A System for Stream Processing of Point Data

Jonas Bösch Renato Pajarola
Visualization and MultiMedia Lab, Unviersity of Zürich
boesch@ifi.uzh.ch, pajarola@acm.org

Technical Report IFI-2008.03, Department of Informatics, Unviersity of Zürich

Abstract

To efficiently handle the continuously increasing raw point data set sizes from high-resolution laser-range scanning devices or baseline stereo and multi-view 3D object reconstruction systems, powerful geometry processing solutions are required. We present a flexible and run-time configurable system for efficient out-of-core geometry processing of point cloud data that significantly extends and greatly improves the stream-based point processing framework introduced in [27]. In this system paper we introduce an optimized and run-time extensible implementation, a number of algorithmic improvements as well as new stream-processing operators. As a consequence of the novel and improved system architecture, implementation and algorithms, a dramatically increased performance can be demonstrated as shown in our experimental results.

1. Introduction

Points as rendering and modeling primitives have become a powerful alternative to traditional polygonal object representation [32, 11, 12]. Note that point samples are the natural raw output data primitives of the geometry capturing stage in most 3D object acquisition systems. In fact, points or 3D coordinates are the fundamental geometry-defining entities. Satisfying provably correct surface sampling criteria as discussed in [24], a set of points in 3D space fully defines the geometry as well as the topology of a surface including boundaries, components and genus. Here we assume that input point data sets reasonably sample the represented surfaces.

With the continually increasing density and extent of raw point cloud data sets, effective algorithms and systems are required to cope efficiently with the massive amounts of point samples. Basic data and geometry processing operations must be supported such as noise removal, outlier detection, normal estimation, or data decimation with many more being conceivable. These operations can only be performed efficiently on large data if memory trashing [7] is avoided. Therefore, data must be paged efficiently into main memory and processed coherently with respect to randomly accessing memory locations.

In [27] the concept of stream-processing point data was introduced which we will briefly review in Section3. The basic idea was to sequentialize the unorganized raw input point data and then feed the resulting point stream through a pipeline of local stream operators. While the approach in [27] is conceptually well designed and showed promising results, it nevertheless is limited in a number of points. First of all, the configuration of the pipeline (chain) of stream operators had to be defined at compile-time, and in fact all selectable operators had to be known and implemented as well at this time. Furthermore, there was no concept of operator-chain overarching data structure to maintain any global information about all points currently residing in main memory. Third, the previous implementation left some room for performance improvements, e.g. pooling of dynamic memory resources. In this system paper we present an improved stream-processing framework that addresses all these issues, and eventually also introduces a number of new stream operators. The main technical contributions are:

i) A novel flexible C++-classes framework that defines run-time configurable geometry processing stream operators.

ii) New concept of chain-operators overarching a chain of individually configured stream operators.

iii) Improved implementation of neighborhood search operator and dynamic memory handling.

2. Related Work

Points as 3D surface modeling and rendering primitives have been introduced as early as in [21] and [13]. A number of efficient hardware supported rendering algorithms

such as [34, 33, 3, 2, 28] have been proposed and subsequently further improved. The fundamental theory and algorithms for point-based modeling and rendering are described in [12], and surveys on point-based rendering (PBR) have been presented in [37, 36] and [20]. Apart from rendering which has been well studied and extended to out-of-core [35, 10, 29] or transparent rendering [43], low-level geometry processing techniques for point data have been discussed in [30, 23, 31, 18, 41]. However, these methods are aimed at processing only moderately sized point sets that fit into main memory.

Sequential organization of point data has been addressed specifically for rendering and network transmission purposes in [5, 29] and [35]. More general low-level geometric operations are applied to a stream of points in [27], which we will review in the following section.

Streaming has chiefly been used in processing digital audio and video data which in contrast to 3D geometry is inherently sequentially organized, i.e. in time. The sweep-line concept in geometry processing [6] is conceptually closer than multimedia streaming, since our basic stream-processing follows a similar idea of sweeping a plane over the point cloud data. In the context of 3D geometry, streaming has been introduced for simplification and compression operations on polygonal meshes [16, 42, 17, 40], which generally grow and process mesh regions sequentially in an order that limits main memory usage. Specifically for rendering, a streaming mesh layout has been proposed in [15]. These streaming approaches on meshes, however, do not support low-level geometry processing operations, and more importantly, do not directly apply to raw point data processing as mesh connectivity is required.

Finally, in graphics the concept of streaming images and geometry data has been used in the context of remote rendering where 3D data is to be displayed on a remote display (e.g. [8], [25] or [4]). Again, low-level data processing is not the focus in these approaches but the network transmission of data to a remote device.

3. Stream-Processing Framework

In this section we briefly review the basic concepts of the stream-processing framework and operators introduced in [27].

3.1. Sequential Processing

The basic idea behind stream-processing point data is to order and process the data sequentially in such a way that: (1) points can be read from an input-stream into main memory one at a time, (2) the so called *active* points in main memory can efficiently be processed independently[1] from others given only some local spatial information, and (3) points are written early to an output-stream as soon as they and any dependent points have been processed. Figure 1 illustrates this basic concept of a sliding-window over the set of input points $\mathbf{p}_1,\ldots,\mathbf{p}_n \in \mathbb{R}^3$. Since all data processing is limited to the points in the active working set $\mathscr{A}$, at any time only a very limited fraction of data is kept in main memory, which together with the sequential processing supports efficient out-of-core operation on huge point data sets.

