Stream-Processing Point Data

Renato Pajarola and Miguel Sainz

UCI-ICS Technical Report No. 04-02
Department of Computer Science
University of California, Irvine

March 2004

Stream-Processing Point Data

Renato Pajarola[*] Miguel Sainz[*]

Computer Graphics Lab
Computer Science Department
University of California Irvine

Abstract

With the fast increasing size of captured 3D models, i.e. from high-resolution laser range scanning devices, it has become more and more important to provide basic point processing methods for large raw point data sets. In this paper we present a novel stream-based point processing framework that orders unorganized raw points along a spatial dimension and processes them sequentially. The major advantage of our novel concept is its extremely low main memory usage and its applicability to process very large data sets out-of-core in a sequential order. Furthermore, the framework supports local operators and is extensible to concatenate multiple operators successively.

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

1. Introduction

Points as rendering and modeling primitives have become a powerful alternative to traditional polygonal object representation. In fact, points or 3D coordinates are the fundamental geometry-defining entities. Satisfying provably correct surface sampling criteria as discussed in [17], a set of points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ in 3D space fully defines the geometry as well as the topology of a surface including boundaries, components and genus. We assume here that input point sample data sets reasonably sample the represented surface, i.e. satisfying the Nyquist sampling criteria. Points are also the natural raw output data primitives of the 3D geometry capturing stage in most 3D geometry acquisition systems, i.e. laser range scanning or large objects [15].

With the dramatically increasing use and precision of 3D capturing devices it is critical to support raw point cloud data in a practical way. In particular, basic point processing operations such as normal direction estimation or fairing must be supported. Such operators can only be computed efficiently if the unorganized point data can be loaded into main memory and organized in some spatial indexing data structure. This approach, while optimal up to some size limit, will decrease quickly in efficiency when the models exceed available physical main memory. In the case of large mismatch between model size and physical main memory size it may almost cause this *main memory* approach to come to a halt. Furthermore, combining multiple operations cannot easily be addressed in an efficient way by merely combining multiple stages of applying operators.

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

The operations that are supported by the proposed concept are local operators $f(\boldsymbol{p})$ that perform a function on a point $\boldsymbol{p}$ and its local neighborhood. Many basic operations such as normal estimation on raw point data sets follow this principle and require the definition of a local neighborhood and then perform a computation of attributes based on this neighborhood [14, 17]. Also filter operations such as fairing are in this category and in general require a local neighborhood to operate on. Surface parameter estimation and filter operations are amongst the standard tasks for processing points. Our new concept supports any direct local operator $f(\boldsymbol{p})$ that is non-recursive in nature and that does not include any traversal of the samples beyond a limited local neighborhood.

2. Related Work

Conceptually, the most closely related work to our approach are sweep-line techniques in computational geometry [dBvKOS97]. The basic concept follows that idea of sweeping a line, or a plane in 3D, through the data set and consider the events when a data element is passed by the sweep-line. We extend this concept to processing a stream of points in 3D and to allow for several operations to be performed in consecutive stages.

Further related work is the approach to process triangle meshes in a sequential order [8, 9]. This technique grows triangle mesh regions sequentially in a fixed order that limits main memory usage by keeping information of active elements only. In particular, the active elements only consist of a small fraction of the mesh. This technique provides very efficient compression [8], rendering and simplification [9] of very large meshes. While this technique operates on a data set with a well defined and compact neighborhood definition given by the mesh connectivity, our proposed framework operates on a much lower level of processing raw points.

A very interesting approach to treat points in a sequential way has been proposed in [4]. There, a novel fast point rendering approach has been proposed by linearizing a multiresolution hierarchy into a single sequence of nodes, and basically performing level-of-detail selection and rendering directly on the GPU. The method does not address any low-level processing tasks and is not directly applicable to large data sets exceeding physical main memory or geometry cache sizes.

Since points as rendering primitives have been discussed in [16] and [12], many rendering techniques and systems have been proposed such as [25, 23, 24, 2, 11, 2]. In general these techniques address high-level point processing tasks such as multiresolution modeling and rendering of points with normals and spatial extent. More low-level processing techniques on point data sets can be found in [18, 20]. However, they are aimed at processing moderate point set sizes in main memory and assume that some basic processing such as an initial normal estimation has been done beforehand.

3. Basic Sweep-Plane Concept

The basic concept of our framework is to process the input points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ in a sequential order such that each point $\boldsymbol{p}_i$ is read only once from the input-stream, kept in an active working set $\mathcal{A}$ for some minimal amount of time and then written to the

[*]pajarola@acm.org, msainz@ics.uci.edu

output-stream as illustrated in Figure 1. This dramatically increases memory-access coherency for large data sets as all operations only involve points from the rather small set $\mathcal{A}$. Moreover, since the set $\mathcal{A}$ is orders-of-magnitude smaller then the entire data set it can generally be maintained efficiently in main-memory and because input and output are streams of points this directly leads to an out-of-core framework for stream-processing points.

