scalar value at that point
	ϕ_i	is the *i*th RBF
	M	is the number of RBFs
	w_0	is a constant term
	w_i	is the weight of the *i*th RBF

In this work we have used Gaussian RBFs:

$$\phi_i(x) = e^{-r^2/2} \tag{2}$$

3.1 Multiresolution Hierarchical Fitting

Since we are targeting regular volumetric data, we can take advantage of the inherent structure of the data and fit the RBFs to a multi-resolution octree hierarchy, from low resolution to high. We start by fitting a single RBF to a 2^3 resolution downsampled version of the given dataset. This becomes the root of our RBF hierarchy. We then take a version of the dataset that has been downsampled to 4^3, evaluate the initial (root) RBF at each of the 4^3 data points, and compute the difference between the root RBF value and the downsampled version. The 4^3 block of residual data is then divided into eight blocks of 2^3 data values, and the entire process is repeated for each of these blocks. This is continued using versions of the data that have been downsampled to 8^3, 16^3, etc. (see fig. 2). Each time a new block is fit, the RBFs in that block and all of its ancestors are sampled at the locations of the data points in the full resolution data, and the current fitting error is computed. If the error is below a user-defined threshold, that block is not subdivided any further. Otherwise, the blocks will continue to be subdivided until they reach the full original data resolution.

This fitting algorithm only ever needs to access the full data in a sequential manner, during the error evaluation stage. All other operations are performed on one small, fixed-size block of data at a time, which will likely fit in a memory cache. Besides being memory-friendly, the overall fitting algorithm also scales in a $O(n\log n)$ manner.

In this method we truncate each RBF at the boundary of its octree cell. Due to fitting errors, this would cause noticeable discontinuities at the borders between octree cells during rendering. To avoid this, we fit each RBF to not only the data within its 2^3 block, but also to data within a 1 data point border around that block (each RBF is thus fitted to 4^3 data values). The fitting area of each block therefore overlaps half of each face-adjacent block, one quarter of each edge-adjacent block, and one eighth of each corner-adjacent block (see fig. 3 for an example on a quadtree). During rendering, the RBF values are blended together based on the distance from the center of each RBF's block (see section 4.3).

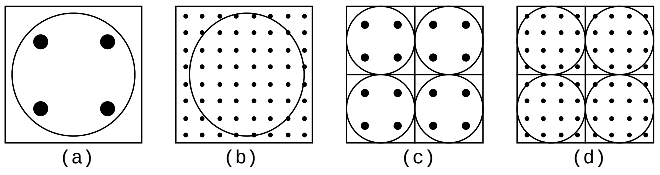

Figure 2: Overview of the fitting process. An RBF is first fit to low-resolution data (a), and the fitting error is evaluated at the full resolution (b). If the error is too high, additional RBFs are fit at a higher resolution (c), and their error evaluated (d). In the actual implementation the borders of each block would overlap neighboring blocks by one data point on each side, giving each block a resolution of 4^3 rather than 2^3 – this is omitted in this illustration for clarity.

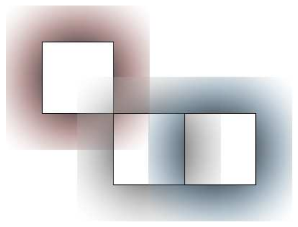

Figure 3: Blending between quadtree nodes. Only one edge-adjacent node (blue) and one corner-adjacent node (red) are shown. Sample points within the central block (grey) would be blended with samples from adjacent blocks whose shaded regions the points fell in.

3.2 Basic Algorithm

In the simplest version of the algorithm the RBF widths σ_i are identical. In this case, the RBF radius r is simply:

$$r = \|\frac{x - \mu_i}{\sigma_i}\| \tag{3}$$

	x	is the position vector of a point in the volume
where	μ_i	is the position vector of the ith RBF's center
	σ_i	defines the width of the i^{th} RBF.

Defining the RBF approximation is then only a matter of choosing a constant term w_0, RBF weights w_i, RBF centers μ_i, and RBF widths σ_i. Some results of this simple algorithm are shown in the first row of table 2. Some possible choices for the initial value of the constant term are zero, the minimum data value, or the mean of the data. In our experiments we found that setting the initial value to zero often produced the best results – this was done for all results reported in this work.

To compute the RBF centers and weights, the following simple algorithm can be used:

1. Perform a linear search over the data to find the data point with the maximum approximation error. This will be the location of the new RBF's center.