The set $\mathscr{A}$ is a FIFO queue keeping only the m currently active points $\mathscr{A} = \mathbf{p}_{j-m},\ldots,\mathbf{p}_m$ in main memory which are to be processed by a chain of local stream operators. As soon as $\mathbf{p}_{j-m} \in \mathscr{A}$ has been processed and is not required by an operation on any subsequent point $\mathbf{p}_{i>j-m}$ it can safely be written to the output stream.

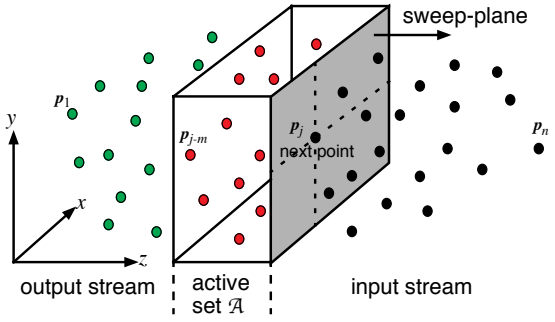

Figure 1. Window of an active set of points sliding over the stream of input point data.

Since raw point data sets rarely come in a spatially ordered sequence, a pre-process is required to linearly order them. Given an ordering measure along one direction in space, such sorting can efficiently be achieved for very large data sets by external sort techniques [19, 39], such as for example the *rsort* implementation [22]. A sorting based on the direction of the longest axis of a data-aligned tight bounding box can efficiently be achieved in two phases as follows. In the first linear pass over the data points $\mathbf{p}_1,\ldots,\mathbf{p}_n \in \mathbb{R}^3$, the generic homogeneous covariance $\widehat{\mathbf{M}} = \sum_{i=1}^{n} \hat{\mathbf{p}}_i \cdot \hat{\mathbf{p}}_i^{\mathsf{T}}$ and center of mass of the points $\mathbf{c} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{p}_i$ are accumulated, with $\hat{\mathbf{p}}$ denoting the homogeneous coordinate extension of $\mathbf{p}$. As shown in [26, 28] this allows us to express and post-compute the actual covariance matrix $\mathbf{M} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{p}_i - \mathbf{c}) \cdot (\mathbf{p}_i - \mathbf{c})^{\mathsf{T}}$ elegantly and efficiently in homogeneous space by $\mathbf{M} = \frac{1}{n}\mathbf{T}(-\mathbf{c}) \cdot \widehat{\mathbf{M}} \cdot \mathbf{T}^{\mathsf{T}}(-\mathbf{c})$ with $\mathbf{T}(-\mathbf{c})$ being the translation matrix moving the center of mass $\mathbf{c}$ to the origin. The sorting axis is now given by the eigenvector $\mathbf{v}$ correspond-

1 or more exactly the dependency is strictly limited to a well defined local spatial neighborhood relation

ing to the largest eigenvalue of **M**. In the second phase, the points are transformed into the new coordinate system given by the eigenvectors and then sorted based on their projection onto **v**.

3.2. Stream Operators

The operations supported in the above described stream-processing framework are defined in [27] as local operators $\Phi(\mathbf{p}_i)$ that perform a computation on a point $\mathbf{p}_i$ and its attributes only taking the point $\mathbf{p}_i$ itself and a limited set of spatial neighbors $\mathbf{p}_j$ into account. The neighborhood $\mathscr{N}_i$ is typically defined as a k-nearest neighbor set or range of points $\mathbf{p}_j$ within a given radius r. The attributes A_i associated with a point $\mathbf{p}_i$ can include a wide range of parameters such as color, normal orientation or curvature. From this definition it is clear that a local operator $\Phi(\mathbf{p}_i)$ can be applied to any point if $\mathbf{p}_i$ itself as well as its neighborhood points $\mathscr{N}_i$ are part of the current active working set $\mathscr{A}$. This includes a large group of important geometry processing operators ranging from surface parameter estimation to filtering operations.

Furthermore, the stream-processing framework is designed to chain together a series of stream operators $\Phi_1,\ldots,\Phi_p$ that are applied in succession to a stream of points as illustrated in Figure 2. Each stream operator Φ_k itself acts as a FIFO queue, passing the stream of points from one to the next operator. The so defined concept then postulates that a stream operator $\Phi_k(\mathbf{p}_i)$ can be executed on $\mathbf{p}_i$ as soon as no preceding operator $\Phi_{l<k}$ modifies any neighbor points $\mathbf{p}_j \in \mathscr{N}_i$ anymore, or still depends on $\mathbf{p}_i$ for its completion. Moreover, each stream operator Φ_k only passes a point $\mathbf{p}_i$ to the next operator Φ_{k+1} if the point and its attributes have fully been processed. More details on these requirements are given in [27].

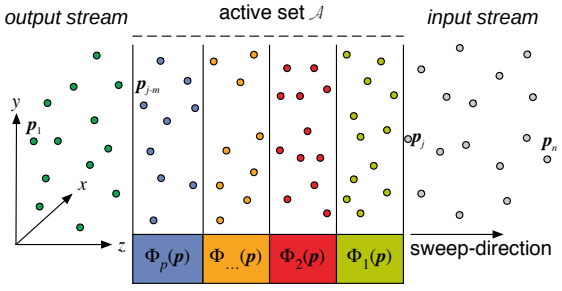

Figure 2. Chain of streamable operators acting on the points passing through the active set.