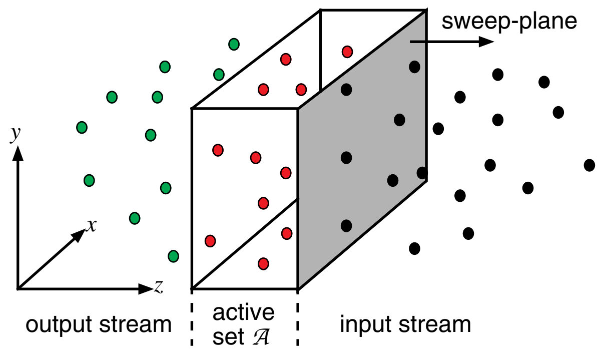

FIGURE 1. Sweep-plane process overview

As shown in Figure 1, we conceptually move a *sweep-plane* through space along the z-axis (without restricting the generality of this concept to any direction). Every time a new point from the input stream is hit by the sweep-plane it is added to $\mathcal{A}$. The active set $\mathcal{A}$ is continuously monitored and the local operator $f(\boldsymbol{p})$ is applied to the point $\boldsymbol{p}$ for which all necessary data is available. Furthermore, the smallest element of $\mathcal{A}$ is examined and if it does not possibly contribute anymore to applying the operator f to any future point it can safely be written to the output stream. In the remainder of this paper we will use *small* and *large* in the context of the sequential ordering. In the following section we describe in more detail the process for local neighborhood construction and application of a local operator.

When points have not yet been read from the input stream, or after they have been written to the output stream, their data generally resides on external disk memory, or possibly within some memory mapped virtual file memory. On the other hand, when elements *live* in the active set $\mathcal{A}$ they reside in virtual main memory and extra data is temporarily stored such as a list of k-nearest neighbors and other attributes.

Since raw point data sets rarely come with the 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 [12,24], and in practice the *rsort* [15] implementation has been used for similar tasks.

4. Nearest Neighbors and Local Operator

4.1 Nearest Neighborhood Determination

Any local operator $f(\boldsymbol{p}_i)$ requires information about the local neighborhood of points $K_i = \{\boldsymbol{p}_{i_1}, \ldots, \boldsymbol{p}_{i_k}\}$ surrounding $\boldsymbol{p}_i$. This neighborhood K_i can either be defined by a maximal *range* or as a number k of nearest neighbors. We will mainly focus on the k-nearest neighbors here but a fixed *range* can also be supported efficiently by varying k for each point.

To compute all k-nearest neighbors efficiently, or any neighborhood set for that matter, it is essential to maintain a spatial index structure Q_K over the the active set $\mathcal{A}$ that allows for very selective spatial (range-) queries. However, since we are processing a stream of points and want the spatial data structure to hold as few active elements as possible, we must remove the smallest active element from this data structure at the earliest possible time. Thus the index Q_K must also incorporate a priority-queue in the sequential ordering. Furthermore, despite frequent insertions and deletions this spatial data structure must also be reasonably balanced to support efficient access. Hence the main challenges in defining and implementing Q_K are:

- exhibit strong spatial selectivity
- support dynamic insertions of new points $\boldsymbol{p}_i$
- support removal of smallest priority-queue element $\boldsymbol{p}_{i-m}$
- maintain a balance of O(log m)
- preserve the sorted priority-queue property dynamically

The selection of choice for Q_K is a kD-heap that combines a dynamic, quasi balanced kD-tree with a priority heap. The kD-tree was selected over other spatial data structures due to its simple binary tree structure, efficient incremental insertion and removal operations, and the ability to influence the balance dynamically. In fact, since the points are streamed in one dimension it makes sense to only have a two-dimensional kD-tree partitioning the sweep-plane. The priority-queue is integrated with the kD-tree as a heap that parallels the kD-tree binary tree structure. Each kD-tree node holds a pointer to the smallest element in its subtree. Figure 2 shows an example kD-tree with alternating x,y split dimensions and heap structure in the z dimension.

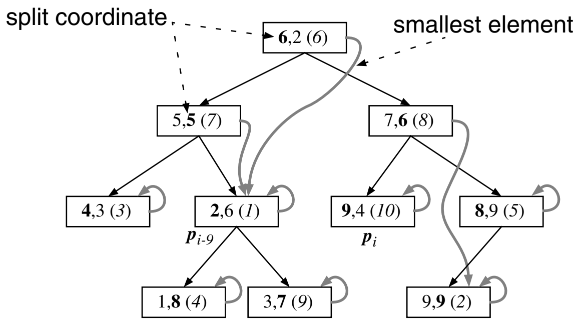

FIGURE 2. Integration of kD-tree and priority-heap.