2. Set the RBF's weight to the error value at this cell. Since Gaussian RBFs have a value of 1 at their center, setting the weight in this way will cause the center data point to have zero error.

3. Update error values stored at each data point within the RBF's radius of influence.

4. Repeat this process until the desired error threshold is reached.

3.3 Anisotropic Width Selection

A natural extension to the basic algorithm is to allow the widths of the RBFs to vary anisotropically. This can be accomplished by making σ in equation 1 a vector and allowing it to take on different values for different dimensions, dividing it componentwise into the $x-\mu_i$ vector. General elliptical RBFs were used by Hong *et al.*[14], Jang *et al.*[15], and Neophytou *et al.*[26], although we restrict our RBFs to be axis-aligned, sacrificing some generality for a more compact representation. The effect of using anisotropic widths is illustrated in table 1.

The simplest way to select the RBF widths is by using a greedy algorithm. For each RBF, we first initialize the value of each $\sigma_{xi}, \sigma_{yi}, \sigma_{zi}$ to some small number (we use 0.3 times the width of a grid cell). We then begin iterating repeatedly over the x, y, z dimensions. For each dimension, we increment the corresponding σ_i by some small number (we use 0.1 times the width of a grid cell), and check if the average error in the RBF's area of effect was made worse by this change. If it was, the change is reverted, and that dimension is excluded from further iterations. Otherwise, the change is kept, and the iterations continued. Intuitively, this algorithm can be viewed as blowing up the RBF like a balloon inside a closed box, with the sides of the box representing the widths at which further inflation along that axis begins increasing the error.

Alternatively, a fast, non-iterative method of selecting the RBF widths is Maximum Likelihood Estimation (MLE)[19]. In this method, the data to be fit is treated as a histogram of samples from a Gaussian probability density function with the previously computed mean. The width σ of the probability density function that would give this set of samples the highest probability of being generated is then computed. This computation can be performed independently for each dimension, giving $\sigma_{xi}, \sigma_{yi}, \sigma_{zi}$.

The equations for computing the MLE width for the x dimension are:

$$\begin{array}{rcl} U_0 &=& \sum f \\ U_x &=& \sum x f \\ U_{xx} &=& \sum (x - U_x/U_0) f \\ \sigma_{xi}^2 &=& U_0 / 2U_{xx} \end{array} \tag{4}$$

where x is the x coordinate of the data point
f is the data value at that point

Other dimensions are computed similarly.

Since the residual data being fit can take on negative values, we must somehow convert this into a positive-valued histogram for the MLE computation. We do this by treating negative data values as zero when the RBF weight is positive, and the opposite when the RBF weight is negative (except then also taking the absolute value of the data values). The rationale for this is that we would like the RBF's region of influence to be restricted to an area in which the data has the same sign as the RBF, since these are the areas in which the RBF will decrease the fitting error rather than increase it. Setting values of the histogram to zero outside this area encourages probability distribution widths that do not produce many samples outside this area, and thus result in RBFs that do not have much influence outside this area.

A comparison between MLE width selection, iterative anisotropic width selection, and iterative isotropic width selection is given in table 1. The MLE method is clearly comparable in accuracy and yet is significantly faster to compute.

3.4 Weight selection

Once the RBF center and widths have been chosen, we compute the weights to minimize the sum of squared errors by forming the following system of linear equations, similar to [16, 33, 24]:

Data Set	Data Size	RBF Type	Width Selection	Fitting Time	Number of RBFs	RMS Error
UNC Head	256^3	Isotropic	Iterative	4.12 m	520 K	1.93%
UNC Head	256^3	Anisotropic	Iterative	9.48 m	455 K	1.50%
UNC Head	256^3	Anisotropic	MLE	2.25 m	485 K	1.75%
VHF Torso	512^3	Isotropic	Iterative	29.9 m	3.48 M	1.68%
VHF Torso	512^3	Anisotropic	Iterative	58.2 m	2.95 M	1.23%
VHF Torso	512^3	Anisotropic	MLE	19.1 m	3.22 M	1.60%
LLNL R-M	1024^3	Isotropic	Iterative	2.32 h	1.59 M	2.22%
LLNL R-M	1024^3	Anisotropic	Iterative	5.25 h	1.54 M	1.64%
LLNL R-M	1024^3	Anisotropic	MLE	1.42 h	1.60 M	1.85%

Table 1: Comparison of iterative fitting of isotropic RBFs, iterative fitting of anisotropic RBFs, and single-step fitting of anisotropic RBFs. See section 3.7 for a description of the data sets. The iterative fitting method can produce a somewhat better fit, but would take 3-4 times longer.

