University of Zurich[UZH]

Renato Pajarola

Processing Streams of Points

TECHNICAL REPORT – No. IFI-2005.01

2005

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Renato Pajarola
Processing Streams of Points
Technical Report No. IFI-2005.01
Visualization and Multimedia Lab
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
http://vmml.ifi.uzh.ch/

Processing Streams of Points

Technical Report ifi-2005.03

Renato Pajarola*
Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

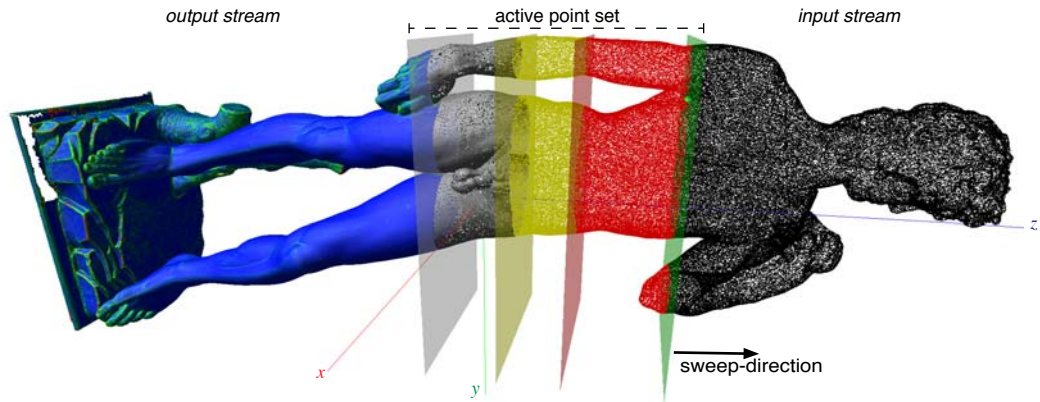

Figure 1. Snapshot of simulated stream-processing stages with (r.t.l.): points to be read from input stream (black points), in nearest neighborhood evaluation (red points), during normal computation (yellow points), amid curvature estimation (shaded grey points) and fully processed and written to output stream (shaded color-coded splats).

Abstract

With the gigantic sizes of captured 3D models, e.g. from high-resolution laser range scanning devices, it has become increasingly important to provide basic and efficient processing methods for large unorganized and raw surface-sample point data sets. In this paper we introduce a novel ***stream-based*** *(and out-of-core) point processing framework. The proposed approach processes points in an orderly sequential way by sorting them and sweeping along a spatial dimension. The major advantages of this novel concept are: (1) support of extensible and multiple concatenable local operators called* ***stream operators****, (2) low main-memory usage and (3) applicability to process very large data sets out-of-core.*

Keywords: point processing, sequential processing, normal estimation, curvature estimation, fairing

1. Introduction

Points as rendering and modeling primitives have become a powerful alternative to traditional polygonal object representation. Note that point samples are the natural raw output data primitives of the geometry capturing stage in most 3D object acquisition systems, e.g. laser range scanning of large objects [LPC+00]. In fact, points or 3D coordinates are the fundamental geometry-defining entities. Satisfying provably correct surface sampling criteria as discussed in [Mee01], a set of points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ in 3D space fully defines the geometry as well as the topology of a surface including boundaries, components and genus. Here we assume that input point data sets reasonably sample the represented surfaces, i.e. satisfying the Nyquist sampling criteria.

With the dramatically increasing use and precision of 3D capturing systems it is critical to support raw point cloud data in a practical way. In particular, basic point processing operations such as surface normal estimation or fairing must be supported. Such operators can be computed efficiently if the unorganized point data can be loaded into main memory and organized in some spatial indexing data structure. However, this approach while optimal up to some limit, is main-memory inefficient and will dramatically decrease in performance when the models exceed available physical main memory. In the case of significant mismatch between model and physical main memory size it may cause such *in-core* approaches nearly come to a halt due to memory trashing [Den70]. Moreover, combining multiple operations cannot easily be addressed in an efficient way by merely linking multiple operators.

In this paper we introduce and set the stage for a new *stream-processing* concept, a novel approach for processing points sequentially to improve memory access coherency and dramatically limit main memory cost. This sequential stream-processing concept directly allows us to process extremely large models *out-of-core* without the need to partition the models or process them on very large server machines. In fact, our stream-processing framework has such a low main memory usage that physical main memory limitation is practically irrelevant and extremely large models can be processed on very memory-limited machines.

The operations that are supported in our stream-processing concept are local operators $\Phi(\boldsymbol{p})$, called *stream operators*, that perform a function on a point $\boldsymbol{p}$ using only its local neighborhood. Many fundamental operations such as normal or curvature estimation on raw point data sets follow this principle, requiring the definition of a local neighborhood and then performing a computation of attributes based on this neighborhood. Also filter operations such as fairing are in this category and in general require a local neighborhood to operate on. Indeed, surface parameter estimation and filter operations are among the most important tasks for processing raw points. Our *stream-processing* concept supports direct non-recursive local operators $\Phi(\boldsymbol{p})$ that include any nearby sample points within a well defined local neighborhood.

2. Related Work

Since point primitives have been discussed in [LW85] and [GD98], many display techniques have been proposed such as [RL00, PZvBG00, RPZ02, BWK02, KV03, BK03, PSG04] (see also surveys [SPL04, SP04]). In general these techniques address higher-level point processing tasks such as multiresolution modeling and rendering of points given normals and spatial extent. More low-level processing techniques of point data are discussed

*pajarola@acm.org

in [PG01, PKKG03] and fairing in [LWZL02, JDZ04]. However, these methods are aimed at processing moderate point set sizes in main memory and assume that some basic processing such as an initial normal estimation has been done beforehand. An interactive point-processing system is presented in [WPK+04] which, however, is strongly limited in the size of models it can handle as it is designed based on a user feedback loop.

Estimation of vertex attributes such as normal orientation is a common data processing task in polygonal surface reconstruction methods [HDD+92, GKS00, MN03], as is fairing in surface modeling [Tau95, DMSB99, CDR00, SK01, JDD03]. But note that these approaches are not aimed at processing models consisting of tens of millions of vertices and more. In particular, such in-core algorithms do not scale to out-of-core sized point data sets.

An interesting approach to treat points in a sequential order has been presented in [DVS03]. There, a point rendering method has been proposed based on linearizing and sequentially traversing a multiresolution hierarchy. However, the technique only provides display functionality and does not address any low-level point (pre-)processing tasks.

Further related work can be found in processing triangle meshes in a sequential order [IG03, ILGS03]. These techniques grow triangle mesh regions sequentially in a fixed order that limits main memory usage by only keeping a small fraction of the mesh active in-core. These algorithms provide efficient compression [IG03], rendering and simplification [ILGS03] of very large meshes. Again no low-level mesh operators are supported, and more importantly, the techniques do not extend to raw point data processing.

Stream-based data handling is naturally found in processing audio and video data which in contrast to 3D geometry is inherently sequentially organized (based on time). In the context of geometry processing, however, sweep-line (and -plane) techniques in computational geometry (see e.g. [dBvKOS97]) are conceptually the most closely related algorithms. Our basic stream-processing concept follows that idea of sweeping a plane in 3D through the data space and considering the events when a data element is passed by the sweep-plane as outlined in the following section.

3. Streaming Concept

The fundamental idea behind streaming is to process data sequentially with only a limited buffer of active data at any time, resembling a sliding window over the ordered data stream. This allows processing huge data sets very efficiently due to coherent memory-access. Moreover, at any given time it only requires a small fraction of the entire data set to reside in in-core main memory while the remainder rests out-of-core.

Figure 2 illustrates our basic concept of *stream-processing points*. Given an ordered set of points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ each point $\boldsymbol{p}_i$ is read exactly once from the input-stream, kept in an active working set $\mathcal{A}$ (a FIFO queue) for some time, and then written to the output-stream. All processing is limited to points in the working set $\mathcal{A}$. As depicted in Figure 2 we conceptually move a *sweep-plane* through space along the axis of spatial ordering,[1] and when a new point $\boldsymbol{p}_j$ is passed, denoting an event in classical line-sweep algorithms [dBvKOS97], it is added to the working set $\mathcal{A}$. See also Figure 1 for an illustration of this concept.

The active set $\mathcal{A} = \{\boldsymbol{p}_{j-m}, \ldots, \boldsymbol{p}_j\}$ is continuously monitored and local operators are applied to points in $\mathcal{A}$ as elaborated in the following sections. Furthermore, as soon as the smallest element $\boldsymbol{p}_{j-m} \in \mathcal{A}$ cannot possibly contribute anymore to an operation on any subsequent point $\boldsymbol{p}_{i>j}$ it can safely be written to the output stream (we will use *small* and *large* with respect to the sequential index i of the ordered points $\boldsymbol{p}_i$). Note that all points $\boldsymbol{p}_i \notin [j-m, j]$ (or $\boldsymbol{p}_i \notin \mathcal{A}$) which have not yet been read from the input stream, or that have been written to the output stream can reside out-of-core (e.g. in a virtual-memory mapped file). On the other hand, points *living* in the working set $\mathcal{A}$ reside in main memory and extra data is temporarily stored with them such as neighborhood information and other attributes.

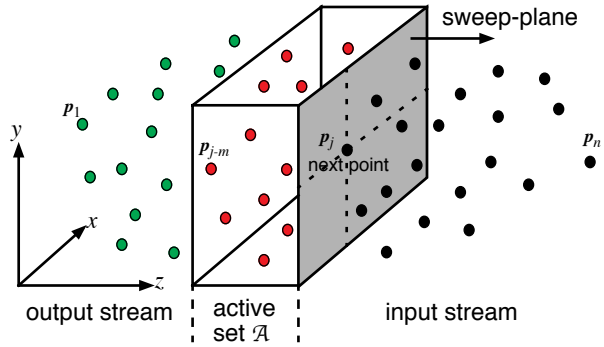

Figure 2. Sweep-plane process overview: unprocessed points are read sequentially from input stream, processed points are written to output stream.

