In *Proceedings IEEE Visualization*, pages –, 2005.

Stream-Processing Points

Renato Pajarola[1]
Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

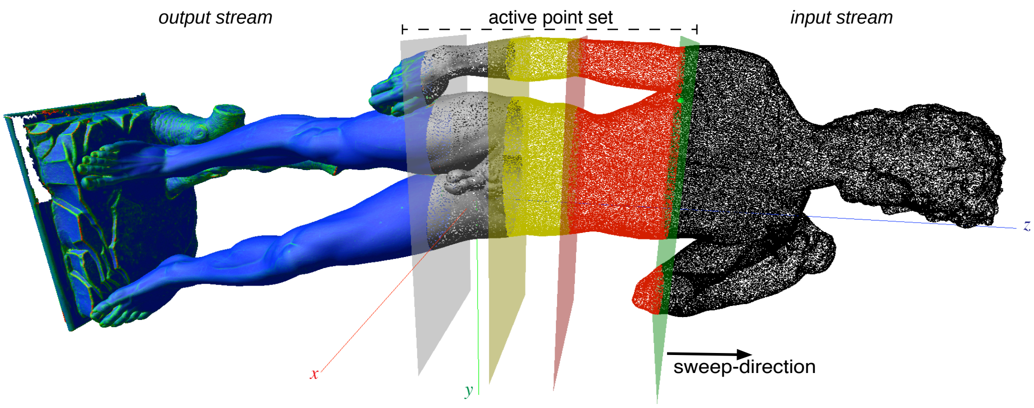

Figure 1. Simulated stream-processing stages with (r.t.l.): points to be read from input stream (black points), in nearest neighborhood evaluation (red points), during normal computation (yellow points), amid curvature estimation (shaded grey points) and fully processed and written to output stream (shaded color-coded splats). Note that the extent of the *active point set* is greatly exagerated in this illustration compared to the real data (see Figure 4).

ABSTRACT

With the growing size of captured 3D models it has become increasingly important to provide basic efficient processing methods for large unorganized 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 new concept are: (1) support of extensible and concatenable local operators called* ***stream operators****, (2) low main-memory usage and (3) applicability to process very large data sets out-of-core.*

CR Categories: I.3 Computer Graphics, I.3.5 Comp. Geometry and Object Modeling, I.3.6 Methodology and Techniques

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

1. Introduction

In any visualization context, ahead of any display the input data must be cleaned, filtered, modeled, or in short *processed*, before it can be rendered and manipulated. This processing, and not rendering itself, of large point sets is the main focus of this paper.

Point samples are the natural raw output data primitives of the geometry capturing stage in most 3D acquisition systems. In fact, points are the fundamental geometry-defining entities. Satisfying provably correct surface sampling criteria, a set of points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ fully defines the geometry as well as the topology of a surface. Here we assume that the input surface data is sufficiently densely sampled.

With the increasing use and precision of 3D acquisition systems it is critical to support raw point cloud data in a practical way in an acquisition and visualization context. The data processing and modeling stages of a visualization system, in particular, must support basic point processing operations such as surface normal estimation or fairing. These operators can be computed efficiently if the point data can be loaded into a main memory spatial indexing structure. However, while optimal up to some limit, this is main-memory inefficient and dramatically decreases in performance when the model exceeds available physical main memory. In the case of significant mismatch between model and physical main memory size it may nearly come to a halt due to memory thrashing [8]. Moreover, combining multiple operations can generally not be done by merely concatenating operators.

In this paper we introduce and set the stage for a new *stream-processing* concept for processing points sequentially to improve memory access coherency and dramatically limit main memory cost. This sequential stream-processing allows us to process large models *out-of-core*, and is insensitive to available main memory. Supported operations include 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 and curvature estimation as well as filter operations such as fairing on point data sets follow this principle. Indeed, surface parameter estimation and filter operations are among the most important tasks for (pre-)processing raw points. Our *stream-processing* concept supports non-recursive local operators $\Phi(\boldsymbol{p})$ that include nearby sample points within a well defined (spatially) local neighborhood.