$$\begin{bmatrix} 1 & \phi_1(x_1) & \cdots & \phi_m(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(x_n) & \cdots & \phi_m(x_n) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \tag{5}$$

	m	is the number of RBFs
	n	is the number of data cells
	x_j	is the position of the jth data point in the volume
where	$f(x_j)$	is the data value at that point
	w_0	is the constant term
	w_i	is the weight of the ith RBF
	ϕ_i	is the ith RBF, with center μ_i and widths σ_i

If $m = n$, then this system can be solved exactly. Typically we will have $m < n$, so in general we compute the best solution in the least-squares sense through the use of singular-value decomposition.

3.5 Choice of the Basis Function

Gaussian RBFs (as in equation 2) have seen much use in prior work [8, 14, 15, 16, 26, 33], largely due to the smooth blending that occurs when these RBFs are brought in close proximity to each other. Since we truncate RBFs at octree cell boundaries and perform explicit blending between the cells, we do not benefit from this natural blending. However, there are still some desirable properties of this basis function. First, it decays to zero rather than going off to infinity, so if an octree cell is poorly fit by its RBF in some region, there is a limit to how much damage the bad RBF can do to the final, blended result. Second, the derivatives of Gaussian RBFs do not degenerate at higher orders, so these basis functions are useful if higher-order derivative information needs to be computed. Still, most of our work is not dependent on the particular choice of basis function, and investigation into other possible basis functions may prove useful.

3.6 Binary Representation

Once the RBF representation has been computed, we store it in the following format. The nodes of the octree from section 3.1 are stored in breadth-first order. Each node contains exactly one RBF and has either eight or zero children.

We first store a 4-byte integer giving the byte offset to the start of this node's children. We then store the maximum and minimum data values contained within the node as two 2-byte integers. The maximum and minimum values are packed into 2 bytes each by subtracting their minimum value, dividing by their range, and then multiplying by the maximum size of a 2-byte integer. We next store the constant term w_0 and RBF weight w_1, each as a 4-byte float. We then store the x, y, z components of the RBF mean, each as a 1-byte integer. The packing computation takes into account the node's size and position in the octree. For each component we subtract the lowest value it could take on in that node, divide by the width of the node, and then multiply by the maximum size of a 1-byte integer. Finally, we store the $\sigma_x, \sigma_y, \sigma_z$ width factors, each as a 4-byte float. This gives a total size of 31 bytes per node, which we pad to 32 bytes so that it fits evenly into two 16-byte texture look-ups. This packing is illustrated in figure 4.

At the start of the file we store a header giving the number of levels in the octree and the minimum values and ranges of the node maximums and minimums, to allow the packing to be undone.

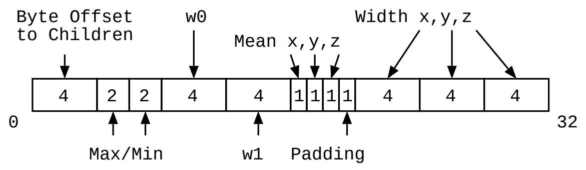

Figure 4: RBF and octree parameters packed into 32 bytes.

3.7 Fitting Results

Data Set	Data Size	File Size	Fitting Time	Number of RBFs	RBF File Size	RMS Error
UNC Head	128^3	4 MB	0.28 m	91.4 K	2.8 MB	2.22%
UNC Head	256^3	32 MB	1.90 m	485 K	15 MB	1.75%
VHF Torso	128^3	4 MB	0.38 m	109 K	3.4 MB	2.14%
VHF Torso	256^3	32 MB	2.10 m	520 K	16 MB	2.00%
VHF Torso	512^3	256 MB	16.2 m	3.22 M	99 MB	1.60%
LLNL R-M	128^3	2 MB	0.22 m	62.2 K	1.9 MB	5.24%
LLNL R-M	256^3	16 MB	1.52 m	402 K	13 MB	4.37%
LLNL R-M	512^3	128 MB	11.3 m	2.58 M	79 MB	3.31%
LLNL R-M	1024^3	1024 MB	85.3 m	16.0 M	489 MB	1.85%

Table 2: Fitting results for several different data sets using anositropic MLE-based RBF width selection. For comparison, the VHF Torso and LLNL R-M data sets have been downsampled to several different resolutions prior to fitting.