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

A more complex operation is the k-nearest neighbor update which is solved in two phases. In the first phase, just before the next element $\boldsymbol{p}_i$ read from the input stream is inserted, a query on Q_K finds the smaller one-sided k-nearest neighbors K_i which are all smaller than $\boldsymbol{p}_i$, since Q_K only contains previous points in the sequential ordering. The element $\boldsymbol{p}_i$ is then inserted into Q_K with its temporary neighborhood K_i. This starts the second k-nearest neighbor search phase where $\boldsymbol{p}_i$ resides in the kD-heap – the first stage while being in the active set $\mathcal{A}$ as shown in Figure 3 – and K_i is continually updated. During the above query when an element $\boldsymbol{p}_{i-j}$ in Q_K is tested to update K_i, simultaneously $\boldsymbol{p}_i$ is tested to update K_{i-j}. Therefore, K_i is continually updated with the larger one-sided k-nearest neighbors while $\boldsymbol{p}_i$ is in Q_K and new elements $\boldsymbol{p}_{i+j}$ are inserted.

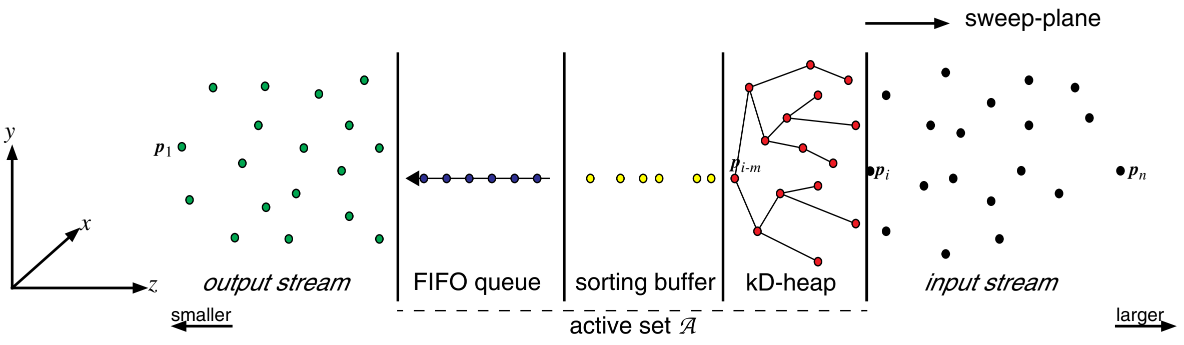

FIGURE 3. Stages of a simple stream-processing pipeline. A point moves from right-to-left through these stages.

The last major operation on the kD-heap Q_K is the removal of smallest elements. As it is imperative to keep the size of Q_K as small as possible for fast k-nearest neighbor updates, we need to remove elements whose k-nearest neighborhood is completed as early as possible. Therefore, our kD-heap supports a query to find a sorted list L, smallest first, of elements $\boldsymbol{p}_j$ in Q_K for which the current sweep-plane has moved beyond the farthest, kth-nearest neighbor in K_j. The elements of this list L can then be removed from Q_K. However, note that L is only partially sorted with respect to the sequential ordering and hence its elements cannot immediately be passed to the next stage. Therefore, the elements $\boldsymbol{p}_j$ of L are first passed to a sorting buffer B – the second stage while being in the active set $\mathcal{A}$ as shown in Figure 3 – that re-establishes the global ordering. The smallest element $\boldsymbol{p}_j$ of B has regained its correct global ordering as soon as it is smaller than the smallest element in Q_K and in that case it can be passed to the next processing stage.

4.2 Local Operator

After passing the kD-heap Q_K and when leaving the sorting buffer B, an element $\boldsymbol{p}_j$ has a fully determined k-nearest neighborhood K_j and the local operator $f(\boldsymbol{p}_j)$ can be applied. However, it must be guaranteed that all elements referred to by K_j can be accessed before performing $f(\boldsymbol{p}_j)$. Note that K_j refers to elements that can be smaller as well as larger than $\boldsymbol{p}_j$ in the sequential ordering. Elements that are larger are guaranteed to be maintained in the active set $\mathcal{A}$ either by the current kD-heap Q_K or the sorting buffer B. On the other hand, also the elements of K_j smaller than $\boldsymbol{p}_j$ must be kept in the active set $\mathcal{A}$. Hence upon leaving the sorting buffer B, just after applying the operator $f(\boldsymbol{p}_j)$, the element $\boldsymbol{p}_j$ cannot be discarded yet and is passed to a FIFO queue F – the third and last stage while being in the active set $\mathcal{A}$ as shown in Figure 3. This FIFO queue F assures that every element $\boldsymbol{p}_l$ referenced by any nearest neighbor set K_j of a larger point $\boldsymbol{p}_j$ in the sorting buffer B or the kD-heap Q_K is forced to stay in the active set $\mathcal{A}$.

