A Framework for Real-time Volume Visualization of Streaming Scattered Data

Sung W. Park* Lars Linsen† Oliver Kreylos* John D. Owens* Bernd Hamann*

* Institute for Data Analysis and Visualization (IDAV)
Department of Computer Science
University of California, Davis
Davis, CA 95616

†Department of Mathematics and Computer Science
Ernst-Moritz-Arndt-Universität Greifswald
Greifswald, Germany

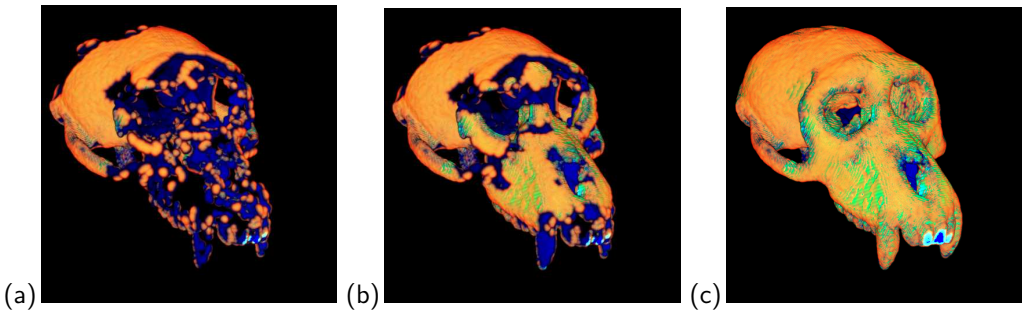

Figure 1: Progressive visualization of scattered data using discrete radial Hermite interpolation applied to Macaque Cranium data set: (a) 11,963, (b) 18,463, and (c) 32,963 samples.

ABSTRACT

Visualization of scattered data over a volumetric spatial domain is often done by reconstructing a trivariate function on some grid using scattered data interpolation methods and visualizing the function using standard visualization techniques. Scattered data reconstruction algorithms are often computationally expensive and difficult to implement. In order to visualize streaming scattered data, where visualization needs to take place in real time while new data is constantly streaming in, efficient approaches to scattered data reconstruction are required. We present a general framework for scattered data interpolation operating on discrete domains. Since common visualization methods require an underlying grid, it suffices to compute the scattered data reconstruction over the same grid. The key idea for speeding up the reconstruction is a re-factorization of the algorithm. The re-factorized version is designed such that it easily maps to graphics hardware architectures exploiting their performance and parallelism. Moreover, it naturally extends to applications for streaming data. As a proof of concept, we have implemented inverse-distance-weighted interpolation, natural neighbor interpolation, and radial Hermite interpolation using our general framework. We apply the framework to two kinds of streaming data: progressive scattered data and real-time sensor data with moving sensors delivering asynchronous measurements. To account for the scattered spatial and temporal distribution of streaming sensor data, we use a four-dimensional extension of our framework, which elegantly handles representation of time-varying data and leads to reconstructions that are smooth in both space and time.

1 INTRODUCTION

Scattered data methods arise naturally in a variety of applications. When data is gathered by measuring quantities of interest at locations in the real world, there is usually no predefined or implied connectivity between individual measurement sites, and the sites are not arranged on a grid. Even when dealing with numerically simulated data, where connectivity is usually required by the computational method, it might be changed or lost when data is post-processed, e. g., by creating multiresolution hiearchies or transmitting data in a progressive or streaming format. Without connectivity, a (scattered) data set can be considered as an unordered set of sample sites inside some domain with associated sample values. When it is understood that the sample values are the values of some underlying field function (temperature, humidity, etc.) at the sample sites, the process of *scattered data reconstruction* recreates this underlying function such that it can be evaluated everywhere inside the domain, not just at the sample sites. This reconstruction is essential to further analyze the data, for example by extracting contour surfaces or rendering it directly as a volume.

An important emerging application area for scattered data methods are wireless sensor networks, where small independent sensors measure environmental functions at their location, and asynchronously send their measurements to a base station using an ad-hoc wireless network. Sensors are typically not placed in pre-planned locations, and some sensors can change position over time by drifting with water or wind, or moving on their own. Sensor networks create new challenges for scattered data methods. Traditionally, scattered data reconstruction has been treated as a pre-processing step at the beginning of a visualization pipeline, and the efficiency of reconstruction methods has therefore not been much of a concern. Sensor network applications such as environmental monitoring or emergency response, on the other hand, require monitoring of "live" streaming sensor data, which implies that all reconstruction and visualization must be performed in real time. As most reconstruction methods are computationally expensive and difficult to implement efficiently, new methods need to be developed, or existing methods need to be reformulated for real-time streaming demands.

A very common usage pattern for scattered data reconstruction is to evaluate a reconstructed function on a regular grid, and to perform all subsequent analysis/visualization using that grid. It turns out that a large subset of existing scattered data methods, those based on *partition-of-unity* interpolation, can be re-factorized in such a way that it becomes possible to exploit computational coherency, and the high floating-point performance of modern programmable GPUs, to reduce the cost of reconstruction to the point

*{sunpark|okreylos|jowens|bhamann}@ucdavis.edu
†linsen@uni-greifswald.de

of real-time processing. Furthermore, these re-factorizations are also amenable to efficiently handle streaming data by only computing local updates to an existing reconstruction whenever new data arrives. We also found that our re-factorization scheme can handle true time-varying data, where asynchronous data from moving sensors is interpolated in both space and time, by treating it as static $(n+1)$-dimensional data, to create smoothly varying reconstructions that are easily visualized or analyzed. Furthermore, when treating samples as individual points in space-time, dynamic insertion/removal of sensors and moving sensors are handled implicitly. We describe our re-factorization in detail in Section 3, and our approaches to reconstructing streaming or time-varying data in Sections 3.2 and 3.3, respectively.