Since the active set $\mathcal{A}$ is orders-of-magnitude smaller then the entire data set, $|\mathcal{A}| = m << n$, it can be maintained efficiently in main-memory even for very large data sets. Moreover, because input and output are streams of points this directly leads to an out-of-core framework for stream-processing huge point set data.

In as much as raw point data sets rarely come with the necessary structure of being sequentially ordered in space, they must be ordered in a pre-process. This can efficiently be done for very large data sets by external sort techniques [Knu98,Vit01], and in practice the *rsort* [Lin96] implementation has been used for similar tasks.

4. Stream Operators

4.1 Definitions

The class of data processing functions supported by the proposed stream-processing concept includes operations performing a computation on a point (or set of points) which only require access to a locally restricted set of neighbors. Or more formally:

Definition 4.1 A *local operator* $\Phi(\boldsymbol{p}_i)$ performs a function on a point $\boldsymbol{p}_i$ that computes or updates a subset of attributes A_i associated with $\boldsymbol{p}_i$. As function parameters, $\Phi(\boldsymbol{p}_i)$ only accepts $\boldsymbol{p}_i$, A_i and a set of points $\boldsymbol{p}_j \in N_i$ within close spatial proximity to $\boldsymbol{p}_i$ (and all their associated attributes A_j).

The *neighborhood* set N_i of points close to $\boldsymbol{p}_i$ may be defined for example as the k-nearest neighbors, or all points $\boldsymbol{p}_j$ within a given distance d of $\boldsymbol{p}_i$. The parameters k and d will generally be specified by the user on an application level but could as well be given individually for each point as suggested in [MN03, AGPS04]. The modifiable attributes A_i can include a wide variety of parameters such as normal orientation or splat size. The above definition of a local operator $\Phi(\boldsymbol{p}_i)$ allows it to be applied to a point $\boldsymbol{p}_i \in \mathcal{A}$ for which all elements of N_i are also part of the current working set, $N_i \subseteq \mathcal{A}$. This formulation includes a wide range of operators for surface parameter estimation and filtering which are amongst the most important tasks in processing raw point cloud data.

In our stream-processing framework, a series of local operators $\Phi_1, \ldots, \Phi_p$ can be concatenated and applied in succession to a stream of points as illustrated in Figure 3. In this context, each

1. Without restricting the generality of the stream-processing concept we assume ordering along the z-axis throughout the paper.

operator Φ_k also acts as a sequential FIFO queue buffer Q_k on the point stream and satisfies the following:

Definition 4.2 A local operator $\Phi_k(\boldsymbol{p}_i)$ is *streamable* if it is computed in one single invocation on $\boldsymbol{p}_i$ and not called recursively on any point $\boldsymbol{p}_j \in N_i$. Additionally, the FIFO semantic of its queue Q_k behavior ensures no interference between consecutive operators $\Phi_{k\pm1}$.

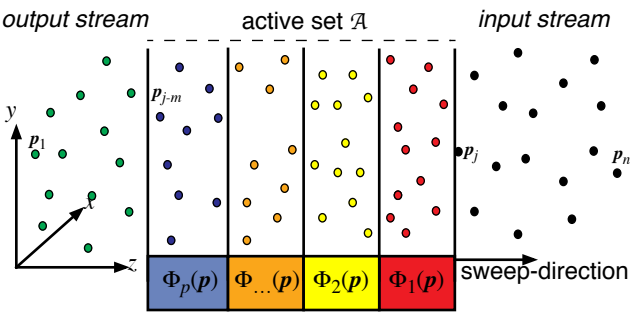

Figure 3. Conceptual stream-processing pipeline: A point $\boldsymbol{p}_i$ moves from right-to-left through the staged stream operators $\Phi_{1\ldots p}$.

The second part of Definition 4.2 deserves further explanation, and is put in practical context in Section 5. It is clear from the above definitions that a stream operator $\Phi_k(\boldsymbol{p}_i)$ postulates the proper existence of the local neighborhood N_i and any required attributes of A_i being part of the input data or computed by preceding stream operators $\Phi_{l<k}(\boldsymbol{p}_i)$ to work. Hence on an application level the correct order of stream operators and the efficient management of attributes must be assured, which is outlined in Section 5.1.

Moreover, each stream operator Φ_k must make sure that a point $\boldsymbol{p}_i$ is only passed on to the next operator Φ_{k+1} if $\boldsymbol{p}_i$ has fully been processed and all affected attributes are updated by Φ_k. This is facilitated and achieved by the FIFO queue semantic Q_k of each operator Φ_k. Note also that while $\boldsymbol{p}_i \in Q_k$ (the buffer of operator Φ_k) it may be that its local neighbor points $\boldsymbol{p}_j \in N_i$ belong to buffers $Q_{k\pm1}$ of pre- or succeeding operators $\Phi_{k\pm1}$. This overlap of local neighborhood sets N_i between consecutive stream operators is indicated in our figures (e.g. in Figure 4) by shingling boxes with cut-out lower-left and upper-right corners. Implementation issues of this dependency between subsequent operators and realization of correct buffer handling is discussed in Section 5.2.

4.2 Fundamental Stream Operators

As previously outlined, establishing the neighborhood N_i of points $\boldsymbol{p}_i$ entering the active set $\mathcal{A}$ from the input stream is of central importance, as is the decision when the smallest point of $\mathcal{A}$ can be written to the output stream. We will treat these constraints in the fundamental I/O and neighborhood-setup operators of our stream-processing framework. In between these bounding operators any regular local operators Φ as defined above can be applied to the point stream as discussed in Section 4.3.

4.2.1 I/O Operators

The canonical first and last stream operators in any stream-processing pipeline facilitate I/O from the input and to the output streams. As depicted in Figure 4 the simple *read operator* $\Phi_R(\boldsymbol{p}_j)$ only needs to read and buffer the next new point $\boldsymbol{p}_j$ entering the active set $\mathcal{A}$ from the input stream. On demand it is passed to the following stream operator and the next point is read from the input stream.

As evident from the definitions of stream operators $\Phi(\boldsymbol{p}_i)$ that operate on the neighborhood N_i, any stream-processing stage following $\Phi(\boldsymbol{p}_i)$ has to make sure that no elements of N_i are altered until $\Phi(\boldsymbol{p}_i)$ has completed. In particular, a point $\boldsymbol{p}_{j-m}$ scheduled to leave the active set $\mathcal{A}$ must be processed carefully. Therefore, we introduce the *deferred-write operator* Φ_W (last in sequence of stream operators). This operator, as illustrated in Figure 4, makes sure that any point $\boldsymbol{p}_{j-m}$ is only removed from $\mathcal{A}$ and written to the output stream if not used by any prior stream operator. That is if $\boldsymbol{p}_{j-m} \notin \cup_i N_i$ for all $\boldsymbol{p}_i$ in prior operator stages Φ_{p-1} to Φ_1. The deferred-write operator is implemented by a simple FIFO queue (see also Section 5.2.2). As soon as a point $\boldsymbol{p}_{j-m}$ can be removed from $\mathcal{A}$, its attributes can be written to the output stream and its main memory can be freed.

4.2.2 Neighborhood Operator

As per definition, any local operator $\Phi(\boldsymbol{p}_i)$ may require a local neighborhood of points $N_i = \{\boldsymbol{p}_{i_1}, \ldots, \boldsymbol{p}_{i_k}\}$ surrounding $\boldsymbol{p}_i$. The neighborhood N_i can either be defined by a maximal *range* or as a number k of nearest neighbors. We will briefly outline the k-nearest neighbors here but a fixed or flexible *range* can also be supported efficiently by varying k for each point (e.g. based on [MN03, AGPS04]). We treat the computation of N_i in our stream-processing framework as a special *neighborhood operator* $\Phi_X(\boldsymbol{p}_i)$ itself which will generally be the second stream operator just after the read operator Φ_R as in Figure 4.

In the context of stream-processing, we must determine the k-nearest neighbor set N_j of a new point $\boldsymbol{p}_j$ passed by the sweep-plane in the neighborhood processing stage just after insertion into the active point set $\mathcal{A}$. To compute all k-nearest neighbors efficiently, or any neighborhood set for that matter, it is essential to maintain a spatial index structure over the relevant point set for fast spatial (range-) queries. However, since we are processing a stream of points and want the index to hold as few points as possible, we must remove (the smallest) elements from this index at the earliest possible time. Hence the index must as well incorporate a priority-queue in the sequential ordering of points. Furthermore, despite frequent insertions and deletions this spatial data structure must be reasonably balanced for good efficiency. The major challenges are to determine efficient indexing and query algorithms that guarantee a correct (or close approximate) k-nearest neighbor solution.

Our streaming (approximate) k-nearest neighbor approach is summarized by the following process: With respect to Figure 4, at insertion of $\boldsymbol{p}_j$ into $\mathcal{A}$ a *left-sided* k-nearest neighbor set N_j is initialized. However, during this insertion of $\boldsymbol{p}_j$ into the spatial index S we also mutually update the k-nearest neighbor sets N_i of points $\boldsymbol{p}_{i<j}$ already in S with respect to the new point $\boldsymbol{p}_j$. Thus at insertion a left-sided k-nearest neighbor set N_j is computed for $\boldsymbol{p}_j$ which is continuously updated to include the right-sided nearest neighbors from subsequent insertions of points $\boldsymbol{p}_{i>j}$.

As spatial index S we use a kD-heap that combines a dynamic, quasi balanced kD-tree with a priority heap. A kD-tree was chosen due to its efficient incremental insertion and removal operations, and the ability to influence its structural balance dynamically. In fact, since points are streamed in one dimension it makes sense to have a two-dimensional kD-tree partitioning the sweep-plane. That is because the streaming dimension of set $\mathcal{A}$ has an extremely small extent compared to the other two dimensions. Furthermore, a priority-queue over the stream indices i of points $\boldsymbol{p}_i \in S$ is integrated with the kD-tree as a heap that parallels the kD-tree binary tree structure.

