Stream-Processing Points

Pajarola, R
Stream-Processing Points

Renato Pajarola

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

pajarola@acm.org

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 exaggerated 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 concatenateable 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

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 \( p_1, \ldots, p_n \in \mathbb{R}^3 \) 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 \( \Phi(p) \), called stream operators, that perform a function on a point \( p \) 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 \( \Phi(p) \) 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
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 \( \Phi(p) \) performs a function on a point \( p \) that computes or updates a subset of attributes \( A_i \) associated with \( p \). As function parameters, \( \Phi(p) \) only accepts \( p, A_i \) and a set of points \( p \in N_i \) within close spatial proximity to \( p \) (and all their associated attributes \( A_i \)).

The neighborhood set \( N_i \) of points close to \( p \) 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 split size. The above definition of a local operator \( \Phi(p) \) allows it to be applied to a point \( p \in A \) for which all elements of \( N_i \) are also in the current working set, \( N_i \subseteq A \). 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 \( \Phi_1, ..., \Phi_p \) can be concatenated and applied in succession to a stream of points as for example illustrated in Figure 3. In this context, each operator \( \Phi_i \) also acts as a sequential FIFO queue buffer \( Q_i \) on the point stream and satisfies the following:

Definition 4.2 A local operator \( \Phi_i(p) \) is streamable if it is computed in one single invocation on \( p \) and not called recursively on points \( p \in N_i \). Additionally, the FIFO semantic of its queue \( Q_i \) ensures no interference between consecutive operators \( \Phi_{i+1} \).

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 \( \Phi(p) \) 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 \( \Phi_{i-1}(p) \) to work. Hence a compatible order of stream operators and attributes must be selected.

Moreover, each stream operator \( \Phi_i \) must assure that a point \( p \) is passed to the next operator \( \Phi_{i+1} \) only if \( p \) is fully processed and all affected attributes are updated by \( \Phi_i \). This is facilitated by the FIFO queue constraint on \( Q_i \) of each operator \( \Phi_i \). Note also that while \( p \in Q_i \) (the buffer of operator \( \Phi_i \)) it may be that its local neighbor points \( p \in N_i \) belong to buffers \( Q_{i-1} \) of preceding or succeeding operators \( \Phi_{i-1} \). 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.
operator stages $\Phi_{k-1}$. The deferred-write operator is implemented by a simple FIFO queue. As soon as a point $p_{i,n}$ 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 $N_j = \{p_j, \ldots, p_{j,n}\}$ of a point $p_j$ can be defined in a number of ways. We outline the most important $k$-nearest neighbor ($k$NN) set here but others could also be supported (e.g. see [25, 21]). The computation of $N_j$ is a special neighborhood operator $\Phi_N(p_j)$ in our stream-processing framework and will generally be the second stream operator after $\Phi_E$ as in Figure 3.

We must determine the $k$NN set $N_j$ of a point $p_j$ passed by the sweep-front just after insertion into the active point set $\mathcal{A}$. To compute all $k$NNs efficiently, or any neighborhood for that matter, it is essential to use a spatial index $\mathcal{S}$ 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 $\mathcal{S}$ must also incorporate a priority-queue over the stream indices $i$ of points $p_i \in \mathcal{S}$. For efficiency reasons we use a kD-heap, a dynamic semi-balanced kD-tree with integrated priority heap, as spatial index $\mathcal{S}$. In fact, since points are streamed in one dimension we use a 2D kD-tree partitioning the sweep-plane. That is sensible because the stream 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 $p_i$ into $\mathcal{A}$ a left-sided kNN set $N_i$ is initialized, a query on $\mathcal{S}$ finding the kNN set $N_j$ with smaller indices $i < j$ since $\mathcal{S}$ only contains prior points in the sequential ordering. Additionally, during the insertion of $p_i$ into the spatial index $\mathcal{S}$ we also update the right-sided kNN sets $N_i$ of points $p_{ij} \in \mathcal{S}$, with respect to the new point $p_i$.

Finally, as it is imperative to keep the size of $\mathcal{S}$ 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 $p_i \in \mathcal{S}$ for which the sweep-plane has moved beyond the farthest $k$th-nearest neighbor in $\mathcal{N}_i$. The set $L$ is then removed from $\mathcal{S}$ and passed to a sorting buffer $\mathcal{B}$ as depicted in Figure 3 which re-establishes the global stream ordering. The smallest element $p_i \in \mathcal{B}$ is correctly stream-ordered if its index $i$ is smaller than the smallest index in $\mathcal{S}$.