2. Related Work

After some early work [22, 13], many point sample display techniques have recently been proposed [33, 31, 32, 4, 20, 3, 27, 35, 34]. An interesting way of treating points sequentially is presented in [6]. In general most techniques address higher-level point processing tasks such as multiresolution rendering, given all point attributes. Lower-level point processing techniques as in [28, 24, 30, 19, 39], however, are aimed at processing moderate point set sizes in main memory and assume that some basic attributes such as normal estimates have already been computed.

Estimation of vertex attributes such as normal orientation is a common data processing task in polygonal surface reconstruction methods [14, 12, 25], as is fairing in surface modeling [37, 9, 5, 36, 18]. However, generally these approaches are not aimed at processing models consisting of tens of millions of vertices or more and do not scale well to out-of-core processing.

Work on processing triangle meshes sequentially can be found in [15, 17]. These techniques sequentially grow mesh regions in a coherent way to limit main memory usage. However, no low-level operators are supported, and more importantly, the techniques do not extend to raw point data processing. In [16] a streaming format

[1]pajarola@acm.org

for rendering indexed meshes is proposed from which the *spectral sequencing* could be applicable to stream-ordering raw points in our context, and in [40] a streaming mesh decimation is presented. None of the related work, however, provides streaming low-level operations and filter functions.

Stream-based data handling is common in processing audio and video data which in contrast to 3D geometry is inherently sequential in time. In the context of geometry processing, however, sweep-line techniques in computational geometry [7] are more closely related. Our basic stream-processing concept follows this idea of sweeping a plane through 3D space and considering events when data elements are passed by the sweep-plane.

3. Streaming Concept

The fundamental idea behind streaming is to process data sequentially with only a limited amount of data active at any time, resembling a sliding window over the 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.

Figures 1 and 2 illustrate the 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 $\mathcal{A}$. Conceptually we move a *sweep-plane* through space along an axis of spatial ordering.[1] When a new point $\boldsymbol{p}_j$ is passed, denoting an event in classical line-sweep algorithms [7], it is added to the working set $\mathcal{A}$. The active set $\mathcal{A} = \{\boldsymbol{p}_{j-m}, \ldots, \boldsymbol{p}_j\}$ is 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 $\boldsymbol{p}_{i>j-m}$ 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 $\boldsymbol{p}_i \notin \mathcal{A}$) not yet read from the input stream, or already written to the output stream, can reside out-of-core (e.g. in a virtual-memory mapped file). Only points *living* in the working set $\mathcal{A}$ reside in main memory, and any of its temporary extra data such as neighborhood information and other attributes.

Since the active set $\mathcal{A}$ is orders-of-magnitude smaller than 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 [21,38], and in practice the *rsort* [23] implementation has been used for similar tasks.

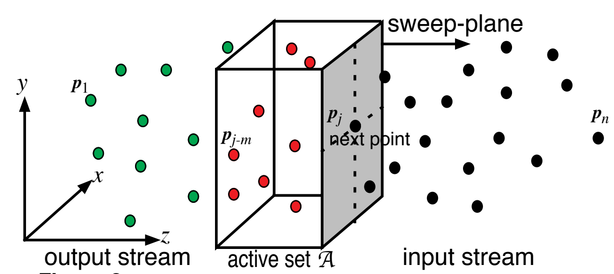

Figure 2. Sweep-plane process overview: unprocessed points are read sequentially from the input stream, processed points are written to the output stream.