We have tested our RBF fitting implementation on three different data sets. The first (UNC Head) is a CT scan of a human head from the University of North Carolina. This data set originally consisted of 113 slices, with each slice being a 256^2 array of 16-bit integers (although only 12 bits of precision are actually used). We linearly interpolated additional slices between the existing ones and duplicated the final slice several times to produce a data set of size 256^3. In our tests we used this data set as well as a version downsampled to 128^3.

The second data set (VHF Torso) is a 512^3 block of the torso area of the Female CT data set from the Visible Human Project[25], in the same format as the UNC Head data. In our tests we used both the full 512^3 version and versions downsampled to 256^3 and 128^3.

The third data set (LLNL R-M) is time step 100 from the Richtmyer-Meshkov Instability data set from LLNL[23], which consists of an array of $2048 \times 2048 \times 1920$ 8-bit integers. We padded this to 2048^3 and then downsampled it to several different resolutions for testing.

We performed our fittings on a 64-bit Linux machine with dual Intel Xeon 3.0 GHz CPUs and 8 GB of RAM. As our fitting program is single threaded, it can only make use of one CPU.

Fitting results are given in table 2. For most of the data sets our fitting method achieves a reduction in file size of about 50% with an RMS error of about 1%-2%. An exception to this was the lower resolutions of the LLNL R-M data set. We suspect the reason for this is that these low resolution versions contain sharp, large-magnitude discontinuities which are not amenable to fitting with smooth Gaussian RBFs.

It is difficult to compare the efficiency of our fitting results with previous work due to a lack of prior reported fitting times. One exception to this is Hong *et al.* [14] who report fitting at the rate of 1300 grid points per minute. Although our results are several orders of magnitude faster, this is not a fair comparison since they are fitting unstructured data while we fit structured.

4 Rendering

We use ray-casting to render implicit RBF-encoded volumes. Data values at points in the volume are computed directly from the RBF representation without reconstructing the full data set in memory. Specifcally, the scalar value at a point is computed by summing up the values of all RBFs that overlap that point. The gradient vectors can be computed in a similar fashion, by summing the gradients of all RBFs that overlap the sample point (this works because the gradient operator distributes across summations). This sampling method can then be used to implement any rendering scheme that works by sampling data values and gradients at various points, such as direct volume rendering or point-based isosurface rendering.

4.1 GPU Direct Volume Rendering

We have implemented a GPU direct volume rendering algorithm using NVIDIA's new CUDA GPU programming system. Our implementation supports piecewise-linear transfer functions, gradient-based lighting, early ray termination based on accumulated opacity, and empty-space skipping based on checking whether the range of values contained within an octree cell overlaps the range which, given the transfer function, would produce any contribution to cumulative color or opacity. We also perform view-dependent level-of-detail by adjusting the ray step size and maximum octree depth based on the current distance along the ray.

Given an eye position and viewing direction, we construct ray origin and direction vectors for a perspective projection on the GPU and store them in GPU memory. We then execute a GPU kernel that traces each ray in parallel and writes the final resulting colors to GPU memory.

The kernel first sets the initial sample point to the ray's intersection with the data volume's bounding cube, and then begins stepping along the ray. At each sample point the kernel executes a space skipping routine that descends through the octree cells containing that point. For each cell, it makes a single 16-byte texture look-up and unpacks some parameters as described in section 3.6. It then checks whether the cell's range of data values would produce any contribution to the final color or opacity. If not, the kernel computes the ray's intersection with that octree cell and moves the sample point just beyond the intersection point. This is repeated until the skipping routine encounters a sample point at which it decends all the way through the octree.

The kernel then evaluates the scalar field and its gradient at the current sample point. This simply requires traversing the octree from the root to a leaf, evaluating the value and gradient of the RBF stored in each cell at the location of the sample point and summing up these values. We make use of the efficient location-code-based octree traversal methods of Frisken and Perry[11]. For each octree cell the kernel makes two 16-byte texture look-ups and unpacks some of the parameters as described in section 3.6. Finally, we sample color and opacity values from the transfer function using linear interpolation and apply diffuse, gradient-based lighting.