### 4.3 Regular Stream Operators

Given the local neighborhood $N_i$ of points $p_i$ in the active set $\mathcal{A}$, many stream operators $\Phi(p_i)$ 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 $\Phi_N(p_j)$ as a variation of plane fitting (see also [1, 29, 25, 30]). A normal estimation stream operator $\Phi_N(p_i)$, 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 $p_i$ and its kNN set $N_j = \{p_j, \ldots, p_{j,n}\}$ is defined by the eigenvalue analysis and eigenvector decomposition of the covariance matrix $M_i$ over $p_i$ and $N_j$. We express a moving least squares (MLS) representation of the covariance as weighted sum [1]:

$$M_i = \left[N\right]^{-1} \cdot \sum_{p_j \in N_j} (p_j - p_i) \cdot (p_j - p_i)^T \cdot \theta(p_j - p_i).$$

The weight $\theta(r)$ is a Gaussian function $\theta(r) = e^{-r^2/2\sigma^2}$, with variance $\sigma^2$ adaptively defined as the local point density estimate $\sigma^2 = \pi \cdot \text{MAX}_{p_j \in N_j} [D(p_j - p_i)]^2 / |N_j|$ as suggested in [25]. Thus the normal $n_i$ of a point $p_j$ 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 $\Phi_C(p)$, which we implement based on the covariance of normals $n_j$ of points $p_j \in N_j$. Similar to Equation 1, we define a MLS of the normal covariance as:

$$C_i = \left[N\right]^{-1} \cdot \sum_{p_j \in N_j} n_j \cdot n_j^T \cdot \theta(p_j - p_i).$$

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 $p_i$ 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 $\Phi_S(p_i)$.

We can determine the ellipse major and minor axis directions, major axis length and aspect ratio for a point $p_i$ 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 $n_i$ 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 $N_j$.

Alternatively, if we have a curvature operator $\Phi_C$ preceding the splat estimation $\Phi_S(p_i)$ 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 $\Phi_S$. 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 $p_i$ and its neighbors $N_i$, we directly extend the smoothing operation of [18] to points as follows:

$$p'_i = \Phi_S(p_i) = \frac{1}{w_j} \cdot \sum w_j f(p_j) \cdot g([\Pi_j(p_j) - p_i]).$$

with summation over all points $p_j \in N_i \cup N_j$. The operator $\Pi_j(p_j)$ denotes the projection of $p_j$ onto the tangent plane of point $p_i$ and the value $w_j$ corresponds to an area weight (i.e. the splat size). The term $w_j$ is $\Sigma c_j \cdot f(p_j - p_i) \cdot g([\Pi_j(p_j) - p_i])$, the sum of weights. The Gaussian weight function $g(r)$ 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 $\Phi_S(p_i)$ must fit into a properly configured stream-processing pipeline as illustrated in Figure 3. In particular, applying the fairing operator $\Phi_S(p_i)$ calls for recomputation of new normals $n_i$, as well as (elliptical) splat parameters. Hence we apply normal and splat size estimation $\Phi_N$ and $\Phi_S$ also after the fairing operator $\Phi_S$ as shown in Figure 3.
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 \( \Phi_i(p) \) can be concatenated and plugged into stream-processing systems like pipelines, and
2. concealing the dependencies between consecutively applied local stream operators \( \Phi_1, ..., \Phi_p \) effectively within the stream-operator abstract data types.

5.1 Attribute Handling

Different stream operators \( \Phi_i(p) \) add or modify different subsets of point attributes \( a^i \subseteq A_i \), 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 \( a^i \) given for each point \( p_i \) in the input stream.

**OpFields** Specifies the temporary attributes \( a^i_{\text{aux}} \) computed by stream operator \( \Phi_i(p) \) for points \( p_i \) in the active set \( A \) but not written to the output stream.

**OpOutFields** Lists the added attributes \( a^i_{\text{out}} \) computed by stream operator \( \Phi_i(p) \) for each point \( p_i \) which are passed along with the point \( p_i \) to the output stream.

**AuxiliaryFields** All auxiliary attributes \( a^i_{\text{aux}} = \bigcup a^i_{\text{aux}} \) computed and required by any stream operator \( \Phi_i(p) \) while a point \( p_i \) is in the active set \( A \) and processed by operators \( \Phi_1, ..., \Phi_p \).