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

4. Stream Operators

4.1 Definitions

The class of functions supported by our stream-processing concept includes operations performing a computation on a point which only require 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 as the k-nearest neighbors, or points within a given distance d. The parameters k or d will usually be given by the user or application but could as well be derived for each point as suggested in [25, 2]. 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 in 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 for example illustrated in Figure 3. In this context, each 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 points $\boldsymbol{p}_j \in N_i$. Additionally, the FIFO semantic of its queue Q_k ensures no interference between consecutive operators $\Phi_{k\pm1}$.

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 a compatible order of stream operators and attributes must be selected.

Moreover, each stream operator Φ_k must assure that a point $\boldsymbol{p}_i$ is passed to the next operator Φ_{k+1} only if $\boldsymbol{p}_i$ is fully processed and all affected attributes are updated by Φ_k. This is facilitated by the FIFO queue constraint on 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 preceeding or succeeding operators $\Phi_{k\pm1}$. This overlap of neighborhood sets N_i between consecutive stream operators is indicated in our figures (e.g. in Figure 3) 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 addressed in Section 5.2.

4.2 Fundamental Stream Operators

4.2.1 I/O Operators

The first and last stream operators in a stream-processing pipeline do the I/O from/to input/output streams. As depicted in Figure 3 the *read operator* $\Phi_R(\boldsymbol{p}_j)$ only reads and buffers one 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.

Note that any stream-processing stage following $\Phi(\boldsymbol{p}_i)$ must make sure that no elements of N_i are altered until $\Phi(\boldsymbol{p}_i)$ has completed. In particular, point $\boldsymbol{p}_{j-m}$ scheduled to leave the active set $\mathcal{A}$ must be handled with care. Hence we introduce the *deferred-write operator* Φ_W (last in sequence of stream operators). This operator, as illustrated in Figure 3, assures that any point $\boldsymbol{p}_{j-m}$ is removed from $\mathcal{A}$ and written to the output stream if and only 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 $\Phi_{k-1..1}$. The deferred-write operator is implemented by a simple FIFO queue. 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

The neighborhood $N_i = \{\boldsymbol{p}_{i_1}, \ldots, \boldsymbol{p}_{i_k}\}$ of a point $\boldsymbol{p}_i$ can be defined in a number of ways. We outline the most important k-nearest neighbor (kNN) set here but others could also be supported (e.g. see [25, 2]). The computation of N_i is a special *neighborhood operator* $\Phi_X(\boldsymbol{p}_i)$ in our stream-processing framework and will generally be the second stream operator after Φ_R as in Figure 3.

We must determine the kNN set N_j of a point $\boldsymbol{p}_j$ passed by the sweep-front just after insertion into the active point set $\mathcal{A}$. To compute all kNNs efficiently, or any neighborhood for that matter, it is essential to use a spatial index S over the relevant point set for fast spatial (range-) queries. However, since we are processing a point stream and want the index to be as small as possible, we must remove elements from this index at the earliest possible time. Hence the index S must also incorporate a priority-queue over the stream indices i of points $\boldsymbol{p}_i \in S$. For efficiency reasons we use a kD-heap, a dynamic semi-balanced kD-tree with integrated priority heap, as spatial index S. In fact, since points are streamed in one dimension we use a 2D kD-tree partitioning the sweep-plane. That is sensible because the streaming dimension of set $\mathcal{A}$ has generally a very small extent compared to the other two dimensions.

Two basic operations are supported: incremental insertion of a new element into the kD-heap, and removal of an arbitrary element while satisfying the kD-tree and priority-heap structure [7].

Our streaming kNN approach is summarized as follows (see also Figure 3): At insertion of $\boldsymbol{p}_j$ into $\mathcal{A}$ a *left-sided* kNN set N_j is initialized, a query on S finding the kNN set N_j – with smaller indices $i<j$ since S only contains prior points in the sequential ordering. Additionally, during the insertion of $\boldsymbol{p}_j$ into the spatial index S we also update the *right-sided* kNN sets N_i of points $\boldsymbol{p}_{i<j}$ already in S, with respect to the new point $\boldsymbol{p}_j$.