Therefore, the first element $\boldsymbol{p}_l$ of F can leave the active set $\mathcal{A}$ and be written to the output stream when it is not anymore referenced by any other (larger) element $\boldsymbol{p}_j$ in $\mathcal{A}$. This relation is managed by a simple priority queue ZQ, over all elements $\boldsymbol{p}_j$ in Q_K and B, that maintains the smallest referenced element (z-reference) in $K = \cup K_j$. Thus the first element $\boldsymbol{p}_l$ of F has fully completed its *service* in the active set $\mathcal{A}$ when it is beyond the smallest reference managed by ZQ. This completes the three stage stream-processing pipeline shown in Figure 3 for establishing a k-nearest neighborhood and computing a local operator. The local operator $f(\boldsymbol{p}_j)$ can be applied on $\boldsymbol{p}_j$ and its neighborhood K_j when it is moved from the sorting buffer B to the FIFO queue F.

As prototypical local operator $f(\boldsymbol{p}_j)$ applied to K_j we select the normal estimation $N(\boldsymbol{p}_j)$ by *least squares fitting* [14, 1, 22, 17] to demonstrate our basic framework. This computes the orientation of a fitting plane through a set of points by an eigenvalue and eigenvector analysis of the covariance matrix. We adopted a *moving least squares* (MLS) approach similar to [1] with a weighted covariance

$$M_j = (|K_j| + 1)^{-1} \cdot \sum_{\boldsymbol{p} \in K_j \cup \boldsymbol{p}_j} (\boldsymbol{p} - \bar{\boldsymbol{p}}) \cdot (\boldsymbol{p} - \bar{\boldsymbol{p}})^{\mathrm{T}} \cdot \theta(|\boldsymbol{p} - \bar{\boldsymbol{p}}|) .$$

The average of the point set $K_j \cup \boldsymbol{p}_j$ is given by $\bar{\boldsymbol{p}}$. The weight function $\theta(r)$ is a smooth, radially symmetric, monotone decreasing Gaussian function $\theta(r) = \mathrm{e}^{-r^2/2\sigma^2}$. The variance σ^2 is adaptively defined as the local point density estimate $\pi \cdot \mathrm{MAX}_{\boldsymbol{p} \in K_j} (\boldsymbol{p} - \boldsymbol{p}_j)^2 / |K_j|$, see also [17]. Thus the normal $\boldsymbol{n}_j$ of a point $\boldsymbol{p}_j$ is computed as the eigenvector of the MLS covariance M_j over $K_j \cup \boldsymbol{p}_j$ corresponding to the smallest eigenvalue of M_j. Potentially it is numerically more robust to define the normal as the normalized vector product of the two eigenvectors corresponding to the two largest eigenvalues of M_j.

In fact, in addition to the normal $\boldsymbol{n}_j$ estimate of a point $\boldsymbol{p}_j$ we also compute a bounding sphere radius r_j over all points in K_j, and also an elliptical disk approximating the anisotropic distribution of points in K_j. These attributes can be used by a point-based rendering system to visualize the processed point set as circular or elliptical disks.

5. Fairing

To demonstrate the applicability of the proposed stream-processing framework we introduce a smoothing filter operation. However, processing data as a stream does not allow for iterative or recursive operations to be performed. For this reason we adopt the non-iterative feature preserving fairing operator presented in [10]. Its applicability to triangle soups makes it also suitable for modification to point sets. Fairing along the lines of [10] requires a tangent plane normal estimate which we described in the previous section. Furthermore, it also requires accessing spatially close neighbors. Therefore, the basic framework to establish a k-nearest neighborhood presented in Section 4 directly applies to the fairing operator as well.