**OutputFields** Includes all attributes \( a^i_{\text{out}} = \bigcup a^i_{\text{out}} \cup a^i_{\text{aux}} \) of a point \( p_i \) 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 \( p_i \).

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 \( a^i_{\text{aux}} \), the reader \( \Phi_R \) assigns an index \( j \) to each point \( p_i \) in the order it is read from the input stream. The \( k \)NN operator \( \Phi_S(p) \) computes all auxiliary fields with respect to a point \( p_i \)’s neighborhood information \( N_i \). This also includes the min and max of referenced indices \( j \) of the points \( p_j \in N_i \) which’s use is further detailed below. The normal operator \( \Phi_N(p) \) computes the normal \( n_i \), which is usually part of the output \( a^i_{\text{out}} \) based on covariance information stored as part of \( a^i_{\text{aux}} \). The splat estimator \( \Phi_S \) is based on existing normal and covariance information and outputs ellipse parameters as part of \( a^i_{\text{aux}} \). For its calculation, the fairness operator \( \Phi_F(p) \) uses some temporary attributes \( a^i_{\text{aux}} \) but adds no output fields. (See also Appendix Figure 10.)

5.2 Stream Operator Classes

Each stream operator \( \Phi_i \) behaves like a buffer \( Q_i \) on the stream of points. After being released from the previous operator \( \Phi_{i-1} \), respectively its buffer \( Q_{i-1} \), a point \( p_i \) enters the next queue \( Q_i \). When all necessary neighborhood conditions are met, operator \( \Phi_i(p_i) \) is performed. The conditions when a point \( p_j \in Q_k \) can be processed by \( \Phi_i(p_j) \) and released to the subsequent operator \( \Phi_{k+1} \) and its queue \( Q_{k+1} \) depend on the type of the stream operator \( \Phi_k \).

The semantic of the buffer \( Q_k \) of a stream operator \( \Phi_k \) is equivalent to a FIFO queue (interface given in Appendix Figure 11) which includes the \textit{front()} and \textit{pop_front()} methods. However, instead of a \textit{push_back()} interface we define the exchange of points between operators as a \textit{pull-push} mechanism, see also Section 5.3. For this each operator \( \Phi_k \) keeps a reference to its previous operator \( \Phi_{k-1} \) in the operator pipeline. Other stream-operator functions include queries on the \textit{smallest element} – index \( i \) of a queued point \( p_j \in Q_k \) on which operator \( \Phi_k \) has not yet actually been computed; and the \textit{smallest referenced neighbor} – index \( j \) of a point \( p_j \in N_i \) of any unprocessed points \( p_i \) in \( Q_k \).

5.2.1 Through-buffer Operators

All stream operators \( \Phi_i(p) \) that given a set of attributes \( A_i = a^i_{\text{all}} \) compute additional new attributes \( a^i_j \) for a point \( p_j \) without affecting any \( k \)NN data in \( N_j \) are called through-buffer operators. This arises from the fact that as soon as a point \( p_i \) is released from a prior operator \( \Phi_k \) it can be processed by \( \Phi_k \) and immediately released to \( \Phi_{k+1} \). In practice its FIFO queue \( Q_k \) will generally be empty as the subsequent operator \( \Phi_{k+1} \) consumes any released points immediately.

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

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 \( \Phi_i(p) \) that either affect the use of \( p_j \in N_j \) in processing other nearest-neighbor related points \( p_j \in \Phi_i(p_j) \), or that modify the neighbor data \( N_j \) of the current point \( p_i \). We observe that:

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

Hence the \textit{pull_push()} method (given in Appendix Figure 14) must pre- as well as post-buffer the processed points \( p_i \). Two queues are necessary to implement the stream operator’s buffer \( Q_k \), one for buffering points \( p_i \) before and one after applying \( \Phi_k \). The \textit{pull_push()} method first grabs all points \( p_i \) released from the pre-
ceding operator \( \Phi_{k-1} \) and queues them in FIFO1. Next, the queue FIFO1 is checked for available points \( p_i \) that can now safely be processed by \( \Phi_k \) and queued in FIFO2. This requires testing for the smallest unprocessed and smallest referenced indices in the previous operators \( \Phi_{k-1} \).