The re-factorization of scattered data reconstruction methods led to a natural separation of the methods into a setup phase that is done once for an entire data set, a per-sample update phase that is done whenever a streaming data set changes (by addition/removal of samples, or change of sample location/value), a per-sample phase during reconstruction, and a per-voxel phase during evaluation of the reconstruction on a regular grid. The setup and update phases are generally performed on the host computer's CPU, using spatial data structures such as kd-trees to quickly locate samples and locally update reconstructions; the per-sample reconstruction phase consists mostly of sending data from the application's address space into the GPU, and the per-voxel reconstruction phase is entirely done on the GPU using vertex and fragment programs. Modern programmable GPUs provide parallelism and high computational performance through streamlined designs, and we found that this hybrid approach leads to efficent implementations for both static and streaming scattered data, and the separation of the reconstruction algorithm into distinct phases leads to a general framework that can be easily adapted for different reconstruction methods. We discuss the effectiveness and generic character of our approach using several example reconstruction methods in Section 4, and investigate our implementation of the radial Hermite method in detail in Section 4.3.

2 Related Work

Since scattered data interpolation has been an area of research for a long time, many different approaches exist. The most common ones can be categorized into methods based on triangulation, inverse distances, radial basis functions, and natural neighbors. For details, we can refer to many surveys on this topic [1, 2, 5, 7, 13, 15].

Triangulation-based methods are fast, allow for higher-order interpolation, and, due to their local nature, are suitable for large data sets [17]. However, they require the computation of a triangulation in a preprocessing step, which becomes a very expensive operation in higher dimensions (e. g., tetrahedrization when operating on a volumetric domain). Sometimes, this drawback can be considered as minor, since the triangulation only has to be computed once (even if data values change over time). In the context of streaming data, however, new samples arise, old ones may vanish, and other samples may change their positions. Each of these changes requires a retriangulation. Thus, to make triangulation-based approaches applicable to streaming data, an efficient local retriangulation method would need to be developed.

Inverse distance methods are widely used due to their simplicity, but many suffer from their well-known artifacts imposed by the fact that each sample point has a radially symmetric influence regardless of the nature of the underlying data [20]. Being the simplest scattered data interpolation scheme, we use inverse-distance-weighted methods to evaluate what frame rates can be achieved for scattered data reconstruction for up to four dimensions using our general framework.

Scattered data interpolation based on radial basis functions like Hardy's multiquadric interpolation method [8] allow for smooth interpolations. As the scheme requires solving a linear equation in the size of the number of samples, only localized versions are applicable to larger data sets, i. e., data sets containing more than 100 samples. Moreover, linear equation solving is an expensive operation that does not map easily to the parallel workflow of graphics hardware. Although our general framework does also apply to interpolation schemes based on radial basis functions, we did not evaluate this option, as solving the linear equations would have become a major bottleneck.

Natural neighbor interpolation is a local approach based on Voronoi tessellation and, thus, works well for both sparse data and dense data with many sample points. Like triangulation-based methods, they suffer from an expensive preprocessing step for domain tessellation. However, unlike triangulation-based methods, there exist approaches for efficient implementations of Voronoi tessellations [10] that even scale to higher dimensions. We described our approach to implementing Sibson's natural neighbor interpolation [21] in detail in a recent paper [19].

Recently, a radial Hermite interpolation for scattered Hermite data has been introduced by Nielson [16]. Scattered Hermite data is scattered data enriched with normal vectors. The interpolation scheme interpolates both sample values and normals at the sample sites. We show that our framework is also applicable to scattered Hermite data, and use radial Hermite reconstruction as an illustrative example for progressive data reconstruction.

The concept of progressive data representation was introduced by Hoppe [11] for triangular meshes, and was adapted to progressive point set surfaces by Fleishman et al. [6], and to progressive representation of volume data by Staadt and Gross [22]. Research on progressive representation [6, 11, 22] focuses on how to generate the "best" progressive hierarchy in terms of minimization of some error measure defined at each point in time during progressive visualization. Determining "optimal" progressive representations for scattered data is beyond the scope of this paper; our interest is to efficiently reconstruct and visualize scattered data in a progressive representation. As far as we are concerned, the progressive representation may be the result of a sophisticated data analysis or merely the order in which the data was originally generated.

Recent advances in the performance and programmability of GPUs have made them increasingly attractive targets for the implementation of complex and computationally demanding scientific and visualization applications such as scattered data interpolation. Harris describes the characteristics of tasks that map well to modern GPUs, including simple control, ample data parallelism, and high arithmetic intensity [9]. Efficient implementations on GPUs can yield substantial performance gains over CPU implementations because of the GPU's superior arithmetic capability [3]; technology trends indicate this gap will only increase in the future [18]. As GPU-based applications are more tightly coupled to data visualization in systems such as Scout [14], the need for effective and efficient data visualization algorithms will continue to increase. Jang et al. [12] describe a method to reconstruct radial basis functions on GPUs, and two approaches to GPU-based natural neighbor interpolation have been described by Fan el al. [4] and Park et al. [19].

3 Re-factorization of Reconstruction Formula

The scattered data reconstruction problem as described in Section 1 is defined as follows: Given a set of samples $S = \{s_i = (\mathbf{p}_i, f_i) \mid \mathbf{p}_i \in \mathbf{R}^n, f_i \in \mathbf{R}\}$, i. e., a set of n-dimensional sample sites $\mathbf{p}_i$ with associated sample values f_i, define a function $f: \mathbf{R}^n \rightarrow \mathbf{R}$ such that $\forall s_i = (\mathbf{p}_i, f_i) \in S : f(\mathbf{p}_i) = f_i$. In general, all scattered data reconstruction methods can be expressed using linear combinations of the sample values, i. e., $f(\mathbf{p}) = \sum_{i=1}^{n} f_i \cdot d(\mathbf{p}, \mathbf{p}_i)$, where d is an appropriately chosen weight function. Depending on the recon-

struction method, weight functions $d(\mathbf{p}, \mathbf{p}_i)$ can either be defined algebraically, as done in Shepard's method, or they can be defined algorithmically, as done in Sibson's method.

A common subclass of interpolation methods uses a *partition-of-unity* approach, where the weighted sum of sample values is divided by the sum of all weights in a final step. In other words, methods in this class are characterized by the formula

$$f(\mathbf{p}) = \frac{\sum_{i=1}^{n} f_i \cdot d(\mathbf{p}, \mathbf{p}_i)}{\sum_{i=1}^{n} d(\mathbf{p}, \mathbf{p}_i)} \tag{1}$$

with an arbitrary weight function d. Weight functions are usually constructed such that they approach infinity as $\mathbf{p}$ approaches $\mathbf{p}_i$ (to ensure interpolation), and approach zero as $\mathbf{p}$ moves away from $\mathbf{p}_i$ (to ensure locality of influence). Even radial basis function methods such as Hardy's can be considered part of this class, although the normalization is performed implicitly when calculating interpolation weights using a set of linear equations.