Two basic operations that are supported are: incremental insertion of a new element into the kD-tree, and removal of an arbitrary element while satisfying the kD-tree structure [dBvKOS97]. Both operations are followed by a bottom-up update of the embedded priority-heap relation.

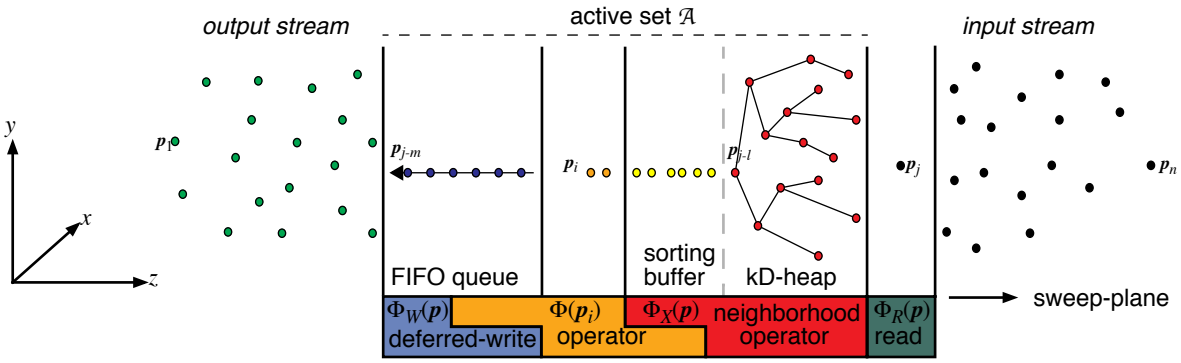

Figure 4. Simple stream-processing pipeline with fundamental stream operators for reading $\Phi_R(\boldsymbol{p})$, writing $\Phi_W(\boldsymbol{p})$ and establishing all k-nearest neighbor sets by $\Phi_X(\boldsymbol{p})$ enclosing a regular stream operator $\Phi(\boldsymbol{p}_i)$.

A more complex operation is the k-nearest neighbor update which is solved in two phases. In the first phase, just before the new element $\boldsymbol{p}_j$ from the input stream is inserted, a query on S finds the left-sided k-nearest neighbors N_j which have all smaller indices – since at that time S only contains prior points in the sequential ordering. The element $\boldsymbol{p}_j$ is then inserted into S with its current left-biased neighborhood N_j. This starts the second k-nearest neighbor search phase where for any point $\boldsymbol{p}_i$ in the kD-heap its N_i is continually updated for the right-sided k-nearest neighbors as follows: During the left-sided query for a new element $\boldsymbol{p}_j$, when $\boldsymbol{p}_i$ is tested to be included in N_j ($i < j$), simultaneously $\boldsymbol{p}_j$ is tested to update N_i on the right side of $\boldsymbol{p}_i$ ($j > i$).

The last major operation on the kD-heap index S is the removal of processed elements. As it is imperative to keep the size of S as small as possible for fast k-nearest neighbor updates, we remove elements whose k-nearest neighborhood is completed as early as possible. Therefore, our kD-heap supports a query to find the list L of elements $\boldsymbol{p}_i$ in S for which the current sweep-plane has moved beyond the farthest kth-nearest neighbor in N_i. The elements of this list L can then be removed from S. However, note that L is not completely sorted with respect to the sequential stream ordering of points. Hence its elements cannot immediately be passed to the next operator stage. Therefore, the elements $\boldsymbol{p}_i \in L$ are passed to a sorting buffer B as depicted in Figure 4, which re-establishes the global stream ordering. The smallest element $\boldsymbol{p}_i$ of B has regained its correct global stream ordering as soon as its index i is smaller than the smallest index in S, and then it can be passed to the next stream operator.

In summary, the outlined neighborhood operator $\Phi_X(\boldsymbol{p}_i)$ incorporates a kD-tree/priority-queue data structure for fast all k-nearest neighbors estimation and a sorting buffer to stream-order the processed points. Its FIFO queue semantic is further explained in Section 5.2.2. The advantages of this solution include:

- the spatial index exhibits a strong k-nearest neighbor query selectivity,
- the kD-tree allows dynamic insertions of new points while preserving a quasi-balanced search structure,
- the embedded priority-heap structure supports fast and early removal of processed elements, and
- the heap and sorting buffer preserve the globally sorted stream-index property dynamically.

4.3 Regular Stream Operators

Given the local neighborhood N_i of points $\boldsymbol{p}_i$ in the active set $\mathcal{A}$, many stream operators $\Phi(\boldsymbol{p}_i)$ are conceivable based on our definitions in Section 4.1, and we outline a small set of meaningful operators that we implemented in our stream-processing points framework. These range from simple normal and curvature estimation, to anisotropic smoothing, and for each we discuss its dependencies and constraints in the stream-processing framework. This extensible list of important operators shows the power and applicability of the proposed stream-processing concept.

4.3.1 Normal Estimation

As prototypical local operator we first introduce the normal estimation $\Phi_N(\boldsymbol{p}_i)$ which is an essential part of processing raw points. Variations of plane fitting have been used in most approaches to estimate the local normal orientation $\boldsymbol{n}_i$ of a point $\boldsymbol{p}_i$ in point cloud data [ABCO+01, PGK02, MN03, PKKG03] and is a standard in surface reconstruction techniques (e.g. as in [HDD+92]).

A *local least squares* (LLS) fit of a plane to a point $\boldsymbol{p}_i$ and its k-nearest neighbors $N_i = \{\boldsymbol{p}_{i_1}, \ldots, \boldsymbol{p}_{i_k}\}$ is defined by the eigenvalue analysis and eigenvector decomposition of the covariance matrix M_i over $\boldsymbol{p}_i$ and N_i. Similar to [ABCO+01], we express a *moving least squares* (MLS) representation of the covariance as weighted sum:

$$M_i = |N_i|^{-1} \cdot \sum_{\boldsymbol{p}_j \in N_i} (\boldsymbol{p}_j - \boldsymbol{p}_i) \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i)^{\mathrm{T}} \cdot \theta(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \,. \qquad \textbf{(1)}$$

The weight function $\theta(r)$ is a smooth, radially symmetric, monotone decreasing Gaussian function $\theta(r) = \mathrm{e}^{-r^2/2\sigma^2}$, with variance σ^2 adaptively defined as the local point density estimate $\sigma^2 = \pi \cdot \mathrm{MAX}_{\boldsymbol{p}_j \in N_i} |\boldsymbol{p}_j - \boldsymbol{p}_i|^2 / |N_i|$ as suggested in [MN03]. Thus the normal $\boldsymbol{n}_i$ of a point $\boldsymbol{p}_i$ is computed as eigenvector of the MLS covariance M_i over N_i corresponding to the smallest eigenvalue of M_i. Potentially it is numerically more robust to define the normal as the normalized vector product of the two eigenvectors corresponding to the two largest eigenvalues of M_i.

A LLS or MLS based normal estimation stream operator $\Phi_N(\boldsymbol{p}_i)$, together with the read, neighborhood and deferred-write fundamental operators, constitutes one of the most basic stream-processing pipeline configurations as illustrated in Figure 4 that performs a meaningful operation on a point set.

As the normal operator $\Phi_N(\boldsymbol{p}_i)$ does not modify any neighborhood information but merely computes a new attribute, the normal $\boldsymbol{n}_i$ of a point $\boldsymbol{p}_i$, its FIFO semantic as introduced in Section 4.1 is very simple: After processing a new point $\boldsymbol{p}_i$, it is directly released to the subsequent stream operator $\Phi_{N+1}(\boldsymbol{p}_i)$ and $\boldsymbol{p}_i$ is only buffered if not consumed immediately by Φ_{N+1}. While a buffer is implemented as discussed in Section 5.2.1, it will generally not buffer any additional points as indicated in Figure 5.

4.3.2 Curvature Estimation

Another elementary operator is the estimation of curvature and its principal directions. Our curvature operator $\Phi_C(\boldsymbol{p}_i)$ implements a curvature estimation based on the covariance of normals $\boldsymbol{n}_j$ of nearest points $\boldsymbol{p}_j \in N_i$. Similar to the normal estimation above we define a MLS representation of the normal covariance as:

$$M_i = |N_i|^{-1} \cdot \sum_{\boldsymbol{p}_j \in N_i} \boldsymbol{n}_j \cdot \boldsymbol{n}_j^{\mathrm{T}} \cdot \theta(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \,. \qquad \textbf{(2)}$$

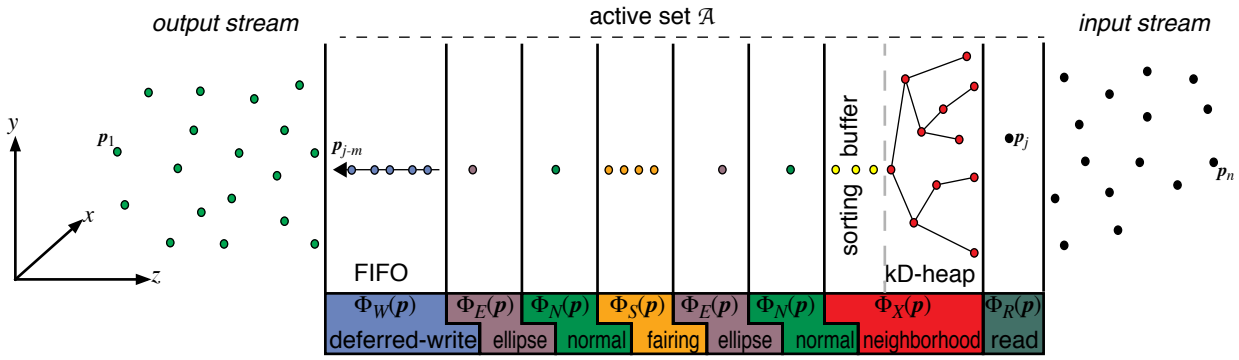

Figure 5. Stages of a complex stream-processing pipeline for fairing, including a smoothing operator $\Phi_S(\boldsymbol{p})$ enclosed by normal and splat size operators $\Phi_N(\boldsymbol{p})$ and $\Phi_E(\boldsymbol{p})$.