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 \( \Phi_k \). Note that the top-most element \( p_i \) of FIFO2 is only released by operator \( \Phi_k \) if it no more references any point \( p \in \mathcal{N}_i \) which is still in the pre-buffer FIFO1 of \( \Phi_k \). This satisfies the constraints that when \( p_i \) is released to the next operator \( \Phi_{k+1}(p_i) \), \( \Phi_{k+1} \) will not operate on a neighborhood \( \mathcal{N}_i \) of \( p_i \) consisting of mixed points \( p_j \) – with respect to being processed or not by the operator \( \Phi_k \).

This category of stream operators \( \Phi_k(p_i) \) must carefully keep track of the smallest index \( j \) of any point \( p_j \in \mathcal{Q}_i \), and the smallest referenced neighbor index \( j \) of any \( p_j \in \mathcal{N}_i \) of any unprocessed point \( p_i \) in \( \mathcal{Q}_i \). This is achieved by maintaining a heap structure of indices for this purpose (see also Appendix Figure 14 and Appendix Figure 15).

The \( k\text{NN} \) and the fairing operators (Sections 4.2.2 and 4.3.4) are pre- and post-buffer stream operators. The fairing operator \( \Phi_{k}(p_i) \) changes the coordinates of a point \( p_i \) and must avoid that any stream operators \( \Phi_{k+1}(p_j) \) act on a mix of pre- and post-faired points \( p_j \in \mathcal{N}_i \). The \( k\text{NN} \) stream operator \( \Phi_k \), however, exhibits a few notable differences. First, the queue FIFO1 is replaced by a \( k\text{D} \)-heap structure as explained in Section 4.2.2 and this \( k\text{D} \)-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 \( k\text{NN} \) sets and removed from the \( k\text{D} \)-heap.