3.1 Reconstruction on Regular Grids

As standard visualization techniques such as slices, isosurfaces, and volume rendering typically cannot be applied to scattered data directly, a very common approach is to evaluate the reconstruction function of a scattered data set on a Cartesian grid, and visualize the resulting gridded data. The algorithm for this resampling step is as follows:

1. for each grid point $\mathbf{p}$:
   (a) $f.c(\mathbf{p}) = 0, f.d(\mathbf{p}) = 0$
   (b) for each sample $s_i = (\mathbf{p}_i, f_i)$:
      i. $f.c(\mathbf{p}) \mathrel{+}= f_i \cdot d(\mathbf{p}, \mathbf{p}_i)$
      ii. $f.d(\mathbf{p}) \mathrel{+}= d(\mathbf{p}, \mathbf{p}_i)$
   (c) $f(\mathbf{p}) = f.c(\mathbf{p}) / f.d(\mathbf{p})$

The approach of scattered data reconstruction over a grid is illustrated in Figure 2(a). The weighted contribution of all samples is accumulated for each grid point, and then divided by the sum of weights to calculate the final reconstructed value. If there are N grid points and n samples, this algorithm evaluates the weight function $N \cdot n$ times. In the case of partition-of-unity methods, the above algorithm can potentially be sped up considerably by exploiting coherency in weight function evaluation when it is re-factorized inside-out:

1. for each grid point $\mathbf{p}$:
   (a) $f.c(\mathbf{p}) = 0, f.d(\mathbf{p}) = 0$
2. for each sample $s_i = (\mathbf{p}_i, f_i)$:
   (a) for each grid point $\mathbf{p}$:
      i. $f.c(\mathbf{p}) \mathrel{+}= f_i \cdot d(\mathbf{p}, \mathbf{p}_i)$
      ii. $f.d(\mathbf{p}) \mathrel{+}= d(\mathbf{p}, \mathbf{p}_i)$
3. for each grid point $\mathbf{p}$:
   (a) $f(\mathbf{p}) = f.c(\mathbf{p}) / f.d(\mathbf{p})$

The re-factorized approach to scattered data reconstruction over a grid is illustrated in Figure 2(b). The weighted contribution of each sample is distributed to (and accumulated at) all grid points inside the weight function's support. After all samples' contributions have been distributed, the final reconstructed value at each grid point is calculated by dividing by the sum of accumulated weights. This approach exploits coherency in two ways: First, the weight functions for each sample are evaluated in sequence and on a regular grid, which can lead to efficient implementations, for example using forward differencing; second, weight functions used in most reconstruction methods have only local support, and the grid points contained in that support can be found more easily. Furthermore, the per-sample processing order leads more naturally to GPU-based implementations exploiting their parallelism and high floating-point performance.

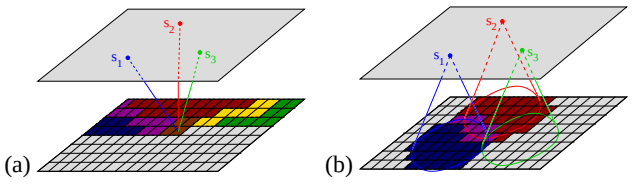

Figure 2: Re-factorization of scattered data reconstruction: (a) When operating on a grid, we iterate over all grid points and accumulate weighted contributions of values f_i for all samples $s_i = (\mathbf{p}_i, f_i)$. (b) When re-factorizing the reconstruction, we iterate over all samples $s_i = (\mathbf{p}_i, f_i)$ and accumulate weighted contributions of each sample value f_i for all grid points. A sample's contribution is often restricted to a local region.

3.2 Reconstruction of Streaming Data

Another major benefit of the re-factorization of the reconstruction algorithm is that it generalizes to streaming scattered data. We consider two different kinds of streaming scattered data:

- Progressive scattered data, where a fixed data set is stored at a remote location, and sent to a local graphics workstation for visualization. We want to generate preliminary visualizations of the data set as soon as the first samples arrive and refine the visualization incrementally over time.
- Time-varying sensor data, where spatially scattered sensors report asynchronous streams of time-varying sample values. Here, we want to update the visualization whenever a new sample value arrives from any of the sensors, or, in the case of moving sensors, when a sensor changes position.

For both progressive scattered data and time-varying sensor data, all changes to a data set's reconstruction due to incremental updates can be expressed by two basic operations:

- A new sample $(\mathbf{p}_i, f_i)$ arrives and is inserted into the reconstruction.
- A sample $(\mathbf{p}_i, f_i)$ becomes invalid and is removed from the reconstruction.

Two common cases, that of a sample changing its associated value and that of a sample moving to a new position, can be expressed by first removing the old sample and then inserting the new one. However, coherency between these two operations often makes it more efficient to represent them explicitly:

- A sample $(\mathbf{p}_i, f_i)$ changes its value to $(\mathbf{p}_i, f_i')$.
- A sample $(\mathbf{p}_i, f_i)$ changes its position and value to $(\mathbf{p}_i', f_i')$.

The special case where a sample only changes position but not its value is uncommon and does not lead to optimizations, and is therefore not treated separately. For many scattered data reconstruction methods, the two basic and two additional operations can be expressed very efficiently with only changing the gridded reconstruction locally.

For both types of streaming data, our basic reconstruction algorithm is modified as follows:

1. Calculate initial reconstruction based on the set of immediately available samples.
2. For each newly arriving sample s_i:

(a) Update any data structures created in the data set set-up phase, e. g., insert the sample into a kd-tree used for neighbor look-up.
(b) If existing samples' contributions change due to the insertion, e. g., deposition radii are adjusted, subtract the contribution of affected samples from the reconstruction, and add their contribution using the adjusted weight function.
(c) Add the new sample's contribution to the reconstruction.
(d) If immediate visualization is required, re-normalize the reconstruction in changed areas by dividing by the sum of deposited weights.