The singular value decomposition of the covariance of normals of Equation 2 gives us an estimate of the curvatures and its principal directions. Figure 1 illustrates the principal curvatures (root mean square curvature (RMS), mean or absolute curvature) on the David point data.

As the curvature operator $\Phi_C(\boldsymbol{p}_i)$ depends on the normals $\boldsymbol{n}_j$ of all k-nearest points $\boldsymbol{p}_j \in N_i$, and hence may follow a normal operator Φ_N, its FIFO semantic is as follows: New points pulled from the prior stream operator $\Phi_{C\text{-}1}$ are buffered until the entire neighborhood N_i has been processed by $\Phi_{C\text{-}1}$ before $\Phi_C(\boldsymbol{p}_i)$ is applied. This pre-buffering constraint is further described in Section 5.2.2.

4.3.3 Splat Size Estimation

High-quality point-based rendering (PRB) techniques display a surface from points by rendering and blending overlapping (elliptical) disks (see also overview [SPL04, SP04]). The elliptical extent of a point $\boldsymbol{p}_i$ can be derived from locally computed Voronoi cells as in [DGH01, DH02], however, given the local neighborhood N_i, a covariance analysis [PGK02, Paj03, PSG04] is more suitable for implementation as an elliptical splat-estimation stream operator $\Phi_E(\boldsymbol{p}_i)$ in our stream processing framework.

We can determine the ellipse parameters such as major and minor axis directions, major axis length and aspect ratio for a point $\boldsymbol{p}_i$ efficiently from the eigenvalue and eigenvector decomposition of the (MLS weighted) covariance matrix M_i given in Equation 1. The eigenvectors, of the covariance M_i projected into the tangent plane given by the normal $\boldsymbol{n}_i$, define the ellipse axis while the eigenvalues determine the aspect ratio. The so defined elliptical disk has then to be scaled to fit the neighbor set N_i. The beauty of the approach in [Paj03, PSG04] can largely be preserved in this context.

Alternatively, if we have a curvature operator Φ_C preceding the splat estimation $\Phi_E(\boldsymbol{p}_i)$ then the ellipse axis directions and their aspect ratio can be inferred from the principal curvatures derived from Equation 2. In that case only an elliptical disk scaling has to be performed to fit the neighbor set N_i.

A normal estimation stream operator $\Phi_N(\boldsymbol{p}_i)$ combined with a splat-estimation stream operator $\Phi_E(\boldsymbol{p}_i)$ compute the essential per-vertex attributes necessary for efficient high-quality PBR algorithms as surveyed in [SPL04, SP04].

Similar to the normal operator, $\Phi_E(\boldsymbol{p}_i)$ computes new point attributes and its FIFO semantic resembles the one of the normal operator described in the previous Section 4.3.1.

4.3.4 Fairing

To demonstrate the power of the proposed stream-processing framework we introduce a smoothing filter operation Φ_S. Point cloud data from high-resolution 3D scanners or imaging devices tend to have a non-negligible amount of noise. To filter noise artifacts many smoothing algorithms have been proposed for meshes such as [Tau95], [DMSB99], [CDR00] or [SK01] just to mention a few of the major approaches. In general these techniques require a manifold mesh representation and are based on iterative numerical methods. In [PG01], fairing of points has been proposed which, however, requires a regular (re-) sampling pattern. For an efficient stream operator based smoothing, specific assumptions on the sampling pattern, iteration or recursion have to be avoided.

Therefore, we adopt the non-iterative feature preserving fairing operator presented in [JDD03], as also proposed in [JDZ04]. Its applicability to triangle soups makes it suitable for adaptation to unorganized point sets, and it conforms to the basic steam operator constraints outlined in Section 4.1. Fairing of points $\boldsymbol{p}_i$ along [JDD03] requires tangent plane normal estimates $\boldsymbol{n}_i$ which we already addressed in Section 4.3.1. Furthermore, it also requires accessing spatial neighbors around $\boldsymbol{p}_i$ which is provided by the k-nearest neighborhood N_i outlined in Section 4.2.2.

Given a point $\boldsymbol{p}_i$ and its neighbors N_i, we directly extend the smoothing operation of [JDD03] to points as follows

$$\boldsymbol{p}_i' = \Phi_S(\boldsymbol{p}_i) = \frac{1}{w_i} \cdot \sum \Pi_j(\boldsymbol{p}_i) a_j \cdot f(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \cdot g(|\Pi_j(\boldsymbol{p}_i) - \boldsymbol{p}_i|)\,, \quad \textbf{(3)}$$

with summation over all points $\boldsymbol{p}_j \in N_i \cup \boldsymbol{p}_i$. The operator $\Pi_j(\boldsymbol{p}_i)$ denotes the projection of $\boldsymbol{p}_i$ onto the tangent plane of point $\boldsymbol{p}_j$ and the value a_j corresponds to an area weight (i.e. the elliptical splat disk size). The normalization term w_i is the sum of weights $\sum a_j \cdot f(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \cdot g(|\Pi_j(\boldsymbol{p}_i) - \boldsymbol{p}_i|)$. The Gaussian weight function $f(r)$ adjusts the influence of a point based on spatial distance and favors nearby points for smoothing, while $g(r)$ tends to preserve sharp features by giving less weight to points with different normal orientations [JDD03].

Note, however, that the fairing operator $\Phi_S(\boldsymbol{p}_i)$ must fit into a properly configured stream-processing pipeline as illustrated in Figure 5. In particular, applying the fairing operator $\Phi_S(\boldsymbol{p}_i)$ calls for recomputation of new normals $\boldsymbol{n}_i$, as well as (elliptical) bounding disk parameters. Hence we apply normal and splat size estimation Φ_N and Φ_E not only before, but also again after the fairing operator Φ_S as shown in Figure 5.

Moreover, since the fairing operator $\Phi_S(\boldsymbol{p}_i)$ changes the coordinates of a point $\boldsymbol{p}_i$ it must strictly be avoided that any pre- or succeeding stream operators $\Phi_{S\pm1}(\boldsymbol{p}_j)$ act on a mix of pre- and post-faired points $\boldsymbol{p}_i \in N_j$. This constraint must correctly be satisfied by the FIFO queue semantic of Φ_S as described in Section 5.2.2. Hence Φ_S in general buffers a limited set of points as indicated in Figure 5.

5. Implementation

A major challenge in the context of stream-processing points as outlined in Section 4 is the systematic definition and development of stream operators. In particular, this includes:

1. defining a common implementation framework and interface such that local stream operators $\Phi_k(\boldsymbol{p}_i)$ can be concatenated and plugged into a stream-processing system like modules, and
2. concealing the dependencies between consecutively applied local stream operators $\Phi_1, \ldots, \Phi_p$ effectively within the stream-operator abstract data types.

In this section we address the above challenges by describing the implementation of our stream-processing points framework based on the definition of a stream-operator abstract data type.

5.1 Attribute Handling

As mentioned in Section 4, different stream operators $\Phi_k(\boldsymbol{p}_i)$ add or modify different subsets of attributes $a_i^k \subseteq A_i$ of points $\boldsymbol{p}_i$. These may be additional to the ones provided initially in the input point stream. Moreover, some attributes may only be needed temporarily and need not be written to the output point stream at all. Therefore, we define the stream-point data type as an extensible set of attribute-field structures as outlined below and exemplified in Figure 6.

InputFields Defines the *initial* point attributes a_i^{in} given for each point $\boldsymbol{p}_i$ in the input stream.

<name>OpFields Specifies the *temporary* attributes $a_i^{k_{\text{aux}}}$ computed by stream operator $\Phi_k(\boldsymbol{p}_i)$ for points $\boldsymbol{p}_i$ in the active set $\mathcal{A}$ but not written to the output stream.

<name>OpOutFields Lists the *added* attributes $a_i^{k_{\text{out}}}$ computed by stream operator $\Phi_k(\boldsymbol{p}_i)$ for each point $\boldsymbol{p}_i$ which are passed along with the point $\boldsymbol{p}_i$ to the output stream.

AuxiliaryFields Consists of all auxiliary attributes $a_i^{\text{aux}} = \cup a_i^{k_{\text{aux}}}$ computed and required by any stream operator $\Phi_k(\boldsymbol{p}_i)$ while a point $\boldsymbol{p}_i$ is in the active set $\mathcal{A}$ and processed by operators $\Phi_1, \ldots, \Phi_p$.

OutputFields Includes all attributes $a_i^{\text{out}} = \cup a_i^{k_{\text{out}}} \cup a_i^{\text{in}}$ of a point $\boldsymbol{p}_i$ that have to be written to the output stream.

AllFields Covers all attributes $A_i = a_i^{\text{all}} = a_i^{\text{out}} \cup a_i^{\text{aux}}$ that are ever referenced by any stream operator while processing point $\boldsymbol{p}_i$.

This design of extensible per-point attribute fields supports the flexibility that a stream-processing framework with varying configurations of stream operators requires. On an application-level, the definition of these attribute fields has to be matched with the configuration of the stream-processing pipeline. For example, the splat ellipse estimation operator $\Phi_E(\boldsymbol{p}_i)$ bases its computation on the existence of a normal $\boldsymbol{n}_i$ which requires a precursory normal operator Φ_N and the proper inclusion of its attributes $a_i^{N_{\text{aux}}}$ and $a_i^{N_{\text{out}}}$.