Given a point $\boldsymbol{p}_i$ and its neighbors K_i, we extend the smoothing operation Φ to points as follows

$$\boldsymbol{p}_i' = \Phi(\boldsymbol{p}_i) = w_i^{-1} \cdot \sum \Pi_j(\boldsymbol{p}_i) a_j \cdot f(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \cdot g(|\Pi_j(\boldsymbol{p}_i) - \boldsymbol{p}_i|) ,$$

with summation over all points $\boldsymbol{p}_j \in K_i \cup \boldsymbol{p}_i$. 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. The normalization term w_i is the sum of weights $\sum a_j \cdot f(|\boldsymbol{p}_j - \boldsymbol{p}_i|) \cdot g(|\Pi_j(\boldsymbol{p}_i) - \boldsymbol{p}_i|)$. The Gaussian weight function $f(r)$ adjusts the influence of a point based on spatial distance and favors nearby points for smoothing, while $g(r)$ tends to preserve sharp features by giving less weight to points with different normal orientations [10]. All necessary information to perform this fairing operator $\Phi(\boldsymbol{p}_i)$ is given by the stream-processing framework as described in Section 4. This includes the normal estimates $\boldsymbol{n}_j$ used for the projection Π_j as well as circular or elliptical bounding disks used to calculate the area a_j of a point $\boldsymbol{p}_j$.

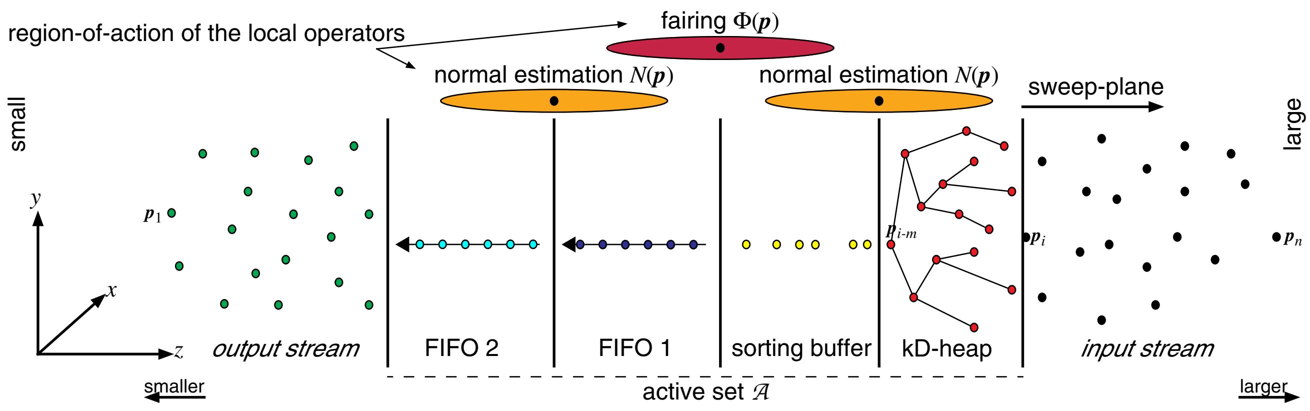

FIGURE 4. Stages of a complex stream-processing pipeline for fairing. Region-of-action overlap is indicated for the different local operators.

The outlined smoothing operator $\Phi(\boldsymbol{p}_i)$ is a local operator similar to the normal estimation $N(\boldsymbol{p}_i)$ explained in Section 4.2 and operates on $\boldsymbol{p}_i$ and its k-nearest neighbors K_i. Note, however, that its application must fit into the stream-processing pipeline correctly with respect to available and referenced data as illustrated in Figure 4. In particular, smoothing points makes it necessary to recompute a new normal, as well as circular or elliptical bounding disk parameters, after applying the fairing operator. Hence we apply a normal estimator $\boldsymbol{n}_i = N(\boldsymbol{p}_i)$ followed by the fairing operator $\boldsymbol{p}_i' = \Phi(\boldsymbol{p}_i)$ and again followed by a normal estimation $\boldsymbol{n}_i' = N(\boldsymbol{p}_i')$. Moreover, as the fairing operator $\Phi(\boldsymbol{p}_i)$ changes the coordinates of points it must strictly be avoided that the normal estimators operate on a mix of faired and non-faired points. Keeping a second copy of the original input point coordinates, however, this can be avoided and the fairing operator can overlap both normal operators in the sequential processing of points as shown in Figure 4.