4.2 View-Dependent Level of Detail

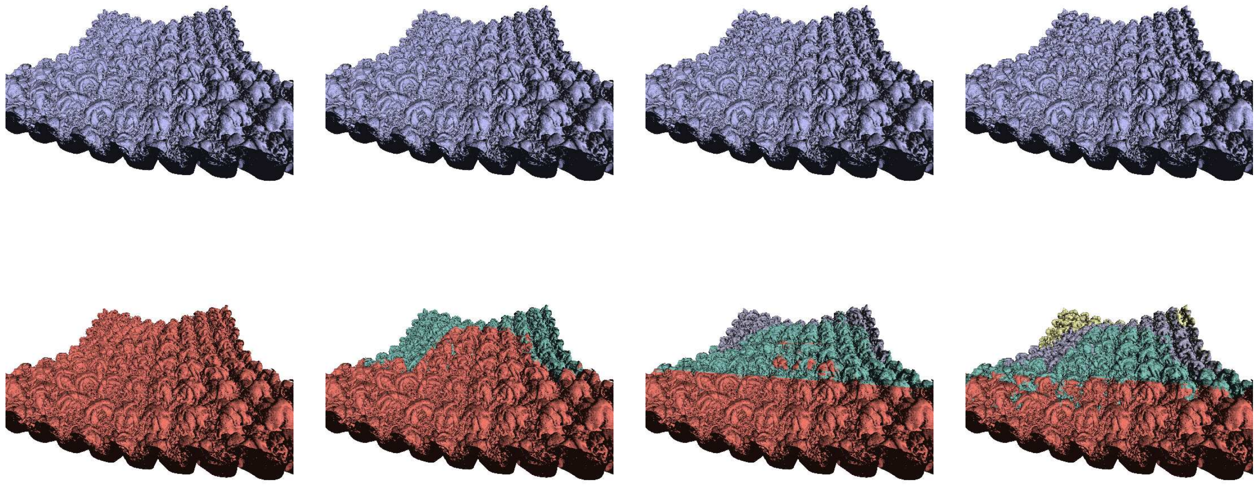

Figure 5: Comparison of varying amounts of view-dependent Level of Detail reduction (see section 4.2) for the 1024^3 LLNL R-M data set. The top row shows the renderings with the LOD reduction applied. The bottom row shows which LOD is being used in which area of the volume by color coding— from front to back, the colors represent levels of detail generated from 10, 9, 8, and 7 levels of the octree. From left to right, the rendering times are 0.59s, 0.53s, 0.48s, and 0.44s.

Our multi-resolution RBF hierarchy allows us to implement view-dependent level-of-detail rendering by reducing the depth in the octree that we visit based on the distance along the ray. Every time a certain amount of distance has been traversed, we reduce the maximum depth in the octree that we can visit by one level. To avoid level-of-detail transition discontinuities one can store the sample value computed at one level above the current level during the octree traversal, and blend between this sample and the final sample based on the distance from the previous level-of-detail transition. We experimented with this blending, but found that in practice the level-of-detail transitions tended to occur far enough away from the viewer that the discontinuities were not perceptible, even without blending. We did not use level-of-detail blending for the images and results in this paper.

In addition to decreasing the octree depth based on the distance along the ray, we also double the ray step size every time we decrease the maximum depth by one level. We account for this during the incremental updating of cumulative color and opacity by keeping track of the integer ratio of the current step size to the original step size and iterating our incremental updates this many times at every step, using the same sample color and opacity values.

Screen shots of renderings using various amounts of view-dependent level-of-detail reduction and rendering rates are given in fig. 5.

4.3 Blending

Truncating the RBFs at octree cell boundaries can lead to visually noticeable discontinuities. To resolve this, we take into account data in adjacent cells when fitting each cell's RBF (see section 3.1). Then, for each sample point, in addition to computing a sample value from the RBFs in that point's octree cells, we also compute sample values at that position from the RBFs in adjacent octree cells. We implement this by simply altering the location code of the sample point to the location code of a point in each adjacent cell.

Since each cell's overlap region extends halfway into each adjacent cell, at each sample point there is the possibility of blending with up to three face-adjacent cells, three edge-adjacent cells, and one corner-adjacent cell. The contribution of each of the adjacent cells is linearly weighted based on the distance from the sample point to the corresponding face, edge, or corner (see fig. 3 for an example on a quadtree). This blending scheme is similar to the one used in Ohtake *et al.*'s multi-level partition of unity implicits[28].