Due to the locality of most reconstruction methods, inserting a new sample only involves updating a constant number of already existing samples, and then inserting the new sample. Therefore, creating the final reconstruction of a data set using the progressive method has the same complexity as the original methods. For several methods (such as Shepard's global method), no existing samples have to be adjusted, and there is no performance difference between direct and progressive reconstruction.

In the case of time-varying sensor data, where new measurements are inserted into the current reconstruction as soon as they arrive, we can optimize the common case of a sample changing its value by merely subtracting its old contribution from the reconstruction, and then adding its new contribution (or adding the difference between the new and the old contribution). This does not require us to adjust the contributions of other samples, and leads to a very efficient implementation. As the adapted algorithm shows, re-factorized reconstruction offers a useful framework for handling streaming scattered data. We show the necessary steps in more detail in Section 4.3, where we discuss our implementation of the radial Hermite method.

3.3 Time Interpolation of Time-varying Data

The approach to time-varying scattered data described in the last section suffers from one serious drawback: Whenever a sample changes its associated value, the reconstruction function changes its value discontinuously. This fact makes it difficult to visually and analytically observe the behavior of a measured phenomenon. A better approach would interpolate sample values over space *and* time. Although space and time are usually treated and interpolated separately, a promising approach is to represent time-varying n-dimensional scattered data as static $(n+1)$-dimensional scattered data, where sample values do not change over time, but are associated with fixed sample positions in space-time. For example, if a single sensor located at position $\mathbf{p}$ generates values $f_1, f_2, \ldots, f_k$ at times $t_1, t_2, \ldots, t_k$, respectively, these changing values would be represented as k static and unrelated sample values $f_1, f_2, \ldots, f_k$ at the space-time positions $(\mathbf{p}, t_1), \ldots, (\mathbf{p}, t_k)$, respectively. Since we no longer explicitly represent individual sensors, but represent individual measurements instead by associating them with a single point in space and time, changing sample values, moving, and even dynamically disappearing and reappearing sensors are handled elegantly. For example, in Figure 3, we consider two moving sensors in a 2D domain that asynchronously generate measurements. The paths of the two sensors in space-time are denoted by the thin lines; the generated samples are shown as black dots. To reconstruct the data at time t_0, we intersect space-time with the plane $t = t_0$. The deposition spheres around close samples intersect the plane to form circles, which are used in our deposition scheme.

In other words, we represent time-varying 3D scattered data as static 4D scattered data in space-time. This allows us to extract 3D reconstructions for arbitrary time points t_0 by intersecting the (not explicitly calculated) 4D reconstruction function $f(x,y,z,t)$ with

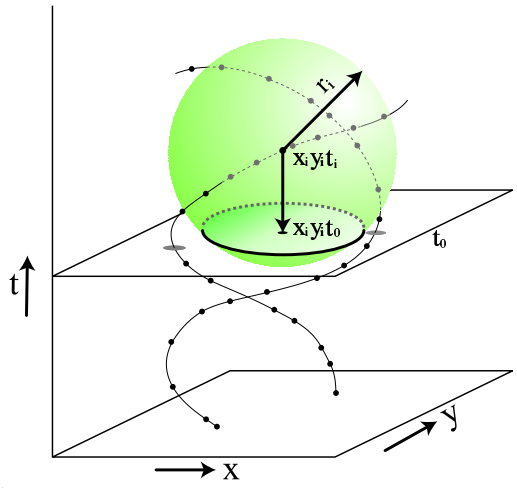

Figure 3: Time interpolation in a 2D domain. The thin lines are the space-time paths of two moving sensors that generate measurements at random times (black dots). Data is reconstructed for time t_0 by only considering the plane $t = t_0$. The intersections of the weight function supports (sphere) of near samples with the plane are deposited into the 2D reconstruction using our normal algorithm.

the time-orthogonal hyperplane $t = t_0$. We can use our existing implementation for 3D reconstruction using our hybrid CPU/GPU approach for the time-varying case by observing that depositing the 4D weighted contribution of a sample s_i in space-time and then intersecting with the $t = t_0$ hyperplane is equivalent to depositing the intersection of the weighted contribution with $t = t_0$ in 3D space. For example, in the case of our modified version of Shepard's reconstruction method [20], we deposit hyperspheres of radius r_i around a sample $s_i = (x_i, y_i, z_i, t_i)$, where the deposited value is the value of Shepard's weight function in four dimensions. The intersection of such a hypersphere with the hyperplane $t = t_0$ is a 3D sphere centered at (x_i, y_i, z_i, t_0) with radius $r_i' = \sqrt{r_i^2 - (t_i - t_0)^2}$. Since projecting this intersection into 3D space means to merely drop the time component, we can deposit just as in the 3D case, as long as we use the full 4D formulation of Shepard's weight function for each grid point.

To analyze the behavior of this approach to time-varying reconstruction, let us first consider the case where a complete data set is reconstructed for a time point t_0. As the data set contains samples with times before *and after* t_0, the algorithm can "look into the future," and if t_0 is slowly moved forward in time, samples in the future smoothly "fade into" existence in the reconstruction, whereas samples passed by t_0 smoothly fade out. This smooth variation in both space and time is better suited to detect, either visually or analytically, the changing behavior of an observed data set.

In the case of real-time monitoring, it is not possible to look into the future; therefore, whenever a new sample arrives and is inserted into the reconstruction, it will always appear exactly at time t_0, where it has maximal influence on the reconstruction. Thus, arrival of new samples will cause discontinuous changes. "Older" samples will still fade out smoothly. This fundamental drawback of real-time monitoring can be overcome in two ways: If the application only requires "near real time" monitoring, the reconstruction time t_0 could always be fixed behind real time by a set amount, which allows the algorithm to look into the future by that amount and smoothly fade in new samples. If this is not an option, another approach is to insert new samples in real time, and to allow a user to rewind the live data stream when a noteworthy change happens, and replay a section in "TiVo mode," which allows looking into the

future because the point of view lies in the past.

4 Applying the General Framework

To demonstrate the versatility of our framework, we briefly describe how to apply it to several different scattered data reconstruction methods. We will describe the radial Hermite interpolation method in more detail in Section 4.3, as it is a good example for a method that requires non-trivial changes to handle streaming data.