The fundamental stream operators introduced in [27] include the basic I/O operators for reading (Φ_R) and writing (Φ_W) points from and to the input and output streams respectively, as well as a neighborhood operator (Φ_X) that establishes the nearest neighborhood relation $\mathscr{N}_i$ for any incoming point $\mathbf{p}_i$. Additional regular geometry processing operators that have previously been presented include surface normal estimation (Φ_N), curvature estimation (Φ_C), elliptical point splat extent determination (Φ_E) as well as feature preserving surface smoothing (Φ_S).

4. System Architecture

In this section we discuss and address the limitations of the initial implementation of the stream-processing framework outlined in Section 3 as given in [27]. In particular, the new implementation approach shows significant improvement in flexibility and extensibility, and additionally results in greatly increased performance which is experimentally demonstrated in Section 7.

4.1. Run-time Configurability

In the original stream-processing approach [27], the operator chain was set up at compile-time. A stream operator defined a set of per-point attribute parameters that it depends on or modifies, some auxiliary data fields used while executing the operator and some attributes that it adds permanently to a point element, which are included and written to the output stream at the end of the stream operator chain. An example of the conceptual compile-time configuration of the attribute fields of stream operators is given in Figure 3. Every stream operator defines a struct that contains members variables for all required data. The auxiliary temporary and final stream-point output structure can be combined by simple multiple inheritance. While the main advantage of this approach is its simplicity, it also lacks in flexibility and consistency. Full flexibility can only be achieved by generating a separate executable for all possible combinatorial configurations of the different stream operators, which grows exponentially by 2^p with the number p of operators. For an increasing and extensible library of stream operators this is clearly a limiting constraint. Furthermore, there is no automatic mechanism to verify consistency of the attribute fields, e.g. such as ensuring that an attribute *xy* which operator X depends on is provided by another operator Y. A similar problem arises with the order of operators: While including the normal attribute field in `OutputFields` allows the inclusion of a curvature operator at compile time, it does not make sure that the normal estimation operator is actually applied first in the chain of stream operators.

```
struct Operator1Fields {
   float covariance[4][4];
}
struct Operator2Fields {
   float area;
}
struct Operator1OutputFields {
   float direction[3];
}
struct Operator3OutputFields {
   int counter;
   float area;
}
struct AxiliaryFields : Operator1Fields,
   Operator2Fields, Operator3Fields { };
struct OutputFields : Operator1OutputFields,
   Operator3OutputFields { };
struct AllFields : OutputFields,
   AxiliaryFields { };
```

Figure 3. Temporary main-memory and persistent output fields structure of the original, compile-time dependent stream-processing application introduced in [27].

Therefore, the stream-processing framework was redesigned to allow setting up an arbitrary operator chain at run-time, by using either an external configuration file or by specifying stream operators as command line arguments. Also, while the attribute fields of different operators are specified dynamically at run-time, a registration mechanism can verify that no required attributes are missing, and that the operators are specified in a compatible order.

In the new framework, dependencies between stream operators are defined solely by dependencies on certain data elements, and not the operators themselves. This has the advantage that every operator can be replaced as long as the replacement operator can generate the same output data fields. Additionally, since there are no direct code-level dependencies between operators, new operators can be integrated by loading them as separate plugins or dynamic shared libraries at run-time.

4.1.1. Alternative Solutions One approach for defining run-time-configurable objects and attribute fields is to use a container, e.g. a std::vector¡¿, of boost::any* or similar ”any-type”-objects [14]. In that case, the data members of a point **p** are indicated by an index into that vector of boost::any elements, and the element is accessed by casting it to the correct type before use. Figure 4 demonstrates the use of a vector of boost::any types to define the attributes of a point **p**.

The above mentioned solutions is quite simple, however, has some important drawbacks. First, the type of the i-th variable field $\mathbf{p}[i]$ is not defined in the header of the operator. boost::any instances are generic, and the type is defined by its first assignment. This means that an external specification for the type of each attribute field is required, and cannot be determined by analyzing the code. Furthermore, in the case of trying to access an attribute field using a wrong type, no compile-time error or warning will be provided, but a run-time error will be generated. In the context of the stream processing application, each stream point attribute field has a fixed type. Therefore, the boost::any approach would not be well suited to the problem domain, in contrast to the dynamic structures approach that we describe in the following section, where attribute fields can be declared in the stream operator header similar to a normal C++ variable. Moreover, boost::any objects are not compatible with memory pooling. Only the any-class wrapper object can be allocated beforehand, but not the actual data, since it is generated on first assignment. Only then the actual type and with it the memory footprint is known and can be allocated. Finally, boost::any uses *run-time type information* (RTTI) to determine the type on every access, which can introduce an additional performance penalty.

```
typedef std::vector<boost::any*> pdata;
int normal_index, nb_index;

void opA::set_normal(const vec3f &normal_) {
   pdata &point = get_current_point();
   vec3f &n = point[normal_index]->as<vec3f>();
   n = normal_;
}

float opB::compute_dotProduct(int nb_index) {
   // get point indices:
   pdata &point = get_current_point();
   pdata &nb = get_neighbor(nb_index);

   // get normal attributes:
   vec3f &n = point[normal_index]->as<vec3f>();
   vec3f &nb_n = nb[normal_index]->as<vec3f>();

   // this compiles, but it will generate
   // an error at run-time.
   char &nb_nf = nb[normal_index]->as<char>();

   return n.dot(nb_n);
}
```

Figure 4. Example of a possible vector-of-any-type implementation using flexible point attributes.

Another alternative would be to just use void* pointers, custom allocated memory fields, type lookup tables, and constant casting by the programmer. This system is easy to implement, but hard to use and quite error prone in practice, as there is neither any kind of type safety, nor a reliable way to detect wrongly assigned types during either compile- or run-time, except for trying to detect invalid output data and/or possible crashes.