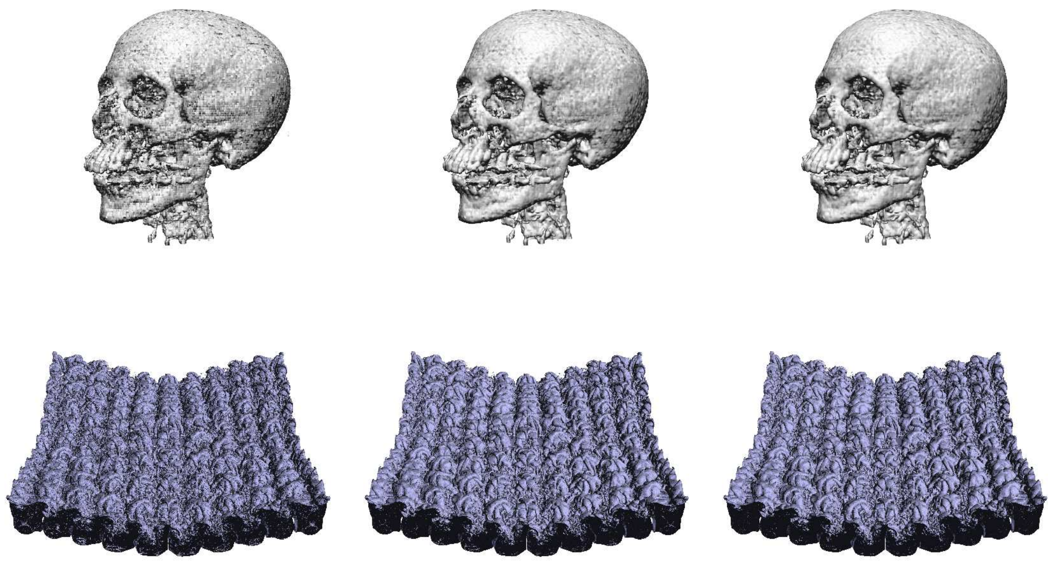

Figure 6: Comparison of (left to right) no blending, four-sample blending, and eight-sample blending (see section 4.3) for the 256^3 UNC Head and 1024^3 LLNL R-M data sets. Rendering times are given in table 3.

Data Set	Blending Type	Rendering Time
LLNL R-M	One-Sample	0.37 s
LLNL R-M	Four-Sample	0.72 s
LLNL R-M	Eight-Sample	1.11 s
UNC Head	One-Sample	0.10 s
UNC Head	Four-Sample	0.21 s
UNC Head	Eight-Sample	0.33 s

Table 3: Comparison of no blending, four-sample blending, and eight-sample blending (see section 4.3) for the 256^3 UNC Head data set and the 1024^3 LLNL R-M data set. Screen shots are given in fig. 6

While completely smooth blending requires blending across faces, edges, and corners, many of the discontinuities can be resolved by simply blending across faces. This reduces the number of samples required per point from eight (1 local + 3 face + 3 edge + 1 corner) to four, reducing rendering time by about a factor of two. A comparison between using no blending, four-sample blending, and eight-sample blending is given in fig. 6 and table 3.

4.4 Rendering Results

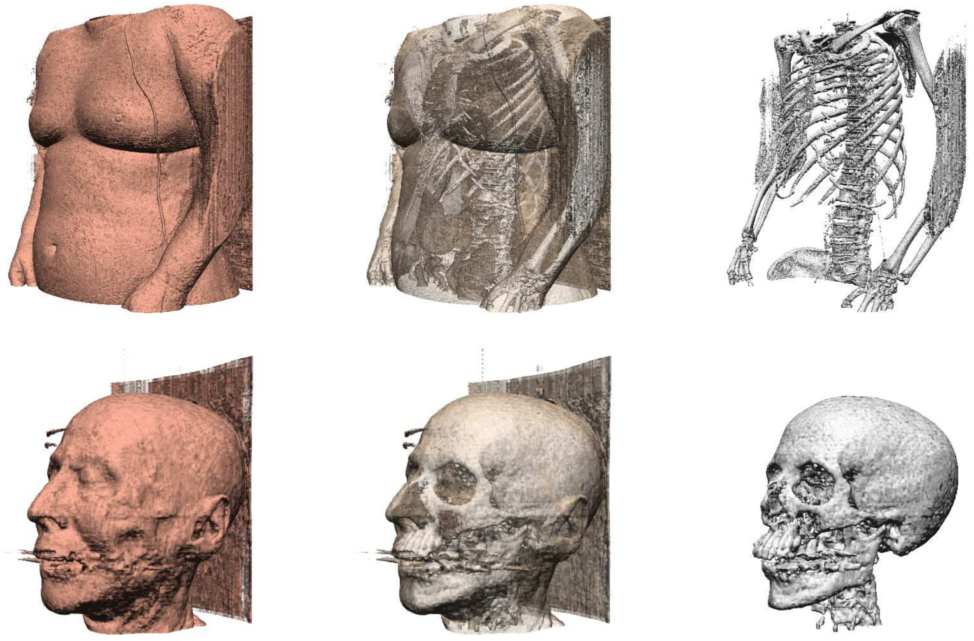

Figure 7: Rendering results using three different transfer functions for the 512^3 VHF Torso and the 256^3 UNC Head. Rendering times and other details are given in table 4.