Finally, as it is imperative to keep the size of S as small as possible we remove points with completed kNN sets as early as possible. Thus our kD-heap is queried to find the list L of points $\boldsymbol{p}_i$ in S for which the sweep-plane has moved beyond the farthest kth-nearest neighbor in N_i. The set L is then removed from S and passed to a sorting buffer B as depicted in Figure 3 which re-establishes the global stream ordering. The smallest element $\boldsymbol{p}_i$ of B is correctly stream-ordered if its index i is smaller than the smallest index in S.

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 of which we outline a small set of meaningful operators that are currently implemented. This extensible list of important operators shows the power and applicability of the proposed stream-processing concept.

4.3.1 Normal Estimation

To demonstrate a regular simple local operator we first introduce normal estimation $\Phi_N(\boldsymbol{p}_i)$ as a variation of plane fitting (see also [1, 29, 25, 30]). A normal estimation stream operator $\Phi_N(\boldsymbol{p}_i)$, together with the read, kNN and deferred-write fundamental operators, constitutes one of the most basic stream-processing pipeline configurations that performs a meaningful operation on a raw point set.

A *local least squares* (LLS) plane fit to a point $\boldsymbol{p}_i$ and its kNN set $N_i = \{\boldsymbol{p}_{i_1}, \ldots, \boldsymbol{p}_{i_k}\}$ is defined by the eigenvalue analysis and eigenvector decomposition of the covariance matrix M_i over $\boldsymbol{p}_i$ and N_i. We express a *moving least squares* (MLS) representation of the covariance as weighted sum [1]:

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

The weight $\theta(r)$ is a Gaussian function $\theta(r) = e^{-r^2/2\sigma^2}$, 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 [25]. Thus the normal $\boldsymbol{n}_i$ of a point $\boldsymbol{p}_i$ is computed as eigenvector M_i corresponding to the smallest eigenvalue of M_i (from singular value decomposition (SVD) of symmetric positive semidefinite matrices).

4.3.2 Curvature Estimation

Another simple operator is curvature estimation $\Phi_C(\boldsymbol{p}_i)$, which we implement based on the covariance of normals $\boldsymbol{n}_j$ of points $\boldsymbol{p}_j \in N_i$. Similar to Equation 1, we define a MLS of the normal covariance as:

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

The SVD of the covariance of normals of Equation 2 gives us an estimate of the curvatures and its principal directions. Figures 1 and 7 illustrate the principal curvatures (root mean square (RMS), mean or absolute curvature).

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 [35, 34]. The elliptical extent of a point $\boldsymbol{p}_i$ could be derived from locally computed Voronoi cells as in [10, 11]. However, given the local neighborhood N_i, a covariance analysis [29, 26, 27] is more suitable for implementation as an elliptical splat-estimation stream operator $\Phi_E(\boldsymbol{p}_i)$.

We can determine the ellipse major and minor axis directions, major axis length and aspect ratio for a point $\boldsymbol{p}_i$ efficiently from the analysis of the covariance matrix M_i given in Equation 1. The eigenvectors of 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.

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. This yields slightly different elliptical splats oriented along ridges and valleys.

4.3.4 Fairing

To demonstrate the potential power and extensibility of the proposed stream-processing framework we introduce a smoothing operator Φ_S. To filter noise artifacts many smoothing algorithms have been proposed for meshes (e.g. [37], [9], [5] or [36]). In [28], fairing of points has been proposed which requires a regular (re-)sampling pattern. Not unlike [19] we adopt the non-iterative feature preserving fairing operator presented in [18]. Its applicability to triangle soups makes it suitable for point sets as well.

Given a point $\boldsymbol{p}_i$ and its neighbors N_i, we directly extend the smoothing operation of [18] to points as follows

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

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 splat size). The term w_i is $\sum a_j \cdot f(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \cdot g(|\Pi_j(\boldsymbol{p}_i) - \boldsymbol{p}_i|)$, the sum of weights. The Gaussian weight function $f(r)$ adjusts the influence based on spatial distance, while $g(r)$ preserves sharp features by giving less weight to points with different normal orientations [18].