The different data structures and stages of the active set $\mathcal{A}$ for a combined normal estimation with bounding disks, and fairing operator are illustrated in Figure 4. In contrast to the basic stream-processing stages described in Section 4, a second FIFO queue is necessary. The sorting buffer B is also modified in its behavior in the following way. Since the fairing operator $\Phi(\boldsymbol{p}_i)$ changes $\boldsymbol{p}_i$ it can only be applied to points that are not in the range of the k-nearest neighborhood K_j of any larger point $\boldsymbol{p}_j$. This is assured by checking a priority queue $ZQ1$ equivalent to the one explained in Section 4. Moreover, the neighborhood K_i can only reference elements $\boldsymbol{p}_l$ that have passed the normal operator since their normals $\boldsymbol{n}_l$ are required for fairing. These requirements extend the scope of the sorting buffer B and the time elements remain in it. The fairing operator $\Phi(\boldsymbol{p}_i)$ is then applied to the elements $\boldsymbol{p}_i$ leaving this modified sorting buffer stage and which are passed to the following FIFO queue $F1$. Because we have a third operator to recompute the normal of the smoothened points, the queue $F1$ buffers points until the set K_i of its smallest element $\boldsymbol{p}_i$ only refers to points that have been faired before, thus are smaller than any element in the sorting buffer B. The second normal estimation is performed when a point leaves $F1$. Finally the FIFO queue $F2$ is necessary to keep all elements $\boldsymbol{p}_i$ active which are needed by the second normal operator. Therefore, the first element $\boldsymbol{p}_i$ of $F2$ can leave the active set $\mathcal{A}$ when it is not referenced by any larger element $\boldsymbol{p}_j$ in $\mathcal{A}$. This relation is assured by another priority queue $ZQ2$ managing the smallest references from all elements $\boldsymbol{p}_j$ in Q_K, B and $F1$.

This completes the extended normal estimation and fairing stream-processing pipeline shown in Figure 4.

6. Analysis

In terms of memory requirements we note that the most critical part in all processing stages is the spatial data structure to provide efficient access to all points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ and their neighbors. In general, constructing as well as querying a balanced spatial data structure requires $O(n \log n)$ time and $O(n)$ space. While this is optimal it may nevertheless consume too much main memory. Our stream-processing framework has the property that only a limited number of $m \ll n$ points are active at any moment in time. The active set $\mathcal{A} = \boldsymbol{p}_{i-1}, \ldots, \boldsymbol{p}_{i-m}$ consists of points not fully processed for which a new point $\boldsymbol{p}_i$ on the sweep plane may be necessary to complete the processing task, or processed points which are still required by others to complete the processing task. Thus in main memory only the m active points must be maintained, organized in an indexing data structure requiring only $O(m)$ main memory space.

The processing performance is mainly determined by the spatial indexing and the determination of all k-nearest neighbors. Since the k-nearest neighbor query is only performed on the elements in the kD-heap Q_K of the active set $\mathcal{A}$, the cost of computing all k-nearest neighbors will strongly depend on the size and balance of the kD-heap data structure.

In the pre-process the points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ are sorted in one direction, using an external memory sort algorithm in $O(n \log n)$ time.

7. Experiments

All point processing experiments were performed on a PowerMac G4 dual 800MHz[1] CPU with 512MB main memory. Pre-process results for ordering the data sets sequentially are ommitted. All data sets are ordered along the z-dimension without considering the actual main orientation of the model.

The streaming of points was implemented using memory mapped arrays. The ordering pre-process writes the sequentially ordered points into a memory mapped file that is read subsequently by the stream-processing algorithm. The output stream is also written to a memory mapped file.

1. Note that the software is not parallelized and runs only on one CPU.

Model	#Points
David 1mm	28,168,109
David 2mm	4,129,534
David head	2,000,646
Lucy	14,022,961
Dragon	435,545
Female	302,948
Balljoint	137,062

TABLE 1. Test model sizes.

For *k*-nearest neighbor finding and normal estimation, Figure 5 shows the sizes of the various data structures of the active set with respect to the progress of the stream-process through the point sequence. One can see that while there is some variation, the general trend is that the active set data structures are orders-of-magnitude smaller than the input point set. In more detail, the kD-heap and the FIFO data structure maintain significantly more elements than the sorting buffer, which is expected in the case of a simple normal estimation operator as described in Section 4. The size of the kD-heap is the most important measuring factor as the *k*-nearest neighbor search heavily depends on it and consumes most of the processing time. The FIFO queue is also of much less importance.

Lucy exhibits some strong growth of the data structures up to maintaining around 300,000 points at a very early time but then dramatically drops to only manage some 20,000 points dynamically in the active set during the remainder of the stream-processing.

The David head model exhibits a similar peak at about 2/3 of the time with about 50,000 points in the kD-heap. For the most part, however, only about 10,000 or less points are dynamically managed in the active set.

The two David models, at 1mm and 2mm resolution, show an interesting scaling behavior. The smaller 4M point David model causes the system to dynamically manage around 25,000 points in the kD-heap. The seven times bigger 28M point David model on the other hand, while peaking a few times into 100,000, for the most part requires only less than 40,000 elements in the kD-heap.

Variations in the active set size are due to variations in the object shape and how the sweep-plane cuts through it, and also how the object is aligned with respect to the *z*-axis which defines the sequential ordering.