Unless stated otherwise, all renderings were done with MLE-fitted data (see section 3.3), four-sample blending (see section 4.3), and an image resolution of 512^2. Renderings were performed using a beta version of NVIDIA's CUDA drivers on a GeForce 8800 GTX with 768 MB of RAM. Rendering

Data Set	Transfer Function	Data Size	Step Size	Rendering Time
UNC Head	Skin	256^3	384^{-1}	0.45 s
UNC Head	Skin/Bones	256^3	384^{-1}	0.67 s
UNC Head	Bones	256^3	384^{-1}	0.21 s
VHF Torso	Skin	256^3	384^{-1}	0.32 s
VHF Torso	Skin/Bones	256^3	384^{-1}	1.09 s
VHF Torso	Bones	256^3	384^{-1}	0.30 s
VHF Torso	Skin	512^3	768^{-1}	0.54 s
VHF Torso	Skin/Bones	512^3	768^{-1}	2.15 s
VHF Torso	Bones	512^3	768^{-1}	0.53 s
LLNL R-M	Isosurface	256^3	384^{-1}	0.26 s
LLNL R-M	Isosurface	512^3	768^{-1}	0.40 s
LLNL R-M	Isosurface	1024^3	1536^{-1}	0.72 s

Table 4: Rendering results for several different transfer functions and data sets of several different sizes. All renderings were done using four-sample blending (see section 4.3). Step Size is the length of a step along the ray given that the data volume is a unit cube. Images of several of these data sets are given in fig. 7.

times for several different data sets and transfer functions are given in table 4, and screen shots are given in fig. 7.

We found that the ray step size required to avoid artifacts was dependent on the size of the smallest octree nodes in the fitting, which is related to the amount of high frequencies in the fitted data. We therefore needed to use smaller steps for higher resolution data. For a given data set and transfer function, most of the differences in rendering rate between original data resolutions are accounted for by the difference in step size.

5 Future Work

A great deal of work has been done in the area of RBF fitting, for example in the machine-learning community. It seems likely that some of that work could further benefit procedural encoding through RBFs. Possible improvements include adding the ability to fit to a given error bound and removing the need to perform explicit blending between octree cells. It also might prove beneficial to investigate basis functions other than the Gaussian RBFs.

Another area of future work might be to investigate 4D fitting of time-varying data sets. Since these data sets often contain a large amount of temporal coherence, fitting multiple time steps simultaneously could result in a significant reduction in the number of basis functions required to represent the data set.

Finally, there are likely many possible uses of the higher-level information provided by the RBF representation that have not yet been explored, perhaps involving the use of high order derivatives that would have been too expensive to compute when using traditional volume rendering techniques.

6 Acknowledgements

We would like to acknowledge the Visible Human Project at the National Library of Medicine, Mark Duchaineau at Lawrence Livermore National Lab, Kelly Gaither at University of Texas at Austin, Klaus Mueller at SUNY Stony Brook, and the Volume Rendering Test Dataset Archive at the University of North Carolina for providing us the volume datasets. We greatly benefited in this work from several discussions with Joseph JaJa, Ramani Duraiswami, and Youngmin Kim (University of Maryland), Andy Cedilnik and Lisa Avila (Kitware), and Dave Luebke (NVIDIA Research). This work has been supported in part by DOD contract 060107778 and NSF grants CNS-0403313 and CCF-0429753.

References

[1] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[2] J. Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):341–355, Nov. 1988.

[3] J. Bloomenthal. An implicit surface polygonizer. In P. Heckbert, editor, *Graphics Gems IV*, pages 324–349. Academic Press, Boston, 1994.

[4] J. Bloomenthal, C. Bajaj, J. Blinn, M. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wywill. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.

[5] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 67–76, New York, NY, USA, 2001. ACM Press.

[6] J. C. Carr, W. R. Fright, and R. K. Beatson. Surface interpolation with radial basis functions for medical imaging. *IEEE Transactions on Medical Imaging*, 16(1):96–107, Feb. 1997.

[7] E. W. Cheney and W. A. Light. *A Course in Approximation Theory*. Brooks Cole, Pacific Grove, 1999.