Note, however, that the fairing operator $\Phi_S(\boldsymbol{p}_i)$ must fit into a properly configured stream-processing pipeline as illustrated in Figure 3. In particular, applying the fairing operator $\Phi_S(\boldsymbol{p}_i)$ calls for recomputation of new normals $\boldsymbol{n}_i$, as well as (elliptical) splat parameters. Hence we apply normal and splat size estimation Φ_N and Φ_E also after the fairing operator Φ_S as shown in Figure 3.

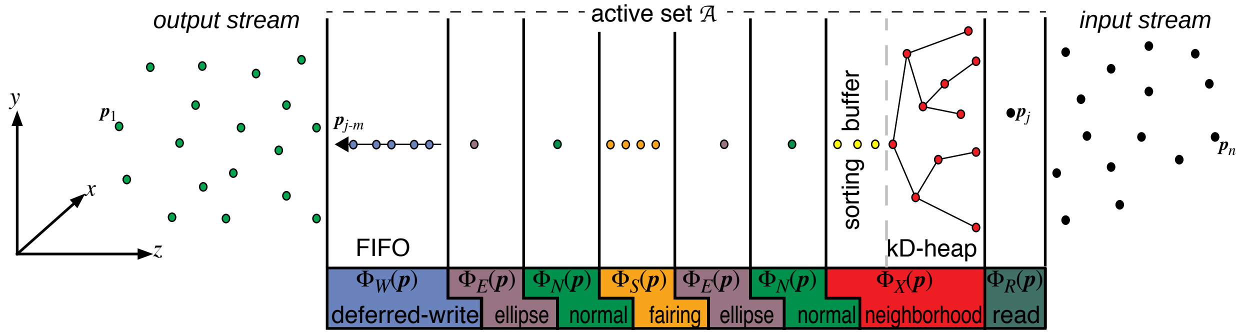

Figure 3. Stages of a complex stream-processing pipeline for fairing, with fundamental stream operators for reading $\Phi_R(\boldsymbol{p})$, writing $\Phi_W(\boldsymbol{p})$ and all k-nearest neighbors $\Phi_X(\boldsymbol{p})$. The smoothing operator $\Phi_S(\boldsymbol{p})$ is enclosed by a pair of normal and splat size operators $\Phi_N(\boldsymbol{p})$ and $\Phi_E(\boldsymbol{p})$. A point $\boldsymbol{p}_i$ moves from right-to-left through the staged stream operators $\Phi_j(\boldsymbol{p}_i)$.

5. Implementation

A major challenge is the systematic definition and development of stream operators. In particular, this includes:

1. defining an 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.

5.1 Attribute Handling

Different stream operators $\Phi_k(\boldsymbol{p}_i)$ add or modify different subsets of point attributes $a_i^k \subseteq A_i$ which may be in addition to the input data. Moreover, attributes may only be needed temporarily and not in the output. Therefore, we define the stream-point data type as an extensible set of attribute-fields (see also Appendix Figure 10):

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 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 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 varying configurations of stream operators in a stream-processing pipeline. 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 kNN operator $\Phi_X(\boldsymbol{p}_i)$ computes all auxiliary fields with respect to a point $\boldsymbol{p}_i$'s neighborhood information N_i. This also includes the min and max of referenced indices j of the points $\boldsymbol{p}_j \in N_i$ which's use is further detailed below. The normal operator $\Phi_N(\boldsymbol{p}_i)$ computes the normal $\boldsymbol{n}_i$, which is usually part of the output a_i^{out}, based on covariance information stored as part of a_i^{aux}. The splat estimator Φ_E is based on existing normal and covariance information and outputs ellipse parameters as part of a_i^{out}. For its calculation, the fairing operator $\Phi_S(\boldsymbol{p}_i)$ uses some temporary attributes a_i^{aux} but adds no output fields. (See also Appendix Figure 10.)