We also performed some experiments with the proposed fairing operator described in Section 5. Similar results are reported in Figure 6. One can observe that the main-memory data structures are generally orders-of-magnitude smaller than the processed input point data set. Hence our proposed stream-processing framework efficiently stream-processes arbitrary large point dat sets efficiently, and in particular with only very limited main memory usage.

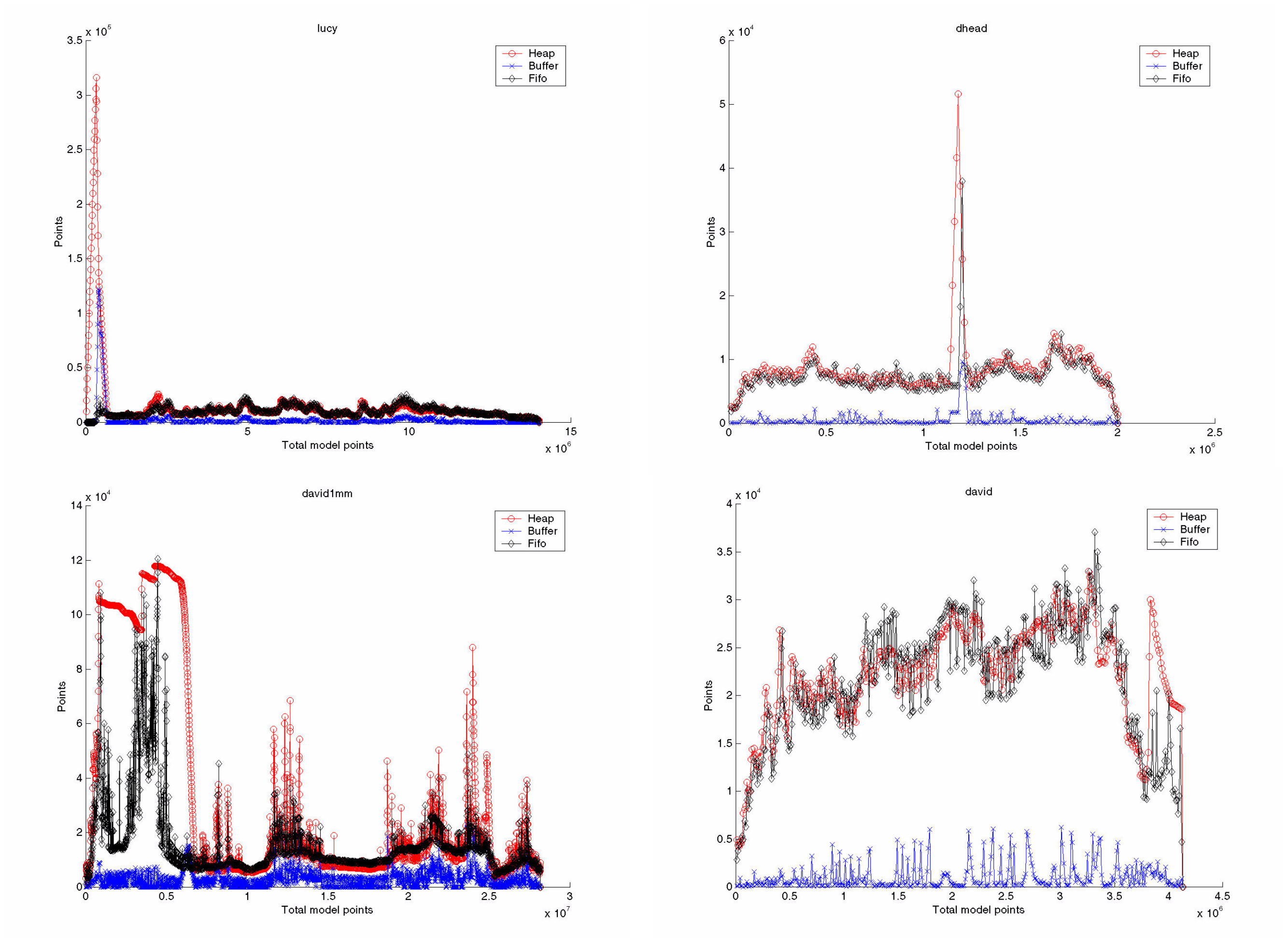

FIGURE 5. kD-heap, sorting buffer and FIFO queue sizes plotted against the progress through the input point stream.