An example point-attribute configuration is given in Figure 6 for a normal computation, elliptical splat estimation and fairing pipeline consisting of: stream reading Φ_R, neighborhood generation Φ_X, normal computation Φ_N, splat size estimation Φ_E, fairing Φ_S and deferred stream writing Φ_W operators. The right configuration and order of operators is depicted in Figure 5. As part of the auxiliary fields a_i^{aux}, the reader Φ_R assigns an index i to each point $\boldsymbol{p}_i$ in the order it is read from the input stream. The k-nearest neighbor operator $\Phi_X(\boldsymbol{p}_i)$ computes all auxiliary fields with respect to a point $\boldsymbol{p}_i$'s neighborhood information N_i. In particular, this also includes the min and max referenced indices j of its nearby points $\boldsymbol{p}_j \in N_i$ which's use is further detailed in the following sections. The normal operator $\Phi_N(\boldsymbol{p}_i)$ computes the normal attribute $\boldsymbol{n}_i$ which is part of the output a_i^{out}, based on covariance information which is stored as part of a_i^{aux}. The elliptical splat estimator Φ_E is also based on normal and covariance information and outputs ellipse major axis, its length and aspect ratio as part of a_i^{out}. For its calculation, the fairing operator $\Phi_S(\boldsymbol{p}_i)$ adds temporary attributes to a_i^{aux} consisting of a copy of the original point position and an area weight.

```
struct InputFields {
  Vector3f v;          // position
  Color3u c;           // color
};

struct ReadOpFields {
  int index;           // element's index i in input stream
};

struct NeighborOpFields {
  int cnt;                          // number of neighbors
  AllFields* list[MAX_K];           // pointers to neighbors
  float dist[MAX_K];                // distances to neighbors
  int min_index;               // smallest referenced index
  int max_index;               // largest referenced index
};

struct NormalOpFields {
  Matrix4d covar;              // covariance information
};

struct NormalOpOutFields {
  Vector3f n;          // normal
};

struct SplatOpOutFields {
  Vector3f axis;       // major ellipse semiaxis orientation
  float length;        // major ellipse semiaxis length
  float ratio;         // semiaxis aspect ratio
};

struct FairOpFields {
  Vector3f position;       // copy of original position
  float area;              // splat area weight
};

struct AuxiliaryFields : ReadOpFields,
   NeighborOpFields, NormalOpFields,
   FairOpFields {};

struct OutputFields : InputFields,
   NormalOpOutFields, SplatOpOutFields {};

struct AllFields : AuxiliaryFields,
   OutputFields {};
```

Figure 6. Attribute-field structures of stream-points for a normal computation, elliptical splat estimation and fairing stream-processing pipeline as illustrated in Figure 5.

5.2 Stream Operator Classes

As introduced in Section 4.1, each stream operator Φ_k behaves like a buffer Q_k on the stream of points. After being released from the previous operator Φ_{k-1} – respectively its buffer Q_{k-1} – a point $\boldsymbol{p}_i$ enters the next queue Q_k. When all necessary neighborhood conditions are met, operator $\Phi_k(\boldsymbol{p}_i)$ is performed. The conditions when a point $\boldsymbol{p}_i \in Q_k$ can be processed by $\Phi_k(\boldsymbol{p}_i)$ and when it is released to the subsequent operator Φ_{k+1} and its queue Q_{k+1}, depend on the class of the stream operator Φ_k as outlined below.

The semantic of the buffer Q_k of a stream operator Φ_k is equivalent to a FIFO queue, and the operators' interface is given in Figure 7 which includes the traditional *front*() and *pop_front*() methods. However, instead of using a standard *push_back*() interface we defined the exchange of points between subsequent operators as a *pull-push* mechanism, see also Section 5.3. For this purpose, each operator Φ_k keeps a reference to the previous operator Φ_{k-1} in the stream-processing pipeline. The semantics and implementations of these queue operations are further discussed next. Additional stream-operator functionality includes queries on the *smallest element* – index i of a queued point $\boldsymbol{p}_i \in Q_k$ – on which operator Φ_k has not yet actually been computed; and the *smallest referenced* neighbor – index j of a $\boldsymbol{p}_j \in \cup N_i$ – of any *unprocessed* points $\boldsymbol{p}_i$ in Q_k.

```
class StreamOperator {
public:
  StreamOperator();
  virtual ~StreamOperator();

  virtual void pull_push();
  virtual AllFields* front();
  virtual void pop_front();

  virtual int smallest_element();
  virtual int smallest_reference();

protected:
  StreamOperator *prev;
};
```

Figure 7. Abstract common interface definition of the virtual stream operator base-class.

5.2.1 Through-buffer Operators

All simple stream operators $\Phi_k(\boldsymbol{p}_i)$ that given a set of attributes $A_i \setminus a_i^k$ compute additional new attributes a_i^k for a point $\boldsymbol{p}_i$ without affecting any nearest neighbor data in N_i are called *through-buffer* operators. This arises from the fact that as soon as a point $\boldsymbol{p}_i$ is released from a prior operator $\Phi_{k\text{-}1}$ it can be processed by Φ_k and immediately released to Φ_{k+1}. The *pull_push*() outline for simple through-buffer stream operators is given in Figure 8. Note that this simple stream operator class implements a FIFO queue to buffer points $\boldsymbol{p}_i$ after being processed by Φ_k. However, in practice this buffer will generally be empty as the subsequent operator Φ_{k+1} consumes the released points immediately.

```
class ThroughBuffer : public StreamOperator {
public:
  virtual void pull_push();
  virtual AllFields* front();
  virtual void pop_front();

  virtual int smallest_element();
  virtual int smallest_reference();

protected:
  deque<AllFields*> FIFO;
};

void ThroughBuffer::pull_push() {
  AllFields *tmp;

  // pull elements from previous stream operator
  while (tmp = prev->front()) {
    prev->pop_front();

    // perform stream operator function
    applyOperator(tmp);
    FIFO.push_back(tmp);
  }
}
```

Figure 8. Class definition and pull-push method of a through-buffer type stream operator.

The standard FIFO queue *front*() and *pop_front*() methods are straightforward implementations for a through-buffer stream operator Φ_k as shown in Figure 9. Basically the FIFO buffer of processed points $\boldsymbol{p}_i$ is queried and if not empty the points are released in order (to the calling operator Φ_{k+1}). Since a through-buffer operator Φ_k does not queue any unprocessed points $\boldsymbol{p}_i$ both index-reference queries are passed to any prior stream operator $\Phi_{k\text{-}1}$ in the stream processing pipeline.

Normal computation as well as elliptical splat-estimation stream operators described in Sections 4.3.1 and 4.3.3 belong to this through-buffer stream operator category. The read operator (Section 4.2.1) is an even simpler through-buffer implementation as it reads and buffers one point at a time from the input stream.

5.2.2 Pre- and Post-buffer Operators

More complex are the FIFO queue implementations for stream operators $\Phi_k(\boldsymbol{p}_i)$ that either affect the use of $\boldsymbol{p}_i \in N_j$ in processing other nearest-neighbor related points $\boldsymbol{p}_j$ by $\Phi_{k\pm1}(\boldsymbol{p}_j)$, or that modify the neighbor data N_i of the current point $\boldsymbol{p}_i$. We observe that:

```
AllFields* ThroughBuffer::front() {
  AllFields *tmp = NULL;

  if (!FIFO.empty())
    tmp = FIFO.front();
  return tmp;
}

void ThroughBuffer::pop_front() {
  if (!FIFO.empty())
    FIFO.pop_front();
}

int ThroughBuffer::smallest_element() {
  if (prev)
    return prev->smallest_element();
  else
    return INT_MAX;
}

int ThroughBuffer::smallest_reference() {
  if (prev)
    return prev->smallest_reference();
  else
    return INT_MAX;
}
```

Figure 9. FIFO queue access and index-reference methods for through-buffer type stream operators.

```
class PrePostBuffer : public StreamOperator {
public:
  virtual void pull_push();
  virtual AllFields* front();
  virtual void pop_front();

  virtual int smallest_element();
  virtual int smallest_reference();

private:
  deque<AllFields*> FIFO1;
  deque<AllFields*> FIFO2;
  HeapOfPairs HEAP;
};

void PrePostBuffer::pull_push() {
  AllFields *tmp;

  // pull elements from previous stream operator
  while (tmp = prev->front()) {
    prev->pop_front();

    // update heap that maintains smallest referenced index
    HEAP.push(tmp->min_ref_index, tmp);

    // defer processing points
    FIFO1.push_back(tmp);
  }

  // check queue of deferred points
  while (!FIFO1.empty()) {
    tmp = FIFO1.front();

    // only update elements fully processed by prior operator
    if (tmp->max_ref_index < prev->smallest_element() &&
        tmp->index < prev->smallest_reference()) {
      FIFO1.pop_front();

      // perform stream operator function
      applyOperator(tmp);

      // transfer to post-buffer
      FIFO2.push_back(tmp);
    } else
      break;
  }
}
```

Figure 10. Outline of class definition and pull-push method of a pre- and post-buffer type stream operator.

1. First, such operators must defer processing a point $\boldsymbol{p}_i$ until all its neighbors $\boldsymbol{p}_j \in N_i$ have been processed by the previous operator $\Phi_{k\text{-}1}$.
2. Second, $\boldsymbol{p}_i \in N_j$ itself must not be accessed by any operator $\Phi_{k\text{-}1}(\boldsymbol{p}_j)$. Finally, point $\boldsymbol{p}_i$ is only released to the subsequent stream operator Φ_{k+1} when it is safe to do so.

Figure 10 demonstrates the underlying pull-push algorithm for stream operators $\Phi_k(\boldsymbol{p}_i)$ that must *pre-* as well as *post-buffer* the processed points $\boldsymbol{p}_i$. Two queues are now necessary to implement the stream operator's buffer Q_k, one for buffering points $\boldsymbol{p}_i$

before and one after applying Φ_k. The *pull_push*() method first gets all points $\boldsymbol{p}_i$ released from the preceding operator Φ_{k-1} and queues them up in FIFO1. Note that at this point the smallest referenced indices j of points $\boldsymbol{p}_j \in N_i$ for all $\boldsymbol{p}_i \in Q_k$ are maintained in a HEAP structure which stores the index j along with the point $\boldsymbol{p}_i$ it is referenced *from*. Next, the queue FIFO1 is checked for available points $\boldsymbol{p}_i$ that can now safely be processed by Φ_k and queued up in FIFO2. This requires testing for smallest unprocessed and smallest referenced indices in the previous operators Φ_{k-1} as pointed out above.