5.2 Stream Operator Classes

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 released to the subsequent operator Φ_{k+1} and its queue Q_{k+1} depend on the type of the stream operator Φ_k.

The semantic of the buffer Q_k of a stream operator Φ_k is equivalent to a FIFO queue (interface given in Appendix Figure 11) which includes the *front*() and *pop_front*() methods. However, instead of a *push_back*() interface we define the exchange of points between operators as a *pull-push* mechanism, see also Section 5.3. For this each operator Φ_k keeps a reference to its previous operator Φ_{k-1} in the operator pipeline. Other stream-operator functions include 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 $\boldsymbol{p}_j \in \cup_i N_i$ – of any *unprocessed* points $\boldsymbol{p}_i$ in Q_k.

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 kNN 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 Φ_{k-1} it can be processed by Φ_k and immediately released to Φ_{k+1}. In practice its FIFO queue Q_k will generally be empty as the subsequent operator Φ_{k+1} consumes any released points immediately.

The standard FIFO queue *front*() and *pop_front*() methods are straightforward implementations for a through-buffer stream operator Φ_k (given in Appendix Figure 13). The *pull_push()* method (given in Appendix Figure 12) basically grabs points from the prior operator Φ_{k-1}, processes and then releases them to the next operator Φ_{k+1}.

Normal computation as well as elliptical splat-estimation stream operators (Sections 4.3.1 and 4.3.3) belong to this 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:

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

Hence the *pull_push()* method (given in Appendix Figure 14) must *pre-* as well as *post-buffer* the processed points $\boldsymbol{p}_i$. Two queues are 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 grabs all points $\boldsymbol{p}_i$ released from the preceding operator Φ_{k-1} and queues them in FIFO1. Next, the queue FIFO1 is checked for available points $\boldsymbol{p}_i$ that can now safely be processed by Φ_k and queued in FIFO2. This requires testing for the smallest unprocessed and smallest referenced indices in the previous operators Φ_{k-1}.

The *pop_front*() interface (given in Appendix Figure 15) pops points exclusively from FIFO2, the post-buffer, as only this queue maintains points already processed by Φ_k Note that 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.

This category of stream operators $\Phi_k(\boldsymbol{p}_i)$ must carefully keep track of the smallest index i of any point $\boldsymbol{p}_i \in Q_k$, and the smallest referenced neighbor index j of any $\boldsymbol{p}_j \in \bigcup N_i$ of any unprocessed point $\boldsymbol{p}_i$ in Q_k. This is achieved by maintaining a heap structure of indices for this purpose (see also Appendix Figure 14 and Appendix Figure 15).

The *k*NN and the fairing operators (Sections 4.2.2 and 4.3.4) are pre- and post-buffer stream operators. The fairing operator $\Phi_S(\boldsymbol{p}_i)$ changes the coordinates of a point $\boldsymbol{p}_i$ and must avoid that any stream operators $\Phi_{S\pm1}(\boldsymbol{p}_j)$ act on a mix of pre- and post-faired points $\boldsymbol{p}_i \in N_j$. The *k*NN 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 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 queuing points with completed *k*NN sets and removed from the kD-heap.

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_{C-1}. That is because $\Phi_C(\boldsymbol{p}_i)$ depends on the normals $\boldsymbol{n}_j$ of all *k*-nearest points $\boldsymbol{p}_j \in N_i$ which may still have to be computed in a prior normal operator.

5.3 Stream-Processing Pipeline

Setting up a stream-processing point pipeline is very simple given the outlined stream-operator framework. Some user-involvement is required to select a proper sequence of stream operators and matching attribute fields.