4.1.2. Run-time Structures In a dynamic run-time configurable version of the stream-processing framework, the size of the structure holding the attribute fields is unknown at compile-time. Therefore, we designed a new *rt_structs* (for run-time structures) concept for the streaming three-dimensional point data structure that efficiently enables the use of type-safe and run-time configurable structures and member variables. Figure 5 shows the class diagram of the main classes which include rt_struct, rt_struct_user, rt_struct_member, rt_struct_info, rt_struct_member_info and rt_struct_factory.

rt_struct The rt_struct data structure is defined as a class with no attribute fields, but with template methods to access the attribute field data as shown in Figure 6 as well as private standard and copy-constructors. At run-time, each rt_struct is allocated enough memory to hold all member attributes. Basically, it represents the storage of a point while it is in main memory as part of the active set. The standard and copy-constructor are kept private to force the use of rt_struct_factory for rt_struct instantiation.

```
template<typename T>
inline T& rt_struct::get
     (const rt_struct_member<T> &member)
{
   return reinterpret_cast<T&>
     (this[member.offset]);
}
```

Figure 6. Template function to access an *rt_struct* attribute member field.

rt_struct_member_info A member-info object contains meta data about a single attribute field or data member of a run-time struct, which includes its name, the size in bytes, and a flag describing if this data is input data, and if it is temporary or if it should be written to the output data stream.

rt_struct_info The rt_struct_info class contains information on all the dynamic attribute data fields in the form of rt_struct_member_info objects. It is used by rt_struct_user-based classes to register and query the set of member fields that are required for a certain processing pipeline, and by rt_struct_factory to compute the memory usage of the respective rt_struct.

rt_struct_user rt_struct_user is provided as a base class for classes that use rt_struct functionality. All stream and chain operators inherit from rt_struct_user. It provides the functionality to easily specify and reserve attribute fields and collects dependencies on fields of previous operators. All member fields that originate from an rt_struct_user-based class such as an operator are registered in the rt_struct_info object. The operator also registers itself with the rt_struct_factory during the setup process. This allows automatically setting the offsets for all rt_struct_members.

rt_struct_member A struct member object is used to access an attribute field or member variable of a stream point during the processing stage. It is a template object, with the template parameter being the type of the variable that rt_struct_member enables access to. It has to be initialized with the byte offset of that attribute variable in the allocated memory space that rt_struct occupies. This initialization is done automatically during setup by rt_struct_factory. An example of how to use an rt_struct_member object to access an attribute data field is given in Figure 7. Note that the compiler will issue a warning when automatic casting cannot be done safely (e.g. assigning a signed to an unsigned integer) or will report a compilation error when automatic casting cannot be done (e.g. when trying to assign a reference of the wrong type). This behavior is consistent with a programmers experience, and is the same as when using normal variables.

```
// member declaration in header:
rt_struct_member<float> _coeff;

// usage in source code implementation file:
void do_something(rt_struct* point)
{
   // get the value of the index member
   float &coeff = point->get(_coeff);

   // two possible ways to set a new value:
   coeff = 2.354;
   point->set(_coeff, 2.354);
}
```

Figure 7. Usage example on how to access dynamic member attribute fields.

rt_struct_factory The factory used in the rt_struct system is a variant of a pooling factory object. It is related to the abstract factory and factory method design patterns [9]. Differ-

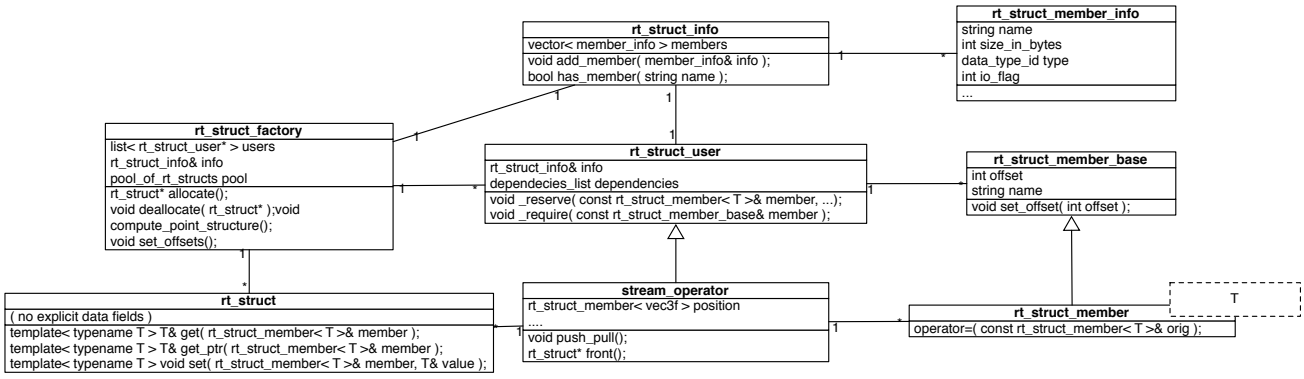

Figure 5. Dynamic *rt_struct* member elements class diagram.

ent stream-operator chain configurations dynamically define sets of data member variables used at run-time that represent a point and its attributes. Hence the different forms of real-time structures are not subclasses of a common base class representing the point data, but instead are simply of type rt_struct with dynamically allocated memory to hold the specified data member fields. The rt_struct_factory class provides the methods to allocate blocks of memory for rt_struct objects. For efficiency reasons, this is implemented using memory pooling, see Section 4.2. The factory class uses the rt_struct_info class to query the information about the attribute variable members. It computes and stores the offsets for each member variable, and sets them in the rt_struct_member instances during setup.