4.1 Inverse-distance-weighted Interpolation

The best results in terms of real-time requirements are to be expected when using the simplest scattered data approach. Shepard [20] introduced the idea of using radial distance functions for scattered data interpolation, i. e., he used weight functions of the form

$$d(\mathbf{p},\mathbf{p}_i) = \frac{|\mathbf{p}-\mathbf{p}_i|^\mu}{\sum_{j=1}^n |\mathbf{p}-\mathbf{p}_j|^\mu} \ . \tag{2}$$

Most commonly, quadratic weights ($\mu = 2$) are used (also for our implementation).

Traditionally, a scattered data reconstruction over a Cartesian grid would iterate over the grid points and compute the interpolated value at each grid point by iterating over the samples and adding up their weighted values. Using our new framework, we exchange inner and outer loop. Thus, we iterate over all samples and compute the contribution of each sample to each grid point by iterating over all grid points. The contribution is defined by the inverse distance weights in Equation (2). We add both the weighted sample values and the weights themselves for each grid point. The reconstructed interpolation value at each grid point is the ratio of the sum of the weighted sample points over the sum of the weights as defined in Equation (1).

A local version of Shepard's method restricts the influence of each sample to a local region around the sample site. Using our framework for the implementation of a local version, the inner loop does not iterate over all grid points but only over the grid points within a certain distance to the sample site. Using radial distances, the contribution of a sample is restricted to an n-dimensional sphere.

4.2 Natural Neighbor Interpolation

Natural neighbor interpolation is based on a Voronoi tesselation of the n-dimensional domain. Only the natural neighbors influence the interpolation (local interpolation scheme), and the weights in Equation (1) are determined by the size of the Voronoi regions. Over a regular grid, the weights can be computed by counting the grid points that fall into the respective region. In [19], we showed that when using the idea of re-factorization the computation of Sibson's approach to natural neighbor interpolation reduces to drawing an n-dimensional sphere around each grid point, whose radius is the distances from the grid point to the closest site. Moreover, the computation of the Voronoi diagram becomes obsolete and is replaced by mere kd-tree look-ups. Our discretization and re-factorization of Sibson's natural neighbor interpolation method is described in detail in a [19].

4.3 Radial Hermite Interpolation

If we know or can precompute gradient information at the sites in a scattered data set, the data is defined by $S = \left\{ s_i = (\mathbf{p}_i, f_i, \mathbf{n}_i) \;\middle|\; \mathbf{p}_i \in \mathbf{R}^n, f_i \in \mathbf{R}, \mathbf{n}_i \in \mathbf{R}^n \right\}$, where $\mathbf{p}_i$ and f_i are as before and $\mathbf{n}_i$ denotes the sample's gradient. Scattered data of this form is referred to as *scattered Hermite data*. To properly reconstruct scattered Hermite data, both sample values and sample gradients need to be interpolated. Thus, we are searching for a function $f\colon \mathbf{R}^n \to \mathbf{R}$ such that $\forall s_i = (\mathbf{p}_i, f_i, \mathbf{n}_i) \in S : f(\mathbf{p}_i) = f_i \wedge \nabla f(\mathbf{p}_i) = \mathbf{n}_i$.

Nielson [16] proposes an approach to scattered Hermite data interpolation using radial Hermite operators, and applies it to point cloud fitting. In this context, an implicit surface is represented by scattered Hermite data, where sample values f_i are ignored, and the normal vectors define a local signed distance field for the implicit surface. Nielson's Hermite interpolant is defined as

$$f(\mathbf{p}) = \frac{\sum_{i=1}^n \langle \mathbf{p}-\mathbf{p}_i, \mathbf{n}_i \rangle \cdot d(\mathbf{p},\mathbf{p}_i)}{\sum_{i=1}^n d(\mathbf{p},\mathbf{p}_i)} \ .$$

The weights $d(\mathbf{p},\mathbf{p}_i)$ could be chosen as in Equation (2), leading to an inverse-distance-weighted interpolation of scattered Hermite data, but Nielson suggested to use another radial function instead. His radial Hermite interpolation weights are calculated as

$$d(\mathbf{p},\mathbf{p}_i) = \left(\frac{(r_i - |\mathbf{p}-\mathbf{p}_i|)_+}{|\mathbf{p}-\mathbf{p}_i|} \right)^\mu \ ,$$

where r_i is the radius of influence for each sample, the exponent $\mu > 1$ plays the same role as in Shepard's method, and $(x)_+ := \max(x, 0)$, i. e., equal to x for positive values and equal to zero otherwise. Nielson chooses the radius of influence for each sample s_i by finding the k nearest neighbors of site $\mathbf{p}_i$, and setting r_i to the distance to the farthest neighbor. We implemented this method in the per-data set set-up phase of our algorithm, by first creating a kd-tree containing all sites, and then finding up the k nearest neighbors for each sample.

We apply our re-factorization approach the same way as for Shepard's method. The only differences are the use of different weights $d(\mathbf{p},\mathbf{p}_i)$, and the fact that f_i is replaced by the dot product $\langle \mathbf{p}-\mathbf{p}_i, \mathbf{n}_i \rangle$. Thus, Step 2 of our framework becomes

2. for each sample $s_i = (\mathbf{p}_i, f_i, \mathbf{n}_i)$:
   (a) for each grid point $\mathbf{p}$:
      i. $f.c(\mathbf{p}) \mathrel{+}= \langle \mathbf{p}-\mathbf{p}_i, \mathbf{n}_i \rangle \cdot d(\mathbf{p},\mathbf{p}_i)$
      ii. $f.d(\mathbf{p}) \mathrel{+}= d(\mathbf{p},\mathbf{p}_i)$

Since this scheme describes a local and radial method, the inner loop, again, only considers grid points within an n-dimensional sphere of radius r_i centered at each site $\mathbf{p}_i$. The deposition of weighted contributions for each sample is performed entirely on the GPU by processing each z-slice of the volume that intersects the sample's influence sphere, and rendering a square that contains the intersection of the influence sphere with the slice. Each square is rendered using a fragment program that takes as input the sample site $\mathbf{p}_i$, the sample normal $\mathbf{n}_i$, and the world position of the pixel $\mathbf{p}$ (all values are passed to the fragment program as texture coordinates). The fragment program then calculates the radial Hermite weight function, and the dot product of the normal vector and the distance vector from $\mathbf{p}$ to $\mathbf{p}_i$. It finally deposits the counter and denominator of the sample's contribution in the image buffer representing the z-slice, which is inserted into the 3D texture representing the reconstruction at the appropriate position.

As mentioned in Section 3.2, reconstruction algorithms have be be adapted to allow reconstruction of progressive or time-varying sensor data. Radial Hermite interpolation serves as a good example to illustrate this adaptation process, as it requires non-trivial operations when inserting/removing samples.

According to the definition of the radii r_i, whenever a new sample s_i is inserted (or removed), the influence radii of all neighboring samples need to be adjusted. More precisely, when removing s_i, the radii r_j of all samples s_j who have s_i as one of their k nearest neighbors have to be recalculated. This is equivalent to updating all samples s_j whose influence spheres contain the site of s_i. We

use an adapted kd-tree that stores each sample as a sphere to efficiently find all affected s_j. First, we subtract the contributions of all found s_j from the reconstruction; then, after inserting (or removing) s_i from the kd-tree, we recalculate the radii of all found s_j by doing another nearest neighbor look-up. Afterwards, we add the (changed) contribution of all s_j back into the reconstruction, and finally add (or subtract) the contribution of sample s_i. Since only a constant number of existing samples is affected by insertion/removal, and adapting an affected sample's radius does not in turn affect other samples, the asymptotic complexity of progressive reconstruction is the same as that of the original algorithm.

5 Implementation

We utilize GPU for optimal performance by doing both the reconstruction and the rendering in the GPU. To take full advantage of the framework described, we use the latest programmable graphics hardware. Although having floating point precision is desired, we are limited to current generation of cards that only support 16-bit blending. 16-bits are sufficient for most cases for doing Radial Hermite or Shepards reconstruction but they do pose problems for doing natural neighbor reconstruction for small and sparse data sets.

We use openGL libraries along with NVIDIA's Cg language for vertex and fragment programming to test our refactorization. We test our results on a Pentium 4, 3.2GHz machine with 2GBs of memory, equipped with an NVIDIA's GeForce 6800GT graphics card with 256MB of video memory.

For reconstruction of both Radial Hermite and Shepard's method using re-factorization in 3D, we reconstruct the volume by considering a slice at a time. For each slice k, we consider all sites that have an influence on k. Since the cross-section of influence on k for these two particular methods is a circle, we render a quad that covers the circle. Along with the quad, we pass site location, site value, and an associated normal in the Radial Hermite case, to the fragment program. For each fragment, the program tests if it lies in the circle of influence using vertex attributes such as the texture coordinates and color. If the fragment lies in the circle, it computes and outputs values of contribution to the frame buffer, which is accumulated in the graphics hardware using blending. For Radial Hermite reconstruction, the values outputted is described in Section 4.3. For variation of Shepard's implementation, output values are similar to the Radial Hermite, only that the dot product is replaced by the site value. Once all the values have been accumulated, it is normalized in the subsequent pass in the GPU before it is rendered.

For streaming implementation, we only update the regions that are affected. With each point streaming in, the region of influence is determined by inserting the point into the kd-tree and doing a local search.

Alternatively, the re-factorized reconstruction step can be entirely computed utilizing the CPU. In certain cases, reconstruction on the CPU is not significantly slower than its computation on the GPU. However, the transfer of the reconstructed data from the CPU to the GPU for rendering purposes becomes a bottleneck when using large volumetric grids.

6 Results and Discussion

In order to test the effectiveness of the re-factorized implementations of the various interpolation schemes, we present results for three representative application scenarios. First, we analyze reconstruction of a 3D dataset using both radial Hermite and Shepard's interpolation. Second, we use radial Hermite and Shepard's interpolation on a streaming version of the same dataset. Third, we demonstrate four-dimensional modified Shepard's interpolation on a time-varying streaming sensor data set.

	Shepard's		Hermite	
Data size	35,000	350,000	35,000	350,000
128^3 grid	0.344 s	0.328 s	1.031 s	1.091 s
256^3 grid	1.250 s	2.870 s	1.420 s	3.230 s

Table 1: Times to reconstruct a 35,000- and 350,000-sample data set on two different-size Cartesian grids using Shepard's and radial Hermite interpolation.

Radial Hermite and Shepard's Interpolation. We generated two scattered Hermite data sets from a polygonal surface model of a Macaque Cranium. The generated data sets consist of 35,000 and 350,000 samples, respectively, with randomly distributed sample sites and sample values representing the distance to the polygonal surface model. We have applied radial Hermite interpolation interpreting the data as scattered Hermite data and modified Shepard's interpolation interpreting the data as regular scattered data (without normal information). We perform the reconstruction over Cartesian grids of size 128^3 and 256^3. Figures 4 and 5(d) show a direct volume rendering of the reconstructed data field over a grid of size 256^3 using radial Hermite interpolation and 35,000 samples. Computation times for reconstruction were on the order of a second, with larger samples, larger grids, and more complex interpolation methods incurring larger computation times. Our timing results are detailed in Table 1.

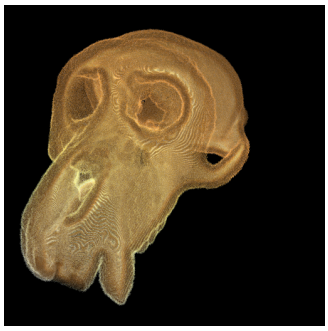

Figure 4: Isosurface rendering of reconstructed Macaque Cranium using 35,000 samples, radial Hermite interpolation and a 256^3 Cartesian grid.

Interpolation of Streaming Data. Using the same data set, we test the streaming capabilities of our framework in the context of progressive visualization. We load the data set incrementally, beginning visualization after receiving 50 samples. The reconstructed grid is updated sample by sample while data is streaming in, continuing until all data has been received. Figure 5 shows the progressive visualization using radial Hermite interpolation to reconstruct the data over a Cartesian grid of size 256^3, using direct volume rendering for visualization of the reconstructed field. The figure shows the visualization after having loaded 1,446, 1,876, 2,036, and 35,000 samples. A video showing a real-time progressive visualization accompanies the paper.[1] Our timing results are detailed in Table 2.

Interpolation of Time-varying Data. Finally, we demonstrate the use of our framework on a time-varying, streaming sensor data set (courtesy of the Monterey Bay Aquarium Research Institute [MBARI]) measuring temperature in Monterey Bay. The data set was generated by four moving sensors, transmitting their

[1]The video is available at http://www.math-inf.uni-greifswald.de/ linsen/final.avi.

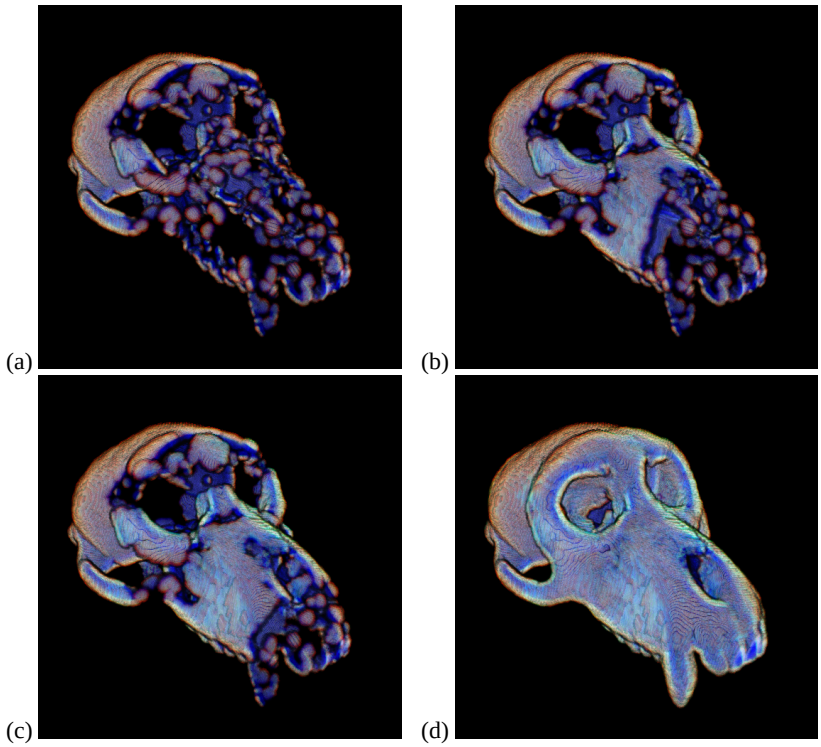

Figure 5: Progressive visualization of scattered data using a re-factorized implementation of radial Hermite interpolation applied to Macaque Cranium data set: (a) 1,446 samples, (b) 1,876 samples, (c) 2,036 samples, and (d) 35,000 samples.

Data set size	Grid size	Streaming window size 50	100	500	1000
35,000	128^3	14.6	12.6	6.7	3.7
	256^3	1.5	2.0	1.7	1.5
350,000	128^3	13.4	12.6	6.1	3.7
	256^3	1.5	2.0	1.7	1.2

Table 2: Frame rates when using progressive reconstruction. The streaming window size determines how many samples are inserted into the reconstruction before the visualization is updated. Frame rates include inserting samples, reconstructing the volume, and volume rendering the result in a 512×512 window. The table entries are averaged over the entire progressive reconstruction process, and given in frames per second.

data asynchronously to a server. The sensors are mounted onto autonomous devices, "gliders," that float and dive through the bay. The devices move on a predefined route but deviate heavily under the influence of the strong currents in the bay.

We reconstruct the temperature data field over a grid of size $128 \times 128 \times 64 \times 1800$ using four-dimensional modified Shepard's interpolation. We perform Shepard's interpolation uniformly over three spatial dimensions and one time dimension. For visualization, we directly volume-render time-orthogonal three-dimensional hyperplanes. Figure 6 shows four visualizations, using two interpolation methods, at two consecutive points in time. Figures 6(a) and (b) show the four-dimensional interpolation using all samples. Figures 6(c) and (d) show the same frames using four-dimensional interpolation using no samples with time coordinates higher than the visualized time slice. In Figures 6(c) and (d), we observe a sudden change between the two consecutive frames. The reconstruction is thus discontinuous in time. Figures 6(a) and (b), on the other hand, show a slight change indicating a smooth interpolation over time. Generating all 1,800 frames for the video took 368.97 s when interpolating from past and future samples, and 346.03 s when interpolating from past samples only.

7 Conclusions

We have presented a framework for real-time volume visualization of streaming scattered data. The scattered data reconstruction evaluates the reconstructed function on a Cartesian grid. Subsequent visualization is performed on that grid. We re-factorized the reconstruction step for partition-of-unity interpolations in a way that allows for exploiting computational coherency and floating-point performance of modern programmable GPUs. In particular, the general framework of our re-factorized approach is amenable to efficiently handle streaming data.

We have addressed two streaming applications: progressive visualization of a static scattered data set and streaming sensor viualization with moving sensors transmitting samples asynchronously over time. Our re-factorized reconstruction framework also scales to higher dimensions, which allows handling time-varying data using four-dimensional space-time interpolation. Treating samples as points in space and time implicitly covers the management of sample insertion, deletion, or movement. We have generated examples using various interpolation schemes to demonstrate the effectiveness and efficiency of our approach.

Acknowledgments

This work was supported by the National Science Foundation under contract ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, through the National Partnership for Advanced Computational Infrastructure (NPACI) and a large Information Technology Research (ITR) grant; and the National Institutes of Health under contract P20 MH60975-06A2, funded by the National Institute of Mental Health and the National Science Foundation. We thank the members of the Visualization and Graphics Research Group at the Institute for Data Analysis and Visualization (IDAV) at the University of California, Davis. Macaque cranium data set courtesy of Eric Delson and the NYCEP Morphometrics Group.

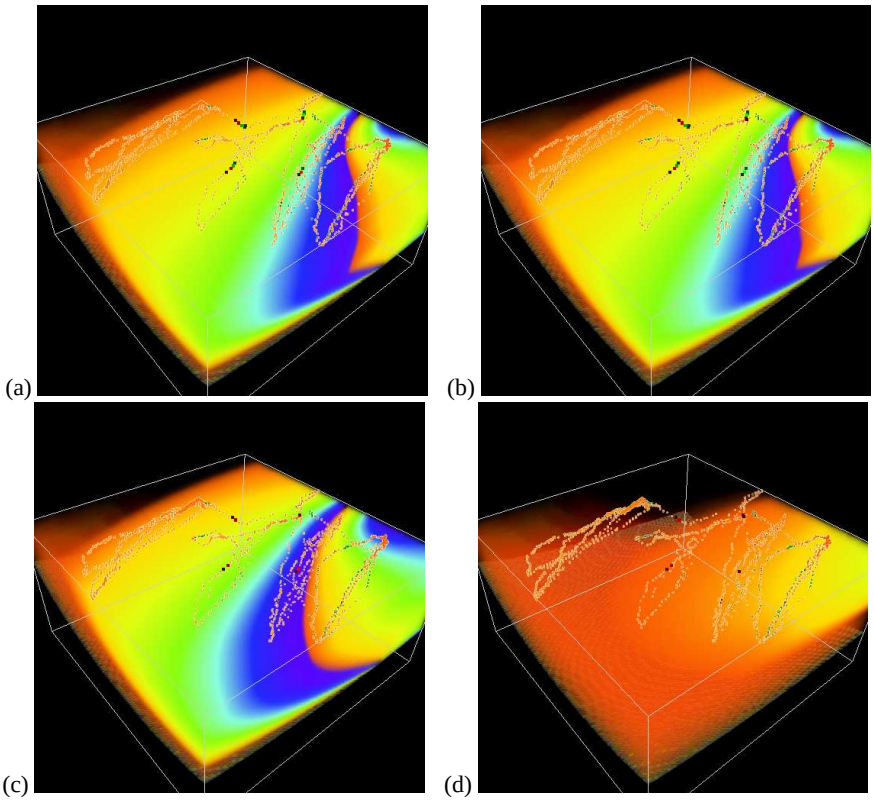

Figure 6: Visualization of streaming temperature data in Monterey Bay scattered over space and time. We perform four-dimensional natural neighbor interpolation to reconstruct the scattered data over a $128 \times 128 \times 64 \times 1800$ grid. These figures show direct volume rendering of a three-dimensional time-orthogonal hyperplane. (a) and (b) show two consecutive frames when considering all space-time samples for interpolation. (c) and (d) show the same frames when only considering samples from the past and present, thus exhibiting a visual discontinuity in time. The video from which these frames were extracted accompanies the paper.

REFERENCES

[1] P. Alfeld. *Mathematical Methods in Computer Aided Geometric Design*, chapter Scattered data interpolation in three or more variables, pages 1–34. Academic Press, 1989.

[2] I. Amidror. Scattered data interpolation methods for electronic imaging systems: A survey. *Journal of Electronic Imaging*, 2(11):157–176, 2002.

[3] I. Buck and T. Purcell. *GPU Gems: A Toolkit for General Purpose Computing on the GPU*, pages 621–636. Addison Wesley, Apr. 2004.

[4] Q. Fan, A. Efrat, V. Koltun, S. Krishnan, and S. Venkatasubramanian. Hardware assisted natural neighbour interpolation. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2005.

[5] G. Farin. *Computer Aided Geometric Design*, chapter Surfaces over Dirichlet tessellations, pages 281–292. 1990.

[6] S. Fleishman, D. Cohen-Or, M. Alexa, and C. T. Silva. Progressive point set surfaces. *ACM Trans. Graph.*, 22(4):997–1011, 2003.

[7] R. Franke and G. M. Nielson. *Geometric Modeling: Methods and Applications*, chapter Scattered Data Interpolation: A Tutorial and Survey, pages 131–160. Springer Verlag, New York, 1991.

[8] R. Hardy. Multiquadratic equations of topography and other irregular surfaces. *Journal of Geophysical Research*, 76:1905–1915, 1971.

[9] M. Harris. *GPU Gems 2: Mapping Computational Concepts to GPUs*, pages 493–508. Addison Wesley, Mar. 2005.

[10] K. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, Aug. 1999.

[11] H. Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 99–108, New York, NY, USA, 1996. ACM Press.

[12] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl. Interactively Visualizing Procedurally Encoded Scalar Fields. In O. Deussen, C. Hansen, D. Keim, and D. Saupe, editors, *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '04*, 2004.

[13] S. K. Lodha and R. Franke. Scattered data techniques for surfaces. In *Proceedings of Dagstuhl Conference on Scientific Visualization*, pages 182–222. IEEE Computer Society Pres, 1999.

[14] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *IEEE Visualization 2004*, pages 171–178, Oct. 2004.

[15] G. M. Nielson. Scattered data modeling. *IEEE Computer Graphics and Applications*, 1:60–70, January 1993.

[16] G. M. Nielson. Radial Hermite operators for scattered point cloud data with normal vectors and applications to implicitizing polygon mesh surfaces for generalized CSG operations and smoothing. In G. Turk, J. J. van Wijk, and H. Rushmeier, editors, *IEEE Visualization 2004*, pages 203–210. IEEE Computer Society Press, 2004.

[17] G. M. Nielson, H. Hagen, and H. Müller. *Scientific Visualisation*. IEEE Computer Society Press, 1997.

[18] J. Owens. *GPU Gems 2: Streaming Architectures and Technology Trends*, chapter 29, pages 457–470. Addison Wesley, Mar. 2005.

[19] S. W. Park, L. Linsen, O. Kreylos, J. D. Owens, and B. Hamann. Discrete Sibson interpolation. *IEEE Transactions on Visualization and Computer Graphics*, to appear, 2005.

[20] D. Shepard. A two-dimensional interpolation function for irregularly spaced data. In *Proceedings of 23rd National Conference*, pages 517–524. ACM, August 1968.

[21] R. Sibson. A vector identity for the Dirichlet tessellation. *Mathematical Proceedings of the Cambridge Philosophical Society*, 87(1):151–155, 1980.

[22] O. G. Staadt and M. H. Gross. Progressive tetrahedralizations. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *Proceedings of IEEE Visualization '98*, pages 397–402. ACM Press, 1998.