After setting up the input fields and initializing the stream operators the input and output point-streams can be set to memory-mapped file arrays of InputFields and OutputFields types. The main processing stage then merely consists of two very simple nested loops as shown below: 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.

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

(Appendix Figure 16 gives the complete main routine corresponding to pipeline in Figure 3.)

6. Analysis

In terms of memory requirements we note that the most critical part 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 index structure requires O(n) space and allows processing all points and *k*NNs 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 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. Hence the expected main memory usage is only in the order of O(m), as only a *sliding window* of m elements is continuously held in the active set $\mathcal{A}$. Moreover, as the processing performance is mainly determined by the *k*NN query, 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*NNs 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 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 wall-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 point data sets are given in Table 1. All data sets are ordered for streaming along the dimension of largest extent. Besides St. Matthew, which was converted from a binary QSplat model [33], 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 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 [21,38] such as the *rsort* [23] tool that has been used in similar situations, however, this is not the main focus here.

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

Model	#Points	Mesh File Size	Point Stream File Size	Preprocess reading	Preprocess 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

7.2 Stream Processing

7.2.1 Overview

In our 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:

- *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. This 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.

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

Model	Pipeline	Point Stream Output Size	Timing: Process (h:mm:ss)	Timing: 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

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 4 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 maintain 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.

7.2.3 Main Memory (In-)Dependence

To back the claim of effective stream-processing of large point sets we carried out two 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 5, 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 that experiments.

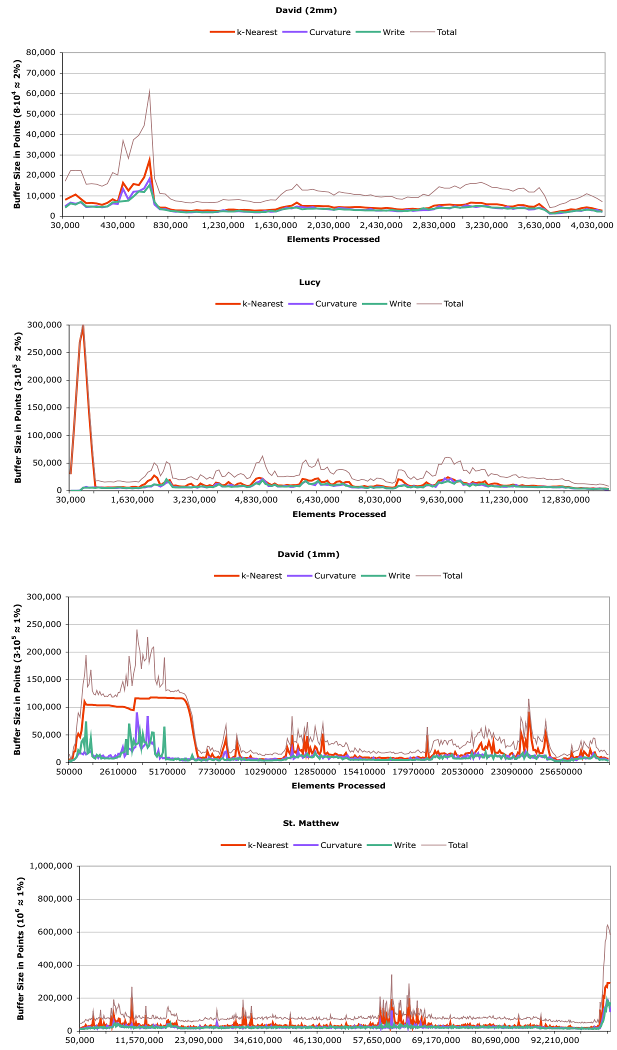

Figure 4. Streaming total active working set and buffer sizes of corresponding stream operators plotted against the progress through the input point stream. (y-axis indicates size only up to 1% or 2% of the entire data set)

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).