[8] C. S. Co, B. Heckel, H. Hagen, B. Hamann, and K. I. Joy. Hierarchical clustering for unstructured volumetric scalar fields. In G. Turk, J. J. van Wijk, and R. Moorhead, editors, *Proceedings of IEEE Visualization 2003*, pages 325–332. IEEE, Oct. 19–24 2003.

[9] J. Duchon. Splines minimizing rotation-invariant semi-norms in sobolev spaces. *In W. Schempp and K. Zeller, editors, Constructive Theory of Functions of Several Variables, number 571 in Lecture Notes in Mathematics*, pages 85–100, 1977.

[10] N. Dyn, D. Levin, and S. Rippa. Numerical procedures for surface fitting of scattered data by radial basis functions. *SIAM Journal on Scientific and Statistical Computing*, 7(2):639–659, 1986.

[11] S. F. Frisken and R. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.

[12] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[13] J. C. Hart. Ray tracing implicit surfaces. In *SIGGRAPH 93 Modeling, Visualizing, and Animating Implicit Surfaces course notes*, pages 13–1 to 13–15. ACM SIGGRAPH, Aug. 1993.

[14] W. Hong, N. Neophytou, K. Mueller, and A. Kaufman. Constructing 3d elliptical gaussians for irregular data. In *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, 2006.

[15] Y. Jang, R. P. Botchen, A. Lauser, D. S. Ebert, K. P. Gaither, and T. Ertl. Enhancing the interactive visualization of procedurally encoded multifield data with ellipsoidal basis functions. *Eurographics*, 25(3), 2006.

[16] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl. Interactively visualizing procedurally encoded scalar fields. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2004)*, 2004.

[17] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.

[18] D. Karla and A. H. Barr. Guaranteed ray intersections with implicit surfaces. *Computer Graphics*, 23(3):297–306, 1989.

[19] S. M. Kay. *Fundamentals of Statistical Signal Processing: Estimation Theory*, chapter 7. Prentice Hall, 1993.

[20] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 255–262, Washington, DC, USA, 2001. IEEE Computer Society.

[21] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[22] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[23] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the ibm-sp system. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 70, New York, NY, USA, 1999. ACM Press.

[24] B. S. Morse, T. S. Yoo, P. Rheingans, D. T. Chen, and K. R. Subramanian. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *Shape Modeling and Applications, SMI 2001 International Conference on*, pages 89–98, 2001.

[25] National Library of Medicine (U.S.) Board of Regents. Electronic imaging: Report of the board of regents. U.S. Department of Health and Human Services, Public Health Service, National Institutes of Health, 1990. NIH Publication 90-2197.

[26] N. Neophytou, K. Mueller, K. McDonnell, W. Hong, X. Guan, H. Qin, and A. Kaufman. Gpu-accelerated volume splatting with elliptical rbfs. In *Joint Eurographics - IEEE TCVG Symposium on Visualization 2006 (EuroVis'06)*, 2006.

[27] H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura. Object modeling by distribution function and a method of image generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, J68-D(4):718–725, 1985. In Japanese (translated into English by Takao Fujiwara while at Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, 1989).

[28] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[29] A. Sherstyuk. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2), June 1999.

[30] G. Turk, H. Q. Dinh, J. F. O'Brien, and G. Yngve. Implicit surfaces that interpolate. In B. Werner, editor, *Proceedings of the International Conference on Shape Modeling and Applications (SMI-01)*, pages 62–73, Los Alamitos, CA, May 7–11 2001. IEEE Computer Society.

[31] G. Turk and J. F. O'Brien. Shape transformation using variational implicit functions. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[32] G. Turk and J. F. O'Brien. Modelling with implicit surfaces that interpolate. *ACM Trans. Graph.*, 21(4):855–873, 2002.

[33] M. Weiler, R. Botchen, S. Stegmaier, T. Ertl, J. Huang, Y. Jang, D. S. Ebert, and K. P. Gaither. Hardware-assisted feature analysis and visualization of procedurally encoded multifield volumetric data. *IEEE Comput. Graph. Appl.*, 25(5):72–81, 2005.

[34] A. P. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277, New York, NY, USA, 1994. ACM Press.

[35] G. Wyvill and A. Trotman. Ray-tracing soft objects. In *New Trends in Computer Graphics (Proceedings of CG International '90)*, pages 467–476. Springer-Verlag, 1990.

[36] G. Yngve and G. Turk. Creating smooth implicit surfaces from polygonal meshes. *Technical Report GIT-GVU-99-42, Graphics, Visualization, and Usability Center. Georgia Institute of Technology, 1999.*, 1999.