Setup Stage To initialize a stream-operator pipeline at runtime, a setup stage is carried out to allocate, initialize and configure the operators and to collect information on the required dynamic member fields.

In the first stage of the setup process, each stream operator reserves the member fields that originate in that operator by calling the _reserve() function of rt_struct_user. This will generate an appropriate rt_struct_member_info object, which will be registered in rt_struct_info. Each operator also calls the _require() function for attribute fields on which the operator depends on. This allows a basic dependency checking of member fields, and therefore of the operator order. The stream processing application will exit with an error message if the requirements of an operator cannot be fulfilled. Additionally, some configuration settings such as minimum number of neighbors can be negotiated between operators; e.g. if a fairing operator is set to require a minimum of 64 neighbors, this will override smaller values of other operators.

In the second stage, the factory computes the size of the stream-point objects, and the offsets to all of its attribute members. These are then stored in the rt_struct_member objects in the stream or chain operators.

Access Stage At run-time, the rt_struct data can easily be accessed using rt_struct_member objects. Since each rt_struct_member is templatized with the type of its variable this allows type-safe access to the stored variable at the cost of one (inlined) function call and a reinterpret_cast¡¿(). This works very efficiently and is additionally beneficial to the programmer: The types of all attribute fields are specified in the header of the respective stream-operator definition. Hence trying to access a field by using a wrong type is detected during compilation, and not only during run-time as it would be the case with the boost::any solution discussed in Section 4.1.1. Also,

Each rt_struct_member object is templatized with the variable type it represents, and the get-function of rt_struct is templatized with the rt_struct_member of the respective type as seen in Figure 7. This allows type-safe access to the data stored in the rt_struct object.

4.2. Memory Mangement

Pools of objects are used where possible to optimize performance by preventing continuous construction and destruction of objects. Currently, memory pools are used for rt_struct objects, for the *k*d-tree nodes in the *k*d-heap-neighbor operator and for all node types in the new chain operator.

5. Operators

The basic semantic of stream operators has been retained from [27] in the new proposed system architecture. In addition to the standard read and deferred-write I/O and the various geometry operators, we have implemented one new

```
// member declaration in header:
rt_struct_member<double> _radius;

// usage in source code implementation file:
void do_something(rt_struct* point)
{
   // works, since radius is of the same type
   // that _radius is templatized with.
   double &radius = point->get(_radius);

   // compiler will issue a warning
   // because of the potentially unsafe
   // implicit cast of the value.
   float radius = point->get(_radius);

   // compiler will exit with an error
   // because of the initialization of the
   // reference with an incorrect type.
   int &radius = point->get(_radius);
}
```

Figure 8. Valid and invalid access to member variables.

local stream operator $\Phi_O(\mathbf{p}_i)$ for outlier detection and removal.

Moreover, the new stream-processing system has an additional novel chain-operator type Ψ which in its scope overarches the entire chain of individual local stream operators Φ. A stream operator $\Phi_k(\mathbf{p}_i)$ is defined in [27] as a local operation on the geometry of point $\mathbf{p}_i$ and is but one element in a chain of stream operators $\Phi_1,\ldots,\Phi_p$ not knowing about the other selected operators. Furthermore, a stream operator only has direct access to the points within its own FIFO queue of points, which is only a subset of all points in the active set $\mathscr{A}$. On the other hand, the chain-operator $\Psi(\mathscr{A})$ in its scope spans the entire set of active points $\mathscr{A}$ and has knowledge of all elements in the chain of stream operators $\Phi_1,\ldots,\Phi_p$ as illustrated in Figure 9.

5.1. I/O Operators

The read operator Φ_R acts on the input stream of point data. During the setup phase, it reads and parses the data header and maps the point data input file to the input stream. Typically, this is done via memory mapping of the input file and sequential traversal through the input data. During the point processing phase, Φ_R reads the input point data and (optionally) converts it to the proper format. By definition, the read operator Φ_R must be the first in a chain of operators. It uses a pool of rt_struct point objects as mentioned in Section 4.2 to efficiently create the new objects.

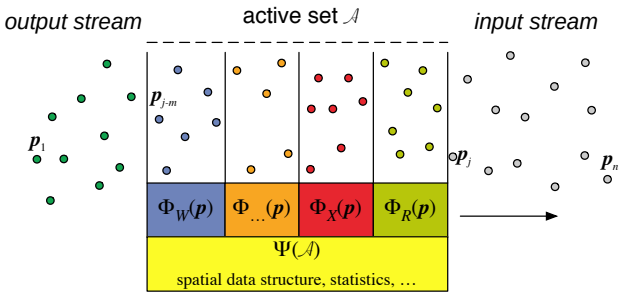

Figure 9. Conceptual diagram of the chain operator Ψ overarching a chain of individual stream operators Φ_k.

In the setup phase, the write operator Φ_W creates and memory maps the output file. During processing, the write operator uses the deferred writing strategy described in [27] to write points out to disk and remove them from main memory as soon as this can be done safely. Φ_W shares a pool of rt_struct point objects with the read operator, as indicated above, to avoid unnecessary memory allocation and deallocation overhead.

5.2. Neighborhood Operator

In the novel stream-processing framework we introduce a new neighborhood operator Φ_X that takes advantage of the spatial data structure provided by the new chain operator Ψ_X, see also below. Upon insertion of a new point $\mathbf{p}_j$ into the active-set $\mathscr{A}$, it will also be inserted into a spatial data structure maintained by Ψ_X. Moreover, the stream operator $\Phi_X(\mathbf{p}_j)$ then queries $\Psi_X(\mathscr{A})$ immediately for an initial *left-sided* nearest neighbors set $\mathscr{N}_j$ which at that point contains the closest points $\mathbf{p}_i$ with index $i < j$ in the input stream. At the same time, the new point $\mathbf{p}_j$ is tested with existing neighborhood sets $\mathscr{N}_i$ and included if appropriate to continuously complete their missing *right-sided* closest neighbors.

The spatial data structure in the chain operator Ψ_X also allows for within-range neighborhood queries. As soon as the next new point $\mathbf{p}_j$ is farther away from an active point $\mathbf{p}_i \in \mathscr{A}$ than the query range r along the streaming dimension, a range query for $\mathbf{p}_i$ can be invoked on $\mathscr{A}$ to find all neighbors within distance r.

While the new nearest neighbor operator is now the default to establish closest points neighborhoods, the old kd-heap approximate k-nearest-neighbor operator used in the original stream processing application [27] is still available, but is deprecated for performance reasons. See experimental results in Section 7.

5.3. Geometry Operators

The normal estimation Φ_N, curvature estimation Φ_C, elliptical point splat-extent estimation Φ_E and smoothing Φ_S operators have been ported to the new dynamic rt_struct member attributes architecture described in Section 4, but are otherwise identical in functionality to the description given in [27].

A new local outlier detection stream operator $\Phi_O(\mathbf{p}_i)$ has been added which quantifies the likeliness of any nearest neighbor $\mathbf{p}_j \in \mathscr{N}_i$ to be an outlier. The outlier quantifier is defined as

$$O_i(\mathbf{p}_j) = h_i(|\mathbf{p}_i - \mathbf{p}_j|) \cdot |\mathbf{p}_j - \Pi_i(\mathbf{p}_j)|, \qquad (1)$$

with $h_i(x)$ being a smooth weighting function, e.g. a Gaussian, and $\Pi_i(\mathbf{p}_j)$ the projection of $\mathbf{p}_j$ onto the plane with normal $\mathbf{n}_i$ through point $\mathbf{p}_i$. The outlier operator $\Phi_O(\mathbf{p}_i)$ compares the values $O_i(\mathbf{p}_j)$ for all $\mathbf{p}_j \in \mathscr{N}_i$ to a threshold value ε and if greater discards $\mathbf{p}_j$ as a nearest neighbor of $\mathbf{p}_i$.

Therefore, the outlier operator $\Phi_O(\mathbf{p}_i)$ measures the offset of $\mathbf{p}_j$ from the tangent plane at $\mathbf{p}_i$ and quantifies a degree of co-planarity. However, it penalizes points $\mathbf{p}_j$ farther away from $\mathbf{p}_i$ less, so they are allowed to deviate more from the tangent plane. Hence with the support radius of the weighting function $h_i(x)$ being adjusted relative to an estimated local curvature at point $\mathbf{p}_i$, the outlier detection can be made adaptive to the local surface feature size.

5.4. Chain Operators

The first new chain operator $\Psi_X(\mathscr{A})$ sorts all the points in the active set $\mathscr{A}$ into a spatial tree structure. By default, a bucketed PR KD-Tree [38] is used, but the operator can be templatized with other tree structures. Any regular stream operator can interact with this tree operator by providing a visitor object [9]. Currently, the tree operator is used by the neighborhood operator Φ_X for efficient *k*-nearest-neighbor searching or range queries. Additionally, the smoothing operator Φ_S uses the new chain operator Ψ_X to update the spatial data structure for the modified point locations.

The statistics operator $\Psi_S(\mathscr{A})$ collects data about current and maximum data size, extent and memory usage of the active set. Having all statistical functionality in an operator makes data collection optional, and statistics can easily be disabled for performance reasons. This also allows the implementation of different operators to customize statistics collection (complete debug and optimization statistics vs. minimal release-mode statistics).

6. Implementation

Setup To demonstrate the run-time procedure of the stream-processing application, a very simple processing job will be shown in detail. A raw point data set should be processed to compute the normal of each point. The resulting pipeline consists of the following stream operators: read, neighborhood, normal and write. The input data set contains only the point positions.

During the setup phase, the read operator will be initialized first, reads and parses the input point header and registers a dynamic member field called position in the rt_struct_info object. Next, the neighboorhood operator is initialized. This operator requires the chain tree operator and so requests the system to instantiate the that chain operator. Then, it will reserve the neighbor-list and neighbor-count dynamic member fields in rt_struct_info, and adds the position to the input-requirements list in the rt_struct_user base-class. Additionally, it will check for a user-specified neighborhood-size option, and set the default if none was given. Next the normal operator is initialized, reserves the normal vector dynamic member field and adds position, neighbor-list and neighbor-count to the input-dependency-list. Finally, the write operator is instantiated. Optionally, a statistics chain operator might be set up if specified by the user. After all the operators are set up, the system will run a dependency check that makes sure all the input requirements for each operator are met. The pipeline specified above is valid and therefore passes that check.

The factory now computes the size of the rt_struct used in this pipeline, sets the offsets to the dynamic members in all rt_struct_member objects of each stream operator and allocates an initial pool of memory for points, that is for the rt_struct objects. The read operator memory-maps the input data file, and the write operator creates and memory-maps a file large enough to contain the final point data, including all the newly created member fields that were marked as persistent output data fields. This phase concludes the setup stage and stream processing can now begin.

Processing The read operator starts reading the point position from the input data file into factory-allocated new rt_struct objects. The points are inserted into all chain operators, including the tree operator containing a spatial data structure. The neighborhood operator then gets the points and will perform a *k*-nearest neighbor query by sending a visitor object to the tree operator's data structure. See Section 5.2 for how the *k*-nearest neighbor search is performed. The normal operator then receives the point from the output buffer of the previous operator and computes an estimated normal based on the points neighbors. Finally, the deferred write operator buffers the processed point until it is not referenced anymore from any of the previous stream operators. Finally, the point-data is written to the output stream and the rt_struct object is returned to the factory. This process continues until all the input points have been processed.

Notes The new stream processing system depends on two libraries: boost [1] and some vector-matrix geometry functions. The memory pooling and program options libraries from boost are used. The stream processing system runs on most UNIX-based operating systems including Mac OS X, GNU/Linux and FreeBSD. It has been organized into a dynamic library. This allows access to the functionality not only by using the included command line tool, but also allows any application to link against it, e.g. to provide a graphical user interface. Additionally, a php-based web framework has been developed that provides access to the stream processor from a web browser, and supports http or bittorrent uploads and downloads of source and target data files. Figure 10 demonstrates how this interface looks like.

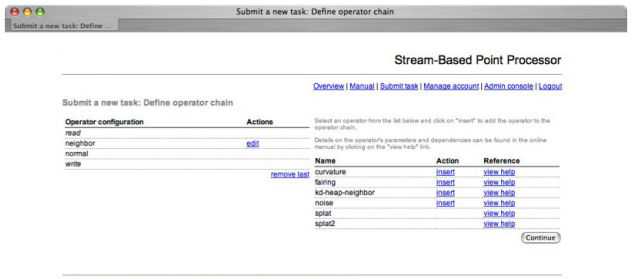

Figure 10. Web interface to the stream-processing framework.

7. Results

All experimental results reported here were achieved on a PowerMac Pro with dual Intel Xeon 2.6GHz processors. The currently single-threaded implementation is executed on a single CPU core. The models that have been tested include: David head (2,000,646 points), David 2mm (4,129,534 points), David 1mm (28,168,109 points) and Lucy (14,022,961 points).

In Figure 11 we compare the performance of the new stream-processing system architecture to the original approach introduced in [27]. As we can see, the new architecture is not only much more flexible with its run-time configurability of the stream-operator chain, but it is also significantly more efficient for large point cloud data sets. In particular, the larger the nearest neighborhood set is defined the larger is the performance improvement over the previous approach.

Significant contributions to the performance improvements come not only from the optimized implementation and memory pooling, but also from the new neighborhood stream operator. As Figure 12 shows, the relative time used by the neighborhood operator has decreased, in particular for the largest David 1mm model.

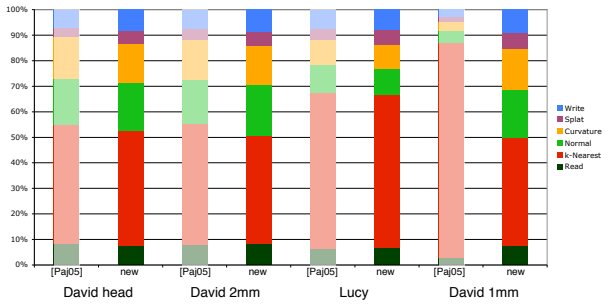

Figure 12. Times spent in the different stream processing operators relative to the overall processing time, in comparison to [27].

Applying a chain of operators consisting of read (Φ_R), nearest-neighbor search (Φ_X), normal estimation (Φ_N), curvature estimation (Φ_C), splat-extent estimation (Φ_E) and deferred-write (Φ_W) the out-of-core effectiveness of the stream-processing system has remained equivalent to [27]. As shown in Figure 13, the core goal of dramatically reducing the number of data elements actively maintained in main memory has well been achieved, as rarely ever more than 1% of data is kept active in main memory.

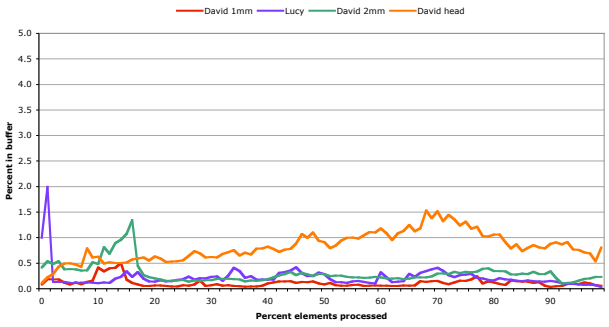

Figure 13. Percentage of active set of points maintained in main memory during stream-processing.

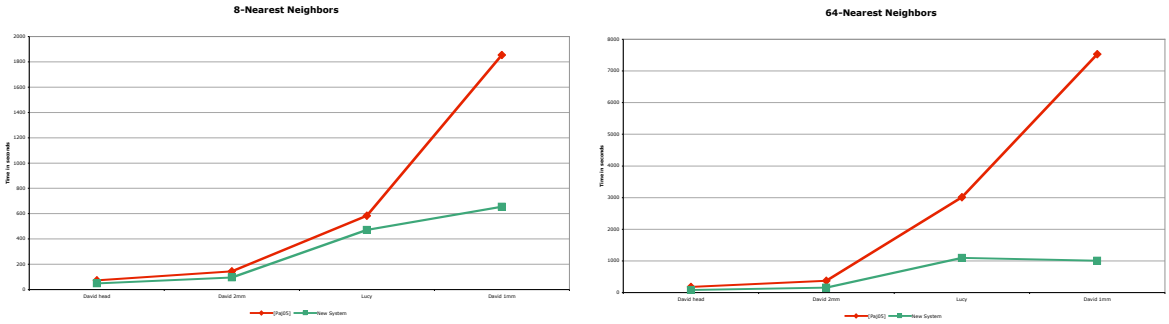

Figure 11. Execution times of a stream-operator chain using 8-, and 64-nearest neighbors.

8. Conclusion

In this paper we have presented a novel stream processing architecture, extending and implementing the conceptual framework introduced in [27] more efficiently. The new architecture allows for efficient and flexible run-time configurability of geometry processing operators that can be applied to an ordered stream of point cloud data. This novel definition and implementation of local stream-processing operators allows operators to be dynamically defined and configured at run-time, and not statically at compile-time as previously required. Through our novel stream-operator implementation, the main stream-processing application program can be compiled without specification of which geometric operators and in what order they will eventually be applied to the point data. In fact, at run-time, the available local geometry processing operators can dynamically be selected and configured on-demand. Moreover, the stream-processing application can automatically check for consistency of the selected chain of stream operators.

Additionally, in the context of stream-processing points by passing them from one geometry processing operator to the next in a chain of multiple successive stream operators, we have introduced the new concept of a chain operator which acts as a global operator overarching the chain of individual stream operators.

Finally, the new system architecture has also shown significant performance improvements.

Acknowledgements

We would like to thank and acknowledge the Stanford 3D Scanning Repository and Cyberware Inc. for providing the 3D geometric test data sets. This work was partially supported by the Swiss National Science Foundation Grant 200021-111746/1.

References

[1] Boost c++ libraries. Website, 2007.

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

[3] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings Eurographics Workshop on Rendering*, pages 53–64, 2002.

[4] L. Cheng, A. Bhushan, R. Pajarola, and M. El Zarki. Real-time 3D graphics streaming using MPEG-4. In *Proceedings IEEE/ACM Workshop on Broadband Wireless Services and Applications*, 2004.

[5] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22(3):657–662, 2003.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

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

[8] K. Engel, O. Sommer, and T. Ertl. A framework for interactive hardware-accelerated remote 3D-visualization. In *Proceedings EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, pages 167–177, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[10] E. Gobbetti and F. Marton. Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(1):815–826, February 2004.

[11] M. H. Gross. Getting to the point...? *IEEE Computer Graphics and Applications*, 26(5):96–99, September-October 2006.

[12] M. H. Gross and H. Pfister, editors. *Point-Based Graphics*. Morgan Kaufmann Publishers - Elsevier, 2007.

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

[14] K. Henney. Valued conversions. *C++ Report*, 12(7):37–40, July-August 2000.

[15] M. Isenburg and P. Lindstrom. Streaming meshes. In *Proceedings IEEE Visualization*, pages 231–238, 2005.

[16] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *Proceedings IEEE Visualization*, pages 465–472. Computer Society Press, 2003.

[17] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *Proceedings Symposium on Geometry Processing*, pages –. Eurographics, 2005.

[18] T. R. Jones, F. Durand, and M. Zwicker. Normal improvement for point rendering. *IEEE Computer Graphics and Applications*, 24(4):53–56, July-August 2004.

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

[20] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[21] M. Levoy and T. 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.

[22] J. P. Linderman. rsort and fixcut. man pages, 1996. revised June 2000.

[23] 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.

[24] G. 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.

[25] Y. Noimark and D. Cohen-Or. Streaming scenes to MPEG-4 video-enabled devices. *IEEE Computer Graphics and Applications*, 23(1):58–64, January/February 2003.

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

[27] R. Pajarola. Stream-processing points. In *Proceedings IEEE Visualization*, pages 239–246. Computer Society Press, 2005.

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

[29] R. Pajarola, M. Sainz, and R. Lario. XSplat: External memory multiresolution point visualization. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing (VIIP)*, pages 628–633, 2005.

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

[31] M. Pauly, R. Keiser, L. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics*, 22(3):641–650, 2003.

[32] H. Pfister and M. Gross. Point-based computer graphics. *IEEE Computer Graphics and Applications*, 24(4):22–23, July-August 2004.

[33] L. Ren, H. Pfister, and M. 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).

[34] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings ACM SIGGRAPH*, pages 343–352. ACM SIGGRAPH, 2000.

[35] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proceedings Symposium on Interactive 3D Graphics*, pages 63–68. ACM SIGGRAPH, 2001.

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

[37] M. Sainz, R. Pajarola, and R. Lario. Points reloaded: Point-based rendering revisited. In *Proceedings Symposium on Point-Based Graphics*, pages 121–128. Eurographics/IEEE VGTC, 2004.

[38] H. Samet. *Foundations of Multidimensiona and Metric Data Structures*. Morgan Kaufmann Publishers - Elsevier, 2006.

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

[40] H. T. Vo, S. P. Callahan, P. Lindstrom, V. Pascucci, and C. T. Silva. Streaming simplification of tetrahedral meshes. *IEEE Transaction on Visualization and Computer Graphics*, 13(1):145–155, January-February 2007.

[41] 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/IEEE VGTC, 2004.

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

[43] Y. Zhang and R. Pajarola. Deferred blending: Image composition for single-pass point rendering. *Computers & Graphics*, 31(2):175—189, 2007.