In Figure 11, the FIFO queue *front*() and *pop_front*() methods of a pre- and post-buffer stream operator Φ_k are given. In this context, the next available point via *pop_front*() must come from the FIFO2, the post-buffer, as only this queue keeps the points already processed by Φ_k. In addition, the top-most element $\boldsymbol{p}_i$ of FIFO2 is only released by operator Φ_k if it no more references any point $\boldsymbol{p}_j \in N_i$ which is still in the pre-buffer FIFO1 of Φ_k! This satisfies the constraints that when $\boldsymbol{p}_i$ is released to the next operator $\Phi_{k+1}(\boldsymbol{p}_i)$, Φ_{k+1} will not operate on a neighborhood N_i of $\boldsymbol{p}_i$ consisting of mixed points $\boldsymbol{p}_j$ – with respect to being processed or not by the operator Φ_k.

Additionally, the pop procedure has the task of updating the HEAP of smallest indices referenced by any point $\boldsymbol{p}_i \in Q_k$ after releasing a point $\boldsymbol{p}_i$ to the next operator. This is simply done by removing all elements at the top of HEAP which are no longer members of Q_k.

Finally, we define the smallest and smallest-referenced index functions at the bottom of Figure 11. The smallest index i of an unreleased point $\boldsymbol{p}_i \in Q_k$ of stream operator Φ_k is given either by the smallest element of queues FIFO2 or FIFO1, or passed along to the preceding stream operator Φ_{k-1} in the stream processing pipeline. The smallest referenced index j of points $\boldsymbol{p}_j \in N_i$ for all $\boldsymbol{p}_i \in Q_k$ is explicitly maintained in the above introduced HEAP data structure, and the query takes this HEAP of operator Φ_k as well as the prior operator Φ_{k-1} into account.

```
AllFields* PrePostBuffer::front() {
  AllFields *tmp = NULL;

  if (!FIFO2.empty() && (FIFO1.empty() ||
        FIFO2.front()->max_ref_index < FIFO1.front()->index))
    tmp = FIFO2.top();
  return tmp;
}

void PrePostBuffer::pop_front() {
  if (!FIFO2.empty()) {
    // remove unused references from HEAP
    while (!HEAP.empty() && HEAP.top().second->index
           < FIFO2.front()->index)
      HEAP.pop();
    FIFO2.pop_front();
  }
}

int PrePostBuffer::smallest_element() {
  if (!FIFO2.empty())
    return FIFO2.front()->index;
  else if (!FIFO1.empty())
    return FIFO1.front()->index;
  else
    return prev->smallest_element();
}

int PrePostBuffer::smallest_reference() {
  int index = prev->:smallest_reference();

  if (!HEAP.empty())
    index = MIN(HEAP.top().first, index);
  return index;
}
```

Figure 11. Outline of FIFO queue access and index-reference methods for pre- and post-buffer type stream operators.

The k-nearest neighbors operation and the fairing function described in Sections 4.2.2 and 4.3.4 are pre- and post-buffer stream operators as characterized in this section. The k-nearest neighbors stream operator Φ_X, however, exhibits a few notable differences. First, the queue FIFO1 is replaced by a kD-heap structure as explained in Section 4.2.2 and in the first while loop of the *pull_push*() method in Figure 10 this kD-heap is queried and updated for the points pulled from the preceding read operator. Second, the FIFO2 queue is replaced by a sorting buffer, and the second while loop in the *pull_push*() method is substituted with a query process to remove elements with completed k-nearest neighbor sets from the kD-heap and queuing them in the sorting buffer. As the combination of the kD-heap and sorting buffer fully implement the FIFO semantic Q_X of the k-nearest neighbor operator Φ_X, the queue access and smallest index-reference methods manifest only slight deviations from the basic methods given in Figure 11 and are thus not further explained here.

The curvature operator $\Phi_C(\boldsymbol{p}_i)$ described in Section 4.3.2 is a simplified pre- and post-buffer stream operator in that it only exhibits a pre-buffer constraint to make sure that any point $\boldsymbol{p}_j \in N_i$ has been released from the prior stream operator Φ_{k-1}.

5.3 Stream-Processing Pipeline

Setting up a stream-processing point pipeline is now very simple given the stream-operator framework outlined in the previous sections. Some minimal user-involvement is required to select a proper sequence of stream operators and to make sure the attribute fields discussed in Section 5.1 match the selected operators.

Figure 12 demonstrates the elegant application of our stream-processing framework to process a stream of raw points. In this example a pipeline corresponding to Figure 5 is set up with point attributes as shown in Figure 6. The input and ouput point-streams can be memory-mapped file arrays of InputFields and OutputFields types. The setup tasks include opening the I/O point-streams and initializing the sequence of stream operators. The main processing stage then merely consists of two very simple nested loops: The outer loop over all points consecutively read from the input stream. The inner loop iterating through the sequence of stream operators and invoking their pull-push methods to process and pass points from one to the next stream operator, with the last one writing the points to the output stream.

```
InputFields *pfile = NULL;   // inpout point stream file
OutputFields *sfile = NULL;  // output point stream file
int npoints;                 // number of input points

int main(int argc, char **argv)
{
  int i, nops = 0;
  StreamOperator *operators[8];

  // open input and output point-stream files
  // e.g. as memory mapped file arrays pfile and sfile

  // initialize stream-operator pipeline
  operators[nops++] = new ReadOperator(pfile, nv);
  operators[nops] = new KNearestOperator();
  operators[nops++]->set_prev(operators[nops-1]);
  operators[nops] = new NormalOperator();
  operators[nops++]->set_prev(operators[nops-1]);
  operators[nops] = new SplatOperator();
  operators[nops++]->set_prev(operators[nops-1]);
  operators[nops] = new FairOperator();
  operators[nops++]->set_prev(operators[nops-1]);
  operators[nops] = new NormalOperator();
  operators[nops++]->set_prev(operators[nops-1]);
  operators[nops] = new SplatOperator();
  operators[nops++]->set_prev(operators[nops-1]);
  operators[nops] = new WriteOperator(sfile, nv);
  operators[nops++]->set_prev(operators[nops-1]);

  // main loops for processing stream of points
  while (operators[0]->position() < npoints)
    for (i = 0; i < nops; i++)
      ops[i]->pull_push();
}
```

Figure 12. Outline of main point stream-processing routine for a normal computation, elliptical splat estimation and fairing stream-processing pipeline as illustrated in Figure 5.

6. Analysis

In terms of memory requirements we note that the most critical part to process a large set of points by local operators is a data structure that provides efficient access to all points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ and their nearest neighbors. In general, a balanced hierarchical spatial indexing structure requires O(n) space and allows processing all points and k-nearest neighbors in O($k \cdot n$ log n) time. While this is theoretically optimal it may nevertheless not be the fastest in practice and consume too much main memory for very large n.

Our stream-processing framework exhibits the extremely important property that only a small number of $m \ll n$ points are active at any moment in time. The active set $\mathcal{A} = \boldsymbol{p}_{i-1}, \ldots, \boldsymbol{p}_{i-m}$ consists of points not fully processed for which a new point $\boldsymbol{p}_i$ on the sweep plane may be necessary to complete all operator tasks. Thus in main memory only the m active points must be maintained and organized in an indexing data structure. Hence the expected main memory usage is only in the order of O(m), as only a *sliding window* of m elements is continuously maintained in the active set $\mathcal{A}$. Moreover, as the processing performance is mainly determined by the determination of all k-nearest neighbors, the expected running time is only O($k \cdot n$ log m). This corresponds to a significantly reduced cost for the stream-processing approach.

As reported in the experimental results section below, the computation of all k-nearest neighbors is dominating the overall workload. Therefore, the end-performance will strongly depend on the parameter k (proportionally) and the number $s<m$ (logarithmically) of points in the kD-heap of the nearest neighbor stream operator Φ_X.

7. Experimental Results

All point processing experiments were performed on a 1.8GHz PowerMac G5. Timing was performed using the Unix *clock*() function to measure individual functions within the code, and the */usr/bin/time* Unix command line tool was used to measure the real-world clock time elapsed between invocation and termination of the executable. Hence the total timings even include any time a process spent waiting for events such as completion of I/O operations (and not only the consumed CPU cycles).

7.1 Preprocessing

Pre-process results for ordering some large point data sets are given in Table 1. All data sets are ordered for streaming simply along the dimension of largest extent. Besides the St. Matthew data set, which was converted from a binary QSplat model [RL00], all models were converted from a plain ASCII PLY triangle mesh format. Any information besides the raw point coordinates and color was omitted in that process.

Generally the ordering and streaming of points is implemented using memory mapped arrays. After reading the raw point data from the input mesh, or QSplat file into a file-memory mapped point array, our current implementation of the sorting pre-process uses a quicksort algorithm to sequentially order the points along a given dimension. As shown in Table 1, quicksort on a memory mapped array performs quite well as it accesses the data in a coherent linear way – doing log(n) passes.

Improved pre-process sorting can be achieved by more sophisticated out-of-core techniques [Knu98,Vit01] such as the *rsort* [Lin96] tool that has been used in similar situations, however, this is not the main focus here.

7.2 Stream Processing

7.2.1 Overview

In our different experiments we have tested various stream processing pipelines consisting of stream operators discussed in Section 4. The three different stream-processing pipelines and their sequence of applied stream operators are:

Model	#Points	Mesh File Size	Point Stream File Size	Preprocess	
				reading	sorting
St. Matthew	102,965,801	N/A	1,571MB	35s	93s
David 1mm	28,168,109	2,288MB	430MB	125s	22s
Lucy	14,022,961	1,085MB	214MB	52s	11s
David 2mm	4,129,534	327MB	63MB	19s	3.4s
David head	2,000,646	165MB	30MB	12s	1.5s
Dragon	435,545	32MB	6.7MB	1.6s	0.3s
Female	302,948	24MB	4.6MB	1.8s	0.2s
Balljoint	137,062	11MB	2MB	0.9s	0.1s

Table 1. Input test model and output point stream sizes. Preprocess timing includes converting and sorting point data.

- *Normal*: Φ_R (read), Φ_X (k-nearest neighbors, k=8), Φ_N (normal estimation) and Φ_W (deferred-write).
- *Curvature*: Φ_R, Φ_X (k=8), Φ_N, Φ_C (curvature), Φ_E (elliptical splat estimation) and Φ_W.
- *Fairing*: Φ_R, Φ_X (k=64…384), Φ_N, Φ_E, Φ_S (smoothing), Φ_N, Φ_E and Φ_W.

In Table 2 we give an overview of the time required to process large models with the *Normal* and *Curvature* stream-processing pipelines, as well as the per-point *lifespan* time that indicates for how long on average a point remained in the active set $\mathcal{A}$ while being processed by the different stream operator stages. The table also includes the size of the generated output point streams (see also input point stream file sizes in Table 1).

Model	Pipeline	Point Stream Output Size	Timing	
			Process h:mm:ss	Lifespan sec.
St. Matthew	*Normal*	3,142MB	*5:02:25*	7.56s
	Curvature	6,284MB	*7:51:14*	13.0s
David 1mm	*Normal*	859MB	*2:33:56*	23.62s
	Curvature	1,719MB	*2:52:45*	29.27s
Lucy	*Normal*	428MB	*26:32*	4.78s
	Curvature	856MB	*33:25*	6.17s
David 2mm	*Normal*	126MB	*6:02*	0.62s
	Curvature	252MB	*7:50*	1.36s
David head	*Normal*	61MB	*2:53*	0.66s
	Curvature	122MB	*3:43*	1.45s

Table 2. Overall timing results of stream-processing points, and average lifespans of points in active set $\mathcal{A}$.

7.2.2 Streaming Working Set

As outlined in Sections 3 and 4, a major goal of the proposed stream-processing framework is to drastically reduce the number of points actively referenced at any time to perform a series of local operators on a point set. This limited working set (i.e. main-memory usage) and the coherent streaming access of points allows effective processing as demonstrated in our experiments.

The graphs in Figure 13 show the sizes of the FIFO buffers corresponding to the different stream operators that together define the *Curvature* pipeline working set $\mathcal{A}$ of active points at any time during stream-processing. Note that the read, normal- and splat-estimation (operator) buffers are omitted as they only keep one point at a time (see also Section 5.2.1). As demonstrated impressively by these charts, the stream-operator buffers hardly ever maintain 0.5% of the large point sets in the active set $\mathcal{A}$ (i.e. in main memory). In fact, for the largest St. Matthew model the buffers rarely even reach a size of 2/1000 (or 0.2%) of the overall model size.

Lucy exhibits some strong growth of the active working set $\mathcal{A}$ up to 2% during the first few 100K points at a very early stage. However, it then dramatically drops to only manage on average much less than 20K points dynamically during the remainder of the stream-processing. Peaks in the active working set $\mathcal{A}$ are due to peculiar data distributions in the point streams.

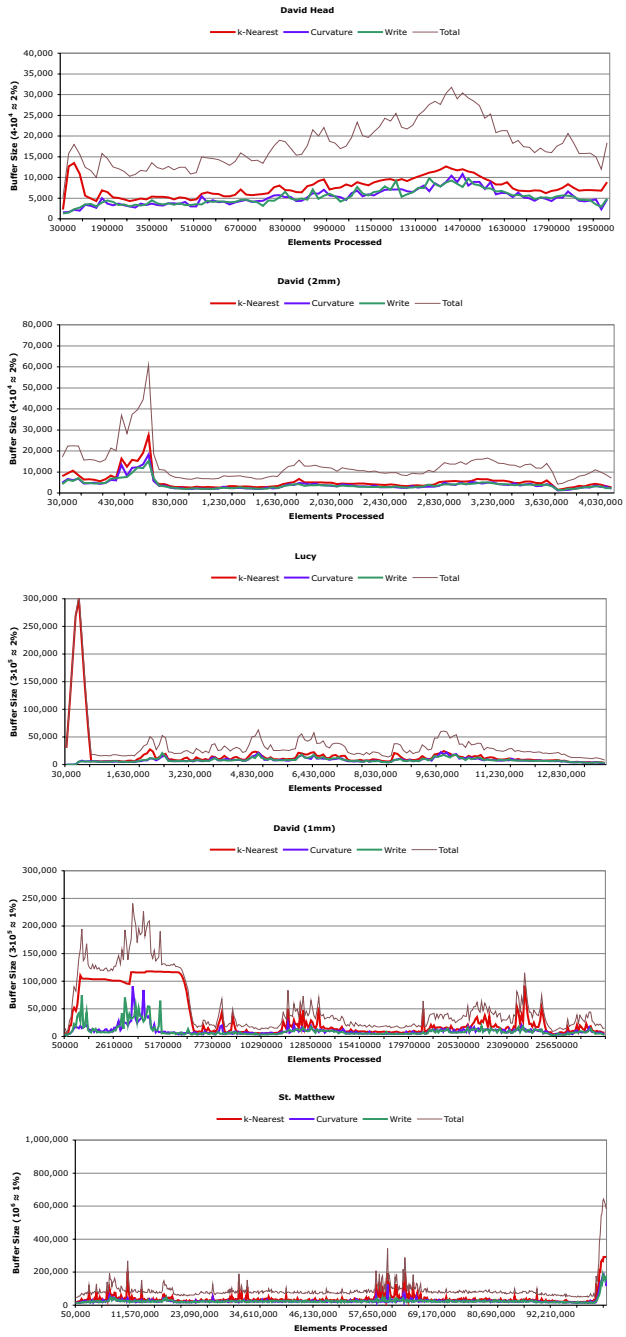

Figure 13. Streaming total active working set and buffer sizes of corresponding stream operators plotted against the progress through the input point stream. (y-axis indicates size only up to 1% or 2% of the entire data set)

7.2.3 Main Memory In-Dependence

To back our claim of effective stream-processing of very large point sets we carried out two sets of experiments with the *Curvature* stream-operator pipeline: (1) Having the test machine configured with 256MB, and (2) with 2GB of main memory. In (1), the Lucy, David 1mm and St. Matthew (output) data sets significantly exceeded the available physical memory, but in (2) only St. Matthew did.

As strongly supported by the chart in Figure 14, the experiments reveal that our stream-processing framework is virtually independent of the available main memory size (as long as it can hold the very limited active working set $\mathcal{A}$). The size of main-memory is essentially irrelevant and has no effect on the overall point processing cost, because all the expensive computational work is limited to the small set of points in the active working set $\mathcal{A}$ which can easily be kept in main memory for huge data sets. Therefore, our stream-processing framework can handle exceedingly large data sets from out-of-core which is equally nicely demonstrated by our experiments.

Moreover, as the streaming concept only relies on an ordered sequential access, the input and output streams can also be much larger than 32-bit virtual address space as demonstrated for the St. Matthew model (e.g. see its *Curvature* output size in Table 2).

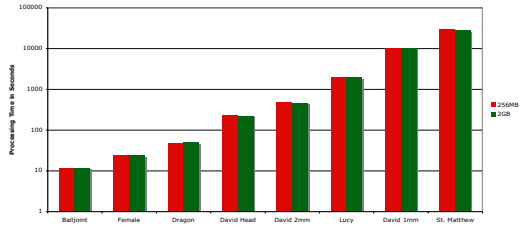

Figure 14. Dependency, or rather in-dependency, of available main memory on total stream-processing cost for various models.

7.2.4 Performance

While the current implementation is not specifically optimized for performance, the experiments show that the major processing cost is the determination of all k-nearest neighbors as shown in Figure 15 for the *Curvature* stream-processing pipeline. The extra large k-nearest search cost for the David 1mm model stems from the fact that for this model the stream operator Φ_X buffers noticeably more elements during the first 6M stream-processed points (see corresponding chart in Figure 13).

As mentioned in Section 6, the average size m of the k-nearest neighbor buffer is the most important performance factor as it contributes to an expected $n \cdot O(k \log m)$ k-nearest neighbor search cost factor for a point stream of size n. The other operators only add constant cost factors as they operate on the fixed k-nearest neighbors. Furthermore, the I/O overhead to read/write the point streams from/to disk does not comprise a major bottleneck of the stream-processing framework and hence the concept is well suited for processing very large data sets (see also Section 7.2.3).

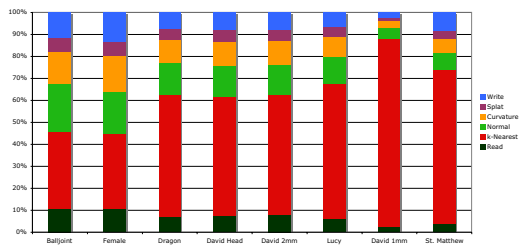

Figure 15. Percentage of time costs of the different stream-operator processing stages.

7.3 Versatility

To demonstrate the practical application of our stream-processing points framework we performed normal, splat-ellipse and curvature estimation, with results shown in Figure 16. The normal and splat estimation operators generate accurate point attributes that can be exploited in high-quality point-based rendering systems. Additionally, the curvature operator provides a robust estimate of the main curvature directions and their qualitative strengths which may be used as the basis for more complex operations such as feature extraction or surface segmentation.

Figure 16. Results of applying normal computation and splat estimation; and curvature stream operators to raw point cloud data sets. The left column shows the high-quality normal estimation and on the right the qualitative (RMS) curvature strength is color coded.

To further demonstrate the versatility of our modular stream-operator framework we also performed some experiments with the proposed fairing operator described in Section 4.3.4. For this purpose we introduced random normal-distributed noise in the magnitude of 0.05% of the bounding-box diagonal to the David head model in Figure 17, and used the noisy Lion model in Figure 18. In both cases we set the variance of the Gaussian weight functions $f(r)$ and $g(r)$ in Equation 3 to 0.5% of the bounding-box diagonal. As demonstrated the results manifest excellent feature-preserving smoothing effects, and substantiate the flexibility of our stream-processing points approach to accommodate a wide range and complexity of different local operators.

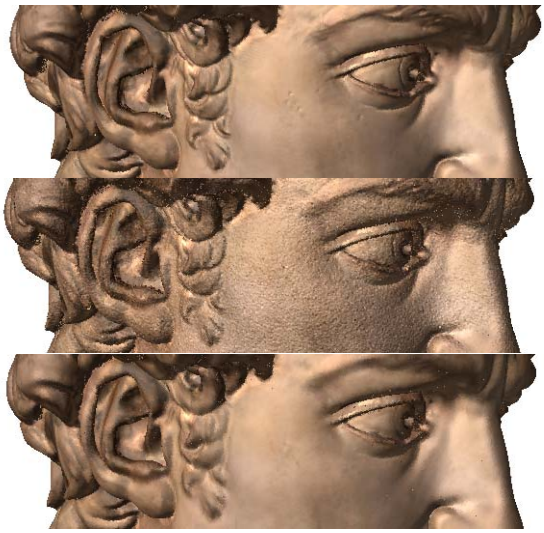

Figure 17. Original smooth surface (top); random noise of 0.05% of diagonal length added to each coordinate (middle); and smoothened model using our stream-process fairing operator (bottom).

Figure 18. Original noisy input model (top); and smoothened model using our stream-process fairing operator (bottom).

8. Discussions

We have presented a novel *point processing* framework based on a linear *streaming* of points, a sweep-plane algorithm for k-nearest neighborhood determination and the definition of concatenable local *stream operators*. To our knowledge this is the first method that can apply local operators such as normal estimation and fairing without a data structure holding the entire data set in in-core or virtual memory, and that is applicable to arbitrary large data sets out-of-core with only limited main memory usage. It is also the only approach processing points as streams and that is extensible in a modular way to apply multiple concatenated local operators consecutively on the point set.

Several performance details are not optimized in the current framework. Among the possible improvements is a much more aggressive balancing strategy to keep the k-nearest neighbor query cost low. Further work includes the development of a specialized sweep-plane spatial search structure for this purpose.

The k-nearest neighborhood sweep-plane algorithm described in Section 4.2.2 can under certain circumstances generate an approximate k-nearest neighbor set instead of the exact solution. However, in practice we observed no difference to the exact solution with several test models. Moreover, a good approximate k-nearest neighbor set may be sufficient for most local operators. Additionally, the framework can easily be modified to compute a fixed-range d neighborhood with variable k for each point, and then an exact distance-d k-nearest neighbor set can be computed.

The major limitations include that extreme spatial outliers of disjoint point clusters with less than k elements may cause the active working set to grow unproportionally. Also significant manipulation of point coordinates in stream operators (i.e. beyond local smoothing) may cause the established stream-order and k-nearest neighbor sets to become intolerably incorrect. These problems may be addressed by new sort-update and k-nearest-update stream operators that are inserted after such coordinate-manipulating operations.

Furthermore, future work will include the development of a wide variety of basic and also more complex point stream operators such as segmentation, simplification or compression.

Acknowledgements

We would like to thank the Stanford *3D Scanning Repository* and *Digital Michelangelo* projects as well as *Cyberware* for freely providing geometric models to the research community. This project was partly supported by a UCI SIIG-2003-2004-19 grant and a Ted & Janice Smith Faculty Seed Funding Award.

References

[ABCO+01] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shackar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *Proceedings IEEE Visualization 2001*, pages 21–28. Computer Society Press, 2001.

[AGPS04] Mattias Andersson, Joachim Giesen, Mark Pauly, and Bettina Speckmann. Bounds on the k-neighborhood for locally uniformly sampled surfaces. In *Proceedings Symposium on Point-Based Graphics*, pages 167–171. Eurographics, 2004.

[BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *Proceedings Pacific Graphics 2003*, pages 335–343. IEEE, Computer Society Press, 2003.

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings Eurographics Workshop on Rendering*, pages 53–64, 2002.

[CDR00] Ulrich Clarenz, Udo Diewald, and Martin Rumpf. Anisotropic geometric diffusion in surface processing. In *Proceedings IEEE Visualization 2000*, pages 397–405. Computer Society Press, 2000.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[DGH01] Tamal K. Dey, Joachim Giesen, and James Hudson. A delaunay based shape reconstruction from larga data. In *Proceedings IEEE Symposium in Parallel and Large Data Visualization and Graphics*, pages 19–27, 2001.

[DH02] Tamal K. Dey and James Hudson. PMR: Point to mesh rendering, a feature-based approach. In *Proceedings IEEE Visualization 2002*, pages 155–162. Computer Society Press, 2002.

[DMSB99] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings ACM SIGGRAPH 99*, pages 317–324. ACM Press, 1999.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *Proceedings ACM SIGGRAPH 03*, pages 657–662. ACM Press, 2003.

[GD98] J.P. Grossman and William J. Dally. Point sample rendering. In *Proceedings Eurographics Rendering Workshop 98*, pages 181–192. Eurographics, 1998.

[GKS00] M. Gopi, S. Krishnan, and C.T. Silva. Surface reconstruction based on lower dimensional localized delaunay triangulation. In *Proceedings EUROGRAPHICS 00*, pages 467–478, 2000.

[HDD+92] H. Hoppe, T. DeRose, T. Duchampt, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *Proceedings ACM SIGGRAPH 92*, pages 71–78. ACM Press, 1992.

[IG03] Martin Isenburg and Stephan Gumhold. Compression for gigantic polygon meshes. In *Proceedings ACM SIGGRAPH 2003*, pages 935–942. ACM Press, 2003.

[ILGS03] Martin Isenburg, Peter Lindstrom, Stephan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In *Proceedings IEEE Visualization 2003*, pages 465–472. Computer Society Press, 2003.

[JDD03] Thouis R. Jones, Fredo Durand, and Mathieu Desbrun. Non-iterative, feature-preserving mesh smoothing. In *Proceedings ACM SIGGRAPH 2003*, pages 943–949. ACM Press, 2003.

[JDZ04] Thouis R. Jones, Fredo Durand, and Matthias Zwicker. Normal improvement for point rendering. *IEEE Computer Graphics and Applications*, 24(4):53–56, July-August 2004.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, 3rd Edition*. Addison-Wesley, 1998.

[KV03] Aravind Kalaiah and Amitabh Varshney. Modeling and rendering points with local geometry. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):30–42, January-March 2003.

[Lin96] John P. Linderman. rsort and fixcut. man pages, 1996. revised June 2000.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital Michelangelo project: 3D scanning of large statues. In *Proceedings SIGGRAPH 2000*, pages 131–144. ACM SIGGRAPH, 2000.

[LW85] Marc Levoy and Turner Whitted. The use of points as display primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill, 1985.

[LWZL02] G. H. Liu, Y. S. Wong, Y. F. Zhang, and H. T. Loh. Adaptive fairing of digitized point data with discrete curvature. *Computer Aided Design*, 32(4):309–320, 2002.

[Mee01] Gopi Meenakshisundaram. *Theory and Practice of Sampling and Reconstruction for Manifolds with Boundaries*. PhD thesis, Department of Computer Science, University of North Carolina Chapel Hill, 2001.

[MN03] Niloy J. Mitra and An Nguyen. Estimating surface normals in noisy point cloud data. In *Symposium on Computational Geometry*, pages 322–328. ACM, 2003.

[Paj03] Renato Pajarola. Efficient level-of-details for point based rendering. In *Proceedings IASTED Invernational Conference on Computer Graphics and Imaging (CGIM 2003)*, 2003.

[PG01] Mark Pauly and Markus Gross. Spectral processing of point-sampled geometry. In *Proceedings ACM SIGGRAPH 2001*, pages 379–386. ACM Press, 2001.

[PGK02] Mark Pauly, Markus Gross, and Leif P. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings IEEE Visualization 2002*, pages 163–170. Computer Society Press, 2002.

[PKKG03] Mark Pauly, Richard Keiser, Leif Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. In *Proceedings ACM SIGGRAPH 2003*, pages 641–650. ACM Press, 2003.

[PSG04] Renato Pajarola, Miguel Sainz, and Patrick Guidotti. Confetti: Object-space point blending and splatting. *IEEE Transactions on Visualization and Computer Graphics*, 10(5):598–608, September-October 2004.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings SIGGRAPH 2000*, pages 335–342. ACM SIGGRAPH, 2000.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings SIGGRAPH 2000*, pages 343–352. ACM SIGGRAPH, 2000.

[RPZ02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings EUROGRAPHICS 2002*, pages 461–470, 2002. also in Computer Graphics Forum 21(3).

[SK01] Robert Schneider and Leif Kobbelt. Geometric fairing of irregular meshes for free-form surface design. *Computer Aided Geometric Design*, 18(4):359–379, 2001.

[SP04] Miguel Sainz and Renato Pajarola. Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879, 2004.

[SPL04] Miguel Sainz, Renato Pajarola, and Roberto Lario. Points reloaded: Point-based rendering revisited. In *Proceedings Symposium on Point-Based Graphics*, pages 121–128. Eurographics Association, 2004.

[Tau95] Gabriel Taubin. A signal processing approach to fair surface design. In *Proceedings SIGGRAPH 95*, pages 351–358. ACM SIGGRAPH, 1995.

[Vit01] Jeffrey S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[WPK+04] T. Weyrich, M. Pauly, R. Keiser, S. Heinzle, S. Scandella, and M. Gross. Post-processing of scanned 3D surface data. In *Proceedings Symposium on Point-Based Graphics*, pages 85–94. Eurographics, 2004.