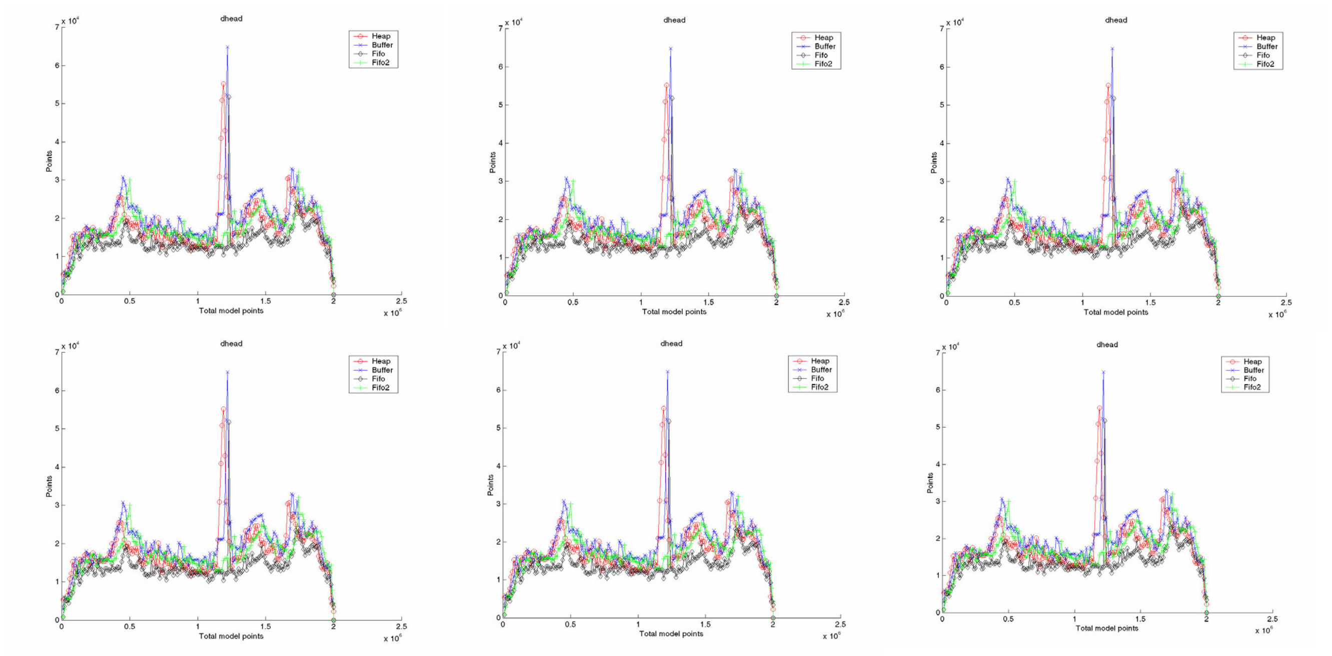

FIGURE 6. kD-heap, sorting buffer and FIFO/FIFO2 queue sizes plotted against the progress through the input point stream.

8. Discussions

We have presented a novel point processing framework based on a linear streaming of points and a sweep-plane algorithm for k-nearest neighborhood determination. 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. It is also the only framework that supports processing points as streams with only limited main memory usage and that is extensible to apply multiple local operators successively on the point set.

Several implementation details are not optimized in the current framework. At present, the aspect ratio of the bounding box of input models is not examined, even though a sequential ordering along the z-dimension may not always be the optimal. In fact, the sequential ordering should be performed along the axis of the longest bounding box side for better performance. Furthermore, most of the buffer and queue data structures are standard C++ STL template containers and not specialized implementations optimized for performance. While the kD-tree priority-heap is a custom data structure and has some minor balancing techniques incorporated, it is not the fastest possible implementation. Among the possible improvements is a much more agressive balancing strategy to keep the k-nearest neighbor query cost low. Further work would include the development of a specialized dynamically balanced spatial indexing structure.

The k-nearest neighborhood sweep-plane algorithm described in Section 4 can under certain circumstances generate an approximate k-nearest neighbor set instead of the exact solution. This has to do with out-of-k-nearest-neighbor-range tests that in fact are extremely difficult to answer exactly. For example, it is unclear how to quickly determine if a point cannot anymore contribute to the k-nearest neighbor set of any subsequent point reached by the sweep-plane. The first observation we can make here is that with several test models no difference to the exact solution was noticed. The second observation is that a provably exact k-nearest neighborhood determination may not even be necessary for local operators such as normal estimation and fairing. Additionally, if the framework is modified to compute a fixed-range d neighborhood with variable k for each point, then all out-of-range tests can exactly be performed with respect to d.

Acknowledgements

We would like to thank the Stanford *3D Scanning Repository* and *Digital Michelangelo* projects as well as *Cyberware* for freely providing geometric models to the research community.

References

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

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

[3] 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.

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

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

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

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

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

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

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

[11] 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.

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

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

[14] 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.

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

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

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

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

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

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

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

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

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

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