The curvature operator \( \Phi_{C}(p_i) \) 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 \( p_i \in \mathcal{N}_i \) has been released from the prior stream operator \( \Phi_{C-1} \). That is because \( \Phi_{C}(p_i) \) depends on the normals \( n_i \) of all \( k \)-nearest points \( p_j \in \mathcal{N}_i \) 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 mapping 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.

```c
// main loops for processing stream of points
while (operators[0]->position() < npoints)
  for (i = 0; i < npops; i++)
    opm[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 \( p_1, \ldots, p_m \) and their nearest neighbors. In general, a balanced hierarchical spatial index structure requires \( O(n) \) space and allows processing all points and \( k\text{NNs} \) in \( O(kn \log n) \) 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 \( m<n \) points are active at any time. The active set \( \mathcal{A} = p_1, \ldots, p_{m-1} \) consists of points not fully processed for which a new point \( p_i \) 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 \( k\text{NN} \) query, the expected running time is only \( O(kn \log m) \). This corresponds to a significantly reduced cost for the stream-processing approach.

As reported in the experimental results section below, the computation of all \( k\text{NNs} \) is dominating the overall workload. Therefore, the end-performance will strongly depend on the parameter \( k \) (proportionally) and the number \( s \ll m \) (logarithmically) of points in the \( k\text{D} \)-heap of the nearest neighbor stream operator \( \Phi_k \).

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>
<thead>
<tr>
<th>Model</th>
<th>#Points</th>
<th>Mesh File Size</th>
<th>Point Stream File Size</th>
<th>Preprocess reading/sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>St. Matthew</td>
<td>102,965,501</td>
<td>N/A</td>
<td>1,371 MB</td>
<td>35 s</td>
</tr>
<tr>
<td>David 1mm</td>
<td>26,168,109</td>
<td>2,197 MB</td>
<td>430 MB</td>
<td>125 s</td>
</tr>
<tr>
<td>Lucy</td>
<td>14,022,961</td>
<td>1,085 MB</td>
<td>214 MB</td>
<td>525 s</td>
</tr>
<tr>
<td>David 2mm</td>
<td>4,129,534</td>
<td>327 MB</td>
<td>639 MB</td>
<td>19 s</td>
</tr>
<tr>
<td>David head</td>
<td>2,000,646</td>
<td>185 MB</td>
<td>308 MB</td>
<td>12.5 s</td>
</tr>
</tbody>
</table>

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: \( \Phi_R \) (read), \( \Phi_X \) (\( k \)-nearest neighbors, \( k=8 \)), \( \Phi_N \) (normal estimation) and \( \Phi_W \) (deferred-write).
• Curvature: $\Phi_k, \Phi_X (k=8)$, $\Phi_N, \Phi_C (curvature)$, $\Phi_E$ (elliptical splat estimation) and $\Phi_W$.
• Fairing: $\Phi_k, \Phi_X (k=64...384)$, $\Phi_N, \Phi_E, \Phi_C$ (smoothing), $\Phi_N, \Phi_E$ and $\Phi_W$.

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 $A$ 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 $A$.

<table>
<thead>
<tr>
<th>Model</th>
<th>Pipeline</th>
<th>Point Stream Output Size</th>
<th>Timing Process</th>
<th>Lifespan</th>
</tr>
</thead>
<tbody>
<tr>
<td>St. Matthew</td>
<td>Normal</td>
<td>3,142MB</td>
<td>5:02:25</td>
<td>7.56s</td>
</tr>
<tr>
<td></td>
<td>Curvature</td>
<td>6,284MB</td>
<td>7:51:14</td>
<td>13.08s</td>
</tr>
<tr>
<td>David 1mm</td>
<td>Normal</td>
<td>859MB</td>
<td>2:33:56</td>
<td>23.62s</td>
</tr>
<tr>
<td></td>
<td>Curvature</td>
<td>1,719MB</td>
<td>2:52:45</td>
<td>29.27s</td>
</tr>
<tr>
<td>Lucy</td>
<td>Normal</td>
<td>428MB</td>
<td>25:32</td>
<td>4.75s</td>
</tr>
<tr>
<td></td>
<td>Curvature</td>
<td>856MB</td>
<td>34:25</td>
<td>6.17s</td>
</tr>
<tr>
<td>David 2mm</td>
<td>Normal</td>
<td>125MB</td>
<td>6:02</td>
<td>0.82s</td>
</tr>
<tr>
<td></td>
<td>Curvature</td>
<td>252MB</td>
<td>7:50</td>
<td>1.35s</td>
</tr>
<tr>
<td>David head</td>
<td>Normal</td>
<td>61MB</td>
<td>2:53</td>
<td>0.66s</td>
</tr>
<tr>
<td></td>
<td>Curvature</td>
<td>122MB</td>
<td>3:43</td>
<td>1.45s</td>
</tr>
</tbody>
</table>

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 $A$ 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 $A$ (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 $A$ 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 $A$ 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 $A$). 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 $A$ 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.

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).

7.2.4 Performance
While the current implementation is not optimized for performance, the experiments show that the major cost is the determin-
tion of all $k$NN as shown in Figure 6 for the Curvature stream-processing pipeline. The extra large $k$NN search cost for the David 1mm model stems from the fact that for this model the stream operator $\Phi_X$ 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 $k$NN buffer is the main performance factor as it contributes to an expected $O(k \log m)$ $k$NN search cost for each point. The other operators only add constant cost factors as they operate on the fixed $k$NN 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.](image)

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.](image)

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.

![Figure 8. Original smooth surface (top); random noise of 0.05% of diagonal length added to each coordinate (middle); and smoothed model using our stream-process fairing operator (bottom).](image)

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.

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 multiresolution generator 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 SIG-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 certainly be addressed in follow up work.REFERENCES

Stream-Processing Points
Renato Pajarola
Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

A. Code Appendix

```c
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
  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.

```c
class StreamOperator {
public:
  StreamOperator();
  ~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.

```c
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;
};
```

```c
void ThroughBuffer::pull_push() {
  AllFields *tmp;
  // pull elements from previous stream operator
  while (tmp = prev->front()) {
    prev->pop_front();
    applyOperator(tmp);
    FIFO.push_back(tmp);
  }
}
```

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

```c
AllFields* ThroughBuffer::front() {
  AllFields *tmp = NULL;
  if (!FIFO.empty())
    tmp = FIFO.front();
  return tmp;
}
```

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

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

```c
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()) {
            tmp = FIFO1.front();
        // defer processing points
            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() && (FIFO2.empty() ||
        !FIFO1.empty() && FIFO1.front()->max_ref_index < FIFO1.front()->index)) {
        tmp = FIFO2.top();
    } else if (!FIFO1.empty()) {
        tmp = FIFO1.top();
    } else return prev->smallest_element();
    return index;
}

int PrePostBuffer::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.

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[0] = new ReadOperator(pfile, nv);
    operators[1] = new KNearestOperator();
    operators[2] = new NormalOperator();
    operators[3] = new SplatOperator();
    operators[4] = new FairOperator();
    operators[5] = new NormalOperator();
    operators[6] = new SplatOperator();
    operators[7] = new WriteOperator(sfile, nv);
    // 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.