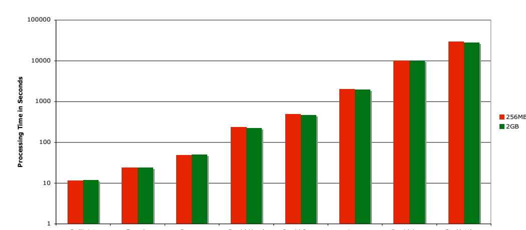

Figure 5. Dependency, or rather independency, of available main memory on total stream-processing cost for various models.

7.2.4 Performance

While the current implementation is not optimized for performance, the experiments show that the major cost is the determination of all kNN as shown in Figure 6 for the *Curvature* stream-processing pipeline. The extra large kNN 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 chart in Figure 4).

As mentioned in Section 6, the average size m of the kNN buffer is the main performance factor as it contributes to an expected $O(k \log m)$ kNN search cost for each point. The other operators only add constant cost factors as they operate on the fixed kNN set. Moreover, disk read/write I/O overhead does not comprise any bottleneck of the proposed stream-processing framework and hence the concept is well suited for processing very large data sets (see also Section 7.2.3).

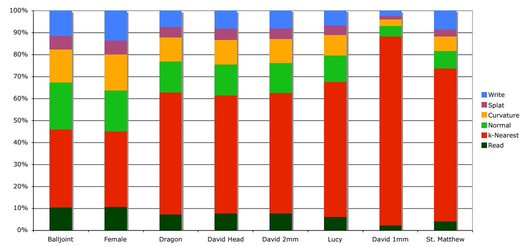

Figure 6. Percentage of time costs of the different stream-operator processing stages.

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 7. The normal and splat estimation operators generate accurate point attributes that can be exploited in high-quality point-based visualization 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 7. Results of applying normal computation, splat estimation and curvature stream operators to raw point cloud data sets. The images show the high-quality normal estimation and the color coded qualitative (RMS) curvature strength.

To further demonstrate the versatility of our modular stream-operator framework we also performed initial 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 8, and used the noisy Lion model in Figure 9. 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.

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

Figure 8. 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 9. Original noisy input model (top); and smoothened model using our stream-process fairing operator (bottom).

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.

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. In particular, a multires-operator to generate a multiresolution output format for efficient level-of-detail visualization is of immediate interest.

ACKNOWLEDGEMENTS

We like to thank the Stanford *3D Scanning Repository*, *Digital Michelangelo* project and *Cyberware* for providing models. This project was partly supported by a UCI SIIG-2003-2004-19 grant and a Ted & Janice Smith Faculty Seed Funding Award. We also thank the reviewers for their helpful comments which, if not in this paper, will be certainly be addressed in follow up work.

REFERENCES

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

[2] 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.

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

[4] 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.

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

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

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

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

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

[10] 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.

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

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

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

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

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

[16] Martin Isenburg and Peter Lindstrom. Streaming meshes. Technical Report UCRL-CONF-201992, Lawrence Livermore National Laboratory, 2004.

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

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

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

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

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

[27] 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.

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

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

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

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

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

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

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

[35] 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.

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

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

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

[39] 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.

[40] Jianhua Wu and Leif Kobbelt. A stream algorithm for the decimation of massive meshes. In *Proceedings Graphics Interface*, pages 185–192, 2003.

Stream-Processing Points

Renato Pajarola[1]
Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

A. Code Appendix

```
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 10. Attribute-field structures of stream-points for a normal computation, elliptical splat estimation and fairing stream-processing pipeline as illustrated in Figure 3.

```
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 11. Abstract common interface definition of the virtual stream operator base-class.

```
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 12. Class definition and pull-push method of a through-buffer type stream operator.

```
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 13. FIFO queue access and index-reference methods for through-buffer type stream operators.

1. pajarola@acm.org

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

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

```
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 16. Outline of main point stream-processing routine for a normal computation, elliptical splat estimation and fairing stream-processing pipeline as illustrated in Figure 3.