Streaming Extraction of Elevation Contours from LIDAR Points

Martin Isenburg[1] Yuanxin Liu[2] Jack Snoeyink[2]

[1]Computer Science Division
UC Berkeley

[2]Computer Science
UNC Chapel Hill

Abstract

Air-borne laser range scanning technology (LIDAR) is able to quickly generate massive amounts of densely spaced points that sample the elevation of a terrain. We describe a streaming technique that is able to extract high-resolution iso-contour elevation lines from such large data sets using significantly less memory than the processed input points and the produced output lines. Optional on-the-fly simplification can makes the produced iso-contours more tractable for immediate visualization and smooth out scanning error. By keeping point, mesh, and line data in streaming formats containing "finalization tags," we connect individual batch operations into a streaming processing pipeline that instantly begins producing final iso-contour output.

1. Introduction

The "pipes" concept, originated by Doug McIlroy[1] in the 1972 AT&T Unix v.3, allowed users to effortlessly combine simple tools operating on a data stream, with the operating system time-slicing between the processes in a pipe. Some tools (notably `grep`, `awk`, `sed`) allowed processing on-the-fly, while others (e.g. `sort`) were batch operations that would read their whole input before producing output.

Since geometric data sets (e.g., point clouds, polygon soups, and meshes) are difficult to characterize and represent, most tools operating on them are either systems that convert geometric data into their own format for the operations that they support, or batch processing tools, like `sort`. (Of course, since no single system provides all operations that a user wants to apply, the large systems are also used as batch processes that write output data to temporary files instead of to pipes.)

Recently we have demonstrated that the concepts of pipes and of stream processing can be brought to geometric data by extending the representations in which geometry is input and output with the concept of *finalization*, which documents in the data format when a data object will no longer be needed. In this paper, we bring together work on *spatial finalization* for point clouds [1, 2] and *topological finalization* for meshes [3, 4] and show that these support the best type of pipelined processing and visualization: the results can begin to be displayed well before the input data has been read, which would valuable even if its only benefit was to give a user an opportunity to abort the computation after a few seconds if the visualization is not quite what was desired. But developers also benefit from the elegance of pipes and filters by being able to focus on the computation while the system takes care of time-slicing.

We consider the task of constructing topographic maps by converting LIDAR (air-borne laser range data, which returns scattered points with x, y, and z coordinates) into contours (isoheight lines) via a TIN (Triangular Irregular Network [5], a common terrain representation for scattered data). Our pipeline for this task works as follows (see Figure 1): while `spfinalize` adds finalization tags to a stream of points, module `sp2delaunay` [1] creates a TIN as a streaming Delaunay triangulation. As triangles are produced, they are passed to `smsmooth` for Laplacian smoothing of z coordinates and `smsimp` for mesh simplification using a quadric error metric [6]; these modules may be used multiple times for additional smoothing or simplification. Module `tin2iso` extracts specified elevation contours as streaming polygons and polylines from the stream of triangles of the smoothed and simplified mesh, and `slclean` filters out small contour lines (based on the length, the number of segments, or size of bounding box), piping the rest to a file or to the `sl_viewer` for optional line simplification and interactive display. The only batch process in this pipeline is at the front: for the point set we need a bounding box partitioned into a grid, with the count of how many points are in each grid cell. If this information comes from the point producer, then all stages of the pipeline can be active simultaneously, and the user can begin to see output when a small percentage of the data has been read.

[1]From the Software Tools Users Group 2004 STUG award to Doug McIlroy: "The one thing, however, that most people think about when they think of UNIX is the power of the command line interface and the elegance of the pipe and filter model. . . . Doug McIlroy helped develop the concept of pipes and stream processing. In order to demonstrate the concept of stream programming, he wrote the original UNIX version of such tools as sort(1), spell(1), diff(1), join(1), graph(1), speak(1), and tr(1), among others." `http://www.usenix.org/about/stug.html`

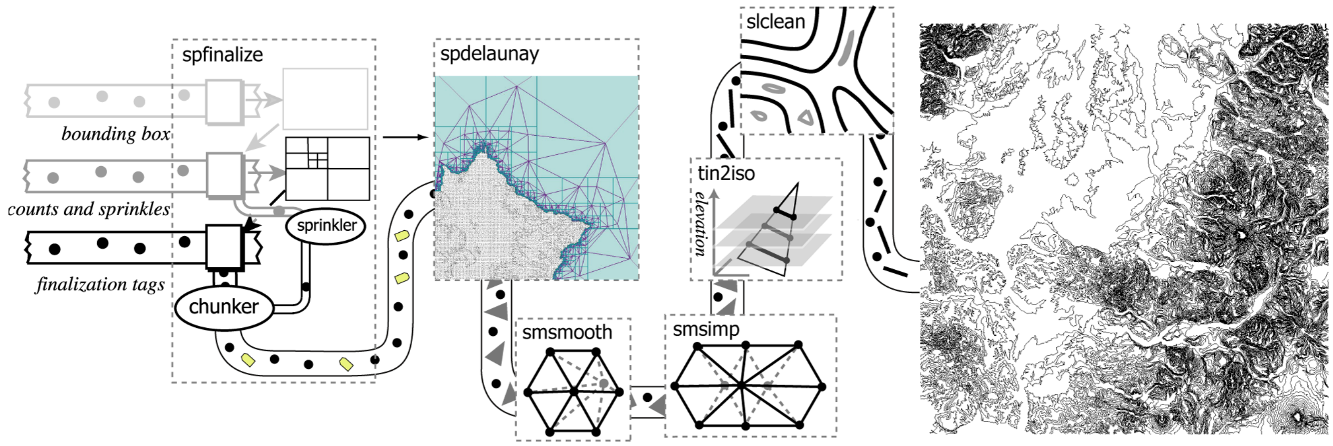

Figure 1: An example processing pipeline realized by piping our processing modules together. Spatially finalized points stream out of `spfinalize` into `spdelaunay` where a TIN is computed with streaming Delaunay computation [1] and streamed onwards in streaming mesh format to `smsmooth` where elevation coordinates are smoothed and on to `smsimp` which simplifies down to a specified fraction. The simplified TIN streams to `tin2iso` where iso-contours are extracted at specified elevations and streamed out in a streaming line format to `slcean` where small polylines are removed. The final lines can be streamed to disk or—for inspection—streamed or through a viewer that renders the lines as they arrive while on-the-fly creating a coarser version in memory solely for interactive rendering.

The benefits of a streaming pipeline for processing large amounts of geometry data are

composability Streaming implementations give us the flexibility to "plug and play"—to string different modules together to solve complex tasks with high throughput.

low latency Finalization allows output from the last module in the string while the first is still reading its input, giving the user rapid feedback from the whole composition.

memory coherence As data is finalized its memory may be released, keeping the memory footprint of each module small so that they need not compete for memory resources on the system.

parallelism Pipes can harness the computing power of an additional processor or multi-core chips without any programming effort; the operating system can take care of the load balancing.

modularity Modules can be developed first as simple prototypes, as ours have been, and later replaced with a "better" techniques, as ours will be. (e.g., the Delaunay triangulator and mesh smoothing operators should respect breaklines)

2. Related Work

There are uncountably many systems that prepare or visualize terrain data in general. The Army Corps of Engineers maintains a survey with links to over 500 commercial packages[2]. And few of these handle LIDAR data, notably QT Modeler from Applied Imagery[3] who use a quadtree [7] to handle hundreds of millions of points within their system without loading them all into memory. Each of these packages provides functionality that works within its own system, so it is not uncommon to have to go from system to system to compose functions, storing intermediate data in files.

As mentioned in the introduction, pipes were introduced in AT&T Unix version 3, "at the suggestion (or perhaps insistence) of M. D. McIlroy, a long-time advocate of the non-hierarchical control flow that characterizes coroutines." [8] Pipes are crucial plumbing in modular visualization systems, such as such as the former SGI Explorer [9] or current AVS[4]. Our contribution is to identify finalization as a way to pipe less-structured data through concurrent coroutines without the support of a large system.

The processing we do is typical of a LIDAR processing pipeline; what is novel is the ability to see the results before any of the steps of the pipeline completes.

Bare-earth extraction: LIDAR sensors capture elevation points from the ground (bare-earth data) as well as from objects on the ground, such as forests and buildings. For contouring and many other applications, it is desirable to use bare-earth data only. However, extracting the bare-earth data is not simple and often involves time consuming manual processing. There have been much research on automatic methods to extract the bare-earth models [10, 11]. Many of these methods filter data points based on local geometry of the points or of a TIN and can be adapted to work in the

[2]`http://www.tec.army.mil/research/products/TD/tvd/survey`

[3]`http://www.appliedimagery.com`

[4]`http://www.avs.com`

streaming pipeline.

Contour smoothing: The contours drawn on a topographic map to depict a natural terrain are expected to be smooth curves. Unfortunately, the contours produced from a TIN are often either jagged or noisy: jagged when contours are computed from a TIN with resolution too coarse, so that the polygonal nature of the contours become visible, and noisy when the TIN is constructed from dense data such as LIDAR that is either capturing featurs that are smaller than the resolution of interest or that is noisy itself.

Contours are best smoothed by smoothing a surface, and then generating contours from the surface. This guarantees that there are no crossing contours, and that they convey a consistent picture of the underlying topography. It is computationally easier to smooth contour lines individually, which we support in our visualizer, but surprisingly difficult to smooth a group of contour lines while guaranteeing no intersections; Estkowski and Mitchell showed that some versions of this problem are NP hard and hard to approximate [12].

Visualization and applications: Once the TIN has been prepared from the point data there are many applications in addition to visualization. Contour lines may also be analyzed for flood level prediction, or volume under the surface may be computed for beach erosion analysis. The surface may be converted into a raster DEM for further processing.

3. Streaming Formats and Modules

Streaming formats seek to present data in an order that allows stream processing modules to perform computation as the data arrives. The most common streaming formats are for one-dimensional time series, such as streaming audio or video. To decompress or filter such time series, the data in a small sliding window is buffered in a FIFO queue. The window size is chosen so that the operation has all the local information that it needs. This allows data to stream out as fast as it streams in, with only a little latency.

For geometric data, including triangle meshes, line segments, or point clouds, there are many operations that depend only on the data in a local neighborhood. Unfortunately, there is no "natural" order of the data for which a small sliding window on a one-dimensional stream will contain all local information that a non-trivial operation needs. Instead, we add explicit "finalization tags" to our streaming input and output data formats; the tag for a data item lets a streaming module know that it has all the data required to perform a computation for that item.

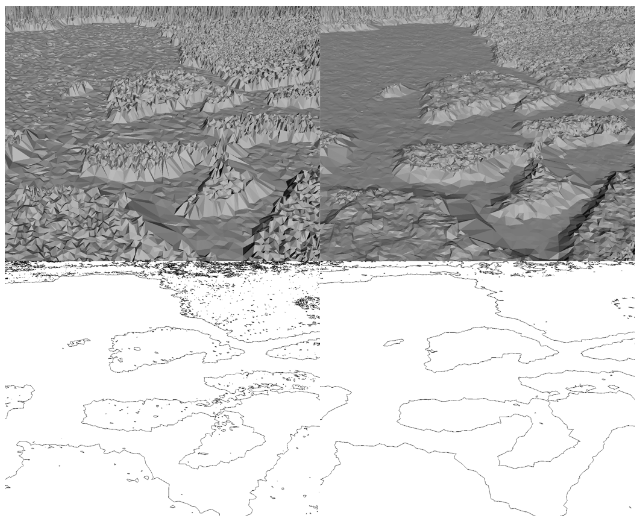
Figure 2: A close-up of a TIN from the raw LIDAR points of "noaa" before (left) and after (right) piping it through `smsmooth`, along with the corresponding one-foot contour lines extracted by `tin2iso`. The "noaa" data is low relief data from a program that monitors coastal areas. Elevations are scaled by 32 for illustration.

3.1. Streaming Meshes, Points, and Lines

Isenburg and Lindstrom [4] describe a streaming format for meshes: vertices and triangles are intermixed, along with *vertex finalization tags* that indicate when all triangles referencing this vertex have already appeared in the stream. This *topological finalization* of mesh vertices allows modules to derive when, for example, the one-ring or two-ring neighborhood around a vertex or an edge has completely appeared in the stream, and perform tasks that need the data from such neighborhoods (e.g. computing vertex normals). Finalization of vertex v tells the application that it can complete all computations that were waiting for v's topology, output partial results, and safely free any data structures that are no longer needed.

In our pipeline we produce topologically finalized TINs as output of our streaming Delaunay triangulator so that we can simplify the TINs with streaming edge-collapse operations [13, 14]. We also use topologically finalized TIN input to extract elevation contours and output them in a streaming line format that includes vertices, line segments, and vertex finalization tags. Having topologically finalized lines can in turn be exploited for streaming topological clean-up that removes connected components smaller than some threshold.

To perform streaming operations on point data requires a different type of finalization. Pajarola [2] sorts the points into a global spatial order that allows him to derive what we could call *k-neighbor finalization*: a point is finalized after its k nearest neighbors have arrived. We, in a paper that is sent as supplemental material [1], use *spatial finalization*: space is partitioned into regions, and a region is finalized

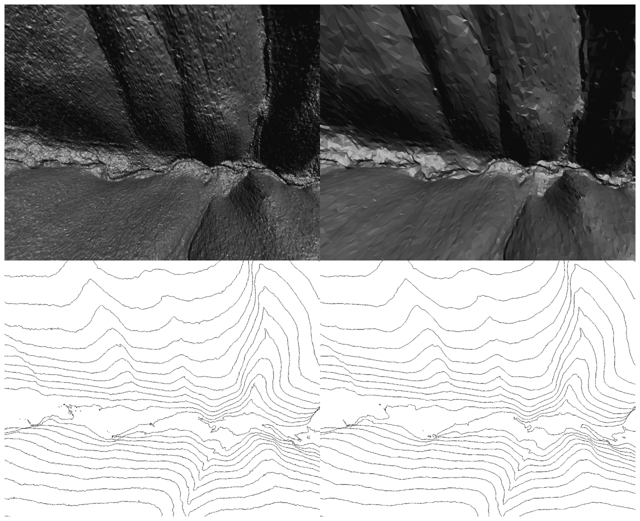
Figure 3: A close-up of a TIN from the raw LIDAR points of "grbm" before (left) and after (right) piping it through `smsimp`, which simplifies down to 10 percent. The corresponding two-foot contour lines extracted by `tin2iso` demonstrate the achieved quality in line simplification. Elevations are scaled by 16.

after the last point in the region appears in the stream.

In our pipeline we need spatially finalized points as input to our streaming Delaunay triangulator. In the ideal case the input format from which we start processing already has *spatial finalization tags*. Otherwise compute this information with two additional passes over the points or (for highly incoherent input) with spatial sort as outlined in [1].

3.2. Stream Processing Modules

We have implemented a number of stream processing modules that operate on points, meshes, and lines, many of which are included in the software that accompanies this paper. They are described here briefly:

`sp2sp,sm2sm,sl2sl` convert format of streaming points, meshes, and lines between ascii, raw binary, and compressed binary representations.

`spreduce` decimates points by random selection.

`spfinalize` computes spatial finalization tags for raw points in three read passes [1]. Consumes raw points and produces streaming points.

`spdelaunay` converts streaming points into a streaming mesh using streaming 2D Delaunay triangulation [1].

`smsmooth` performs Laplacian smoothing on the z coordinate of a TIN. Consumes a streaming mesh and produces a streaming mesh (see Figure 2).

`smsimp` simplifies a TIN down to a chosen fraction with the simplification technique of [14]. Specialized towards TINs which allow more efficient topology checks of edge-collapse validity (see Figure 3).

`tin2iso` extracts elevation contours (iso-height lines) for specified elevations with linear interpolation. Consumes a streaming mesh, produces streaming lines.

`slclean` removes short polylines that have few segments or short total length. Consumes streaming lines produces streaming lines. Buffers only those polylines that are below the threshold but still active.

`sp_viewer,sm_viewer,sl_viewer` render streaming points, meshes, and lines for quick inspection. Unoptimized OpenGL based viewers. Simplify input on-the-fly using vertex clusterting to retain interactivity for larger inputs. Support out-of-core rendering of a the chosen view-point at full resolution.

4. Performance Measurements

In Table 1 we report data production, timings, and main memory consumption for various configurations of our stream modules into a geometry processing pipeline. We always start from a binary file containing raw points in single-precision floating point format on one disk and end with the respectively produced data (e.g. streaming points, meshes, lines) on the other disk. Measurements are taken on a Dell Inspiron 6000D laptop with a 2.13 GHz mobile Pentium processor and 1 GB of memory, running Windows XP. The raw point file is read from an external LaCie 5,400 RPM firewire drive and the data produced by the last module is written to the 5,400 RPM local harddisk.

To demonstrate the minimal end to end delay in our streaming pipeline we report the time that it takes until the first result appear at the end of the pipe once the raw input points are finalized. Computing spatial finalization takes an additional 2+2 seconds for the smaller "grbm" data set and 27+27 seconds for the larger "puget" data set as `spfinalize` needs to make two strictly I/O limited passes over the points before it can start producing finalized output points. If the raw points are already stored in a finalized format on disk this preprocessing is not necessary and our pipe starts producing output almost instantly.

The total main memory use of our pipe is well below the availability of memory resources on household PC. The biggest memory hog is `spfinalize` that buffers point to stream them out in "chunks". That processing the 10 times larger inputs and outputs of "puget" uses only slighly more memory than that of "grbm" demonstrated the scalability of our pipeline.

While there is plenty of memory for adding more modules to the pipe, this slows down the pipe as it adds another consumer of CPU cycles. The slowest component of our pipe is `smsimp` which consumes most of the CPU time. As the slowest module dictates the speed for the whole pipe, adding a second CPU can only increase the throughput to

“grbm” (6,016,833 points, 69 MB)		number of produced		size	time (sec)			memory (MB)		
added module	produced data	objects	(comp)	MB	prep	first	last	other	cmd	total
`spfinalize`	finalized LIDAR	6,016,833		71	2+2	1	3	-	28	28
`spdelaunay2d`	raw TIN	12,018,597		213	2+2	1	27	28	7	35
`tin2iso`	raw lines	1,236,155	7,127	29	2+2	1	35	35	6	41
`slclean`	clean lines	1,180,364	228	28	2+2	1	36	41	1	42
`smsmooth`	smooth TIN	12,018,597		213	2+2	2	43	42	6	48
`smsimp`	smooth simpl TIN	1,201,860		21	2+2	15	170	44	15	59
	smooth simpl lines	398,694	608	10						
	smooth simpl clean lines	390,357	145	9						

“puget” (67,125,109 points, 568 MB)		number of produced		size	time (sec)			memory (MB)		
added module	produced data	objects	(comp)	MB	prep	first	last	other	cmd	total
`spfinalize`	finalized LIDAR	67,125,109		776	27+27	1	42	-	21	21
`spdelaunay2d`	raw TIN	134,207,228		2328	27+27	1	426	21	10	31
`tin2iso`	raw lines	8,861,024	8,434	205	27+27	1	469	31	11	42
`slclean`	clean lines	8,697,263	3,382	201	27+27	1	475	42	1	43
`smsmooth`	smooth TIN	134,207,228		2328	27+27	2	547	43	12	55
`smsimp`	simpl TIN	13,420,723		232	27+27	28	2118	50	16	66
	smooth simpl lines	2,853,504	7,894	65						
	smooth simpl clean lines	2,708,640	1,796	63						

```
spfinalize -i points.raw | spdelaunay2d -ispb -osmb | smsmooth -ismb -osmb |
    smsimp -ismb -osmb | tin2iso -ismb -oslb | slclean -islb -o lines.slb
```

Table 1: Performance of streaming pipeline when adding more stream modules, for two data sets, “grbm” and “puget”. As you read down the tables, stream modules are added to the pipe one by one, cumulating in the above pipeline. Each line in the table reports the additional time, amount of produced data and memory use for the newly added module. Time is broken down into preprocessing time, and from then the time until the first result, and until the last result streams out at the end.

that of `smsimp`, which can simplify around 100 thousand triangles per second. One possibility to better leverage multiple CPUs offered by our modular approach is a pipe of multiple `smsimp` modules together that each simplify less.

Even on one CPU, in less than 35 minutes our pipe has turned over 700 MB of point data into 63 MB of simplified iso-contour elevation lines. Along the way—to create intersection free contours—it creates a temporary TIN, uses its connectivity to smooth elevations among neighboring points, and then simplifies it to 10 percent. Neither the raw nor the smoothed and simplified TIN need every to be stored as they stream from one module to the other. As all described modules and the “grbm” data set are included with the paper, we encourage the reviewers to verify the reported performance or even build their own pipe on their own data.

5. Conclusions

We have shown that streaming formats for geometric data support the incremental development of a suite of simple streaming modules that handle complex processing tasks efficiently, are easy to work with, and allow for immediate inspection in an interactive workflow.

Note that streaming is useful even if your input and outputs fit into memory. In an interactive workflow streaming can drastically reduce the time from loading the points from file until “seeing” the first extracted isoline. When exploring the data one instantly notices when the pipe is running with the wrong parameters, and can stop the computation without having to wait for its completion to rerun again. In other words, we obtain confirmation within seconds that the result we are creating is what we actually wanted … only then do we commit to letting the pipe run through.

As a contour production tool, our prototype pipeline needs a few improvements. The contours on a high quality topographic map can quickly convey the three dimensional shape of a terrain to a human reader. However, the output from our contour production pipeline may contain visual artifacts, due to the “noise” in the data and the simplicity of our interpolation and contour extraction algorithms. Fortunately, the modularity of pipeline processes supports incremental development by easy substitution of new algorithms.

One of the immediate needs is what the geographers call “hydrological enforcement.” Terrain models are often used to analyze drainage characteristics of a landscape; anyone who reads topographic maps knows the that a common visual cue for a river valley is a sequence of sharp “V”-shaped contours, with the pointed end of the “V” pointing upstream. Unfortunately, the TIN construction sometimes puts a “dam” in a river valley, and subsequent contours can

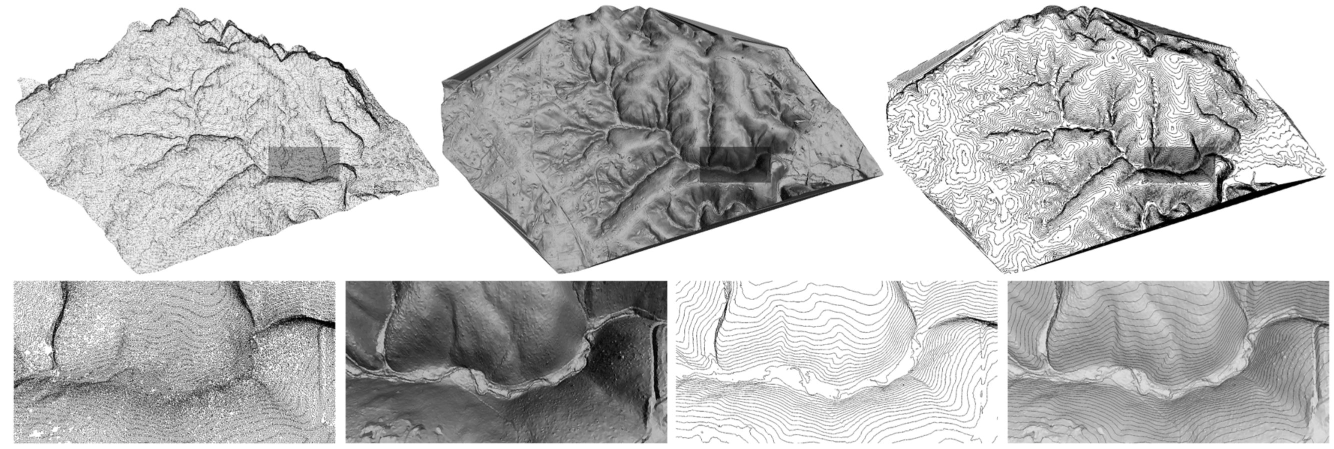

Figure 4: Our stream processing pipeline operating on the "grbm" data set. The original 6 million LIDAR points (left) are spatially finalized, Delaunay triangulated, smoothed, and simplified into a TIN of 1.2 million triangles (middle) from which we extract 2 feet elevation contours and output them into a file while removing poly lines shorter than 10 feet.

look blatantly incorrect [15]. Traditional photogrammetric techniques avoid such mistakes because the (human) contour generator looks at the photographs of a terrain. To automatically generate hydrologically correct contours, a TIN construction algorithm must respect hydrological enforcements so that it does not generate terrains that obstruct water flow.

The easiest way to do this is to add additional data for river and ridge lines, known as breaklines, into the construction and smoothing of the TIN. Thus, we plan to extend our streaming Delaunay triangulator [1] to accept hydrological breaklines as input and produce constrained Delaunay triangulations and to modify the smoothing operators to respect breaklines in the TIN.

We hope that the efficiency of our streaming pipeline has presented a compelling case that it is worthwhile to develop new algorithms with streaming in mind.

References

[1] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink, "Streaming computation of Delaunay triangulations," Currently Under Review, Preprint available at `http://www.cs.unc.edu/~isenburg/scdt.pdf`, 2006.

[2] R. Pajarola, "Stream-processing points," in *Visualization'05 Proceedings*, 2005, pp. 239–246.

[3] M. Isenburg and S. Gumhold, "Out-of-core compression for gigantic polygon meshes," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 935–942, July 2003.

[4] M. Isenburg and P. Lindstrom, "Streaming meshes," in *Visualization '05 Proceedings*, 2005, pp. 231–238.

[5] T. K. Peucker, R. J. Fowler, J. J. Little, and D. M. Mark, "The triangulated irregular network," in *Amer. Soc. Photogrammetry Proc. Digital Terrain Models Symposium*, 1978, pp. 516–532.

[6] M. Garland and P. Heckbert, "Surface simplification using quadric error metrics," in *SIGGRAPH 97 Proceedings*, 1997, pp. 209–216.

[7] Hanan Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, 1989.

[8] Dennis M. Ritchie, "The evolution of the unix time-sharing system," *AT&T Bell Labs Technical Journal*, vol. 63, no. 6, pp. 1577–93, Oct. 1984, Part 2. Reprint of paper at Language Design and Programming Meth. conf., Sydney, Sept 1979. `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`.

[9] D. Foulser, "IRIS Explorer: A framework for investigation," *Computer Graphics*, pp. 12–16, May 1995.

[10] R. A. Haugerud and D. J. Harding, "Some algorithms for virtual deforestation (VDF) of LIDAR topographic survey data," *International Archives of Photogrammetry and Remote Sensing*, vol. XXXIV-3, pp. 22–24, 2001.

[11] G. Sithole and G. Vosselman, "Experimental comparison of filter algorithms for bare earth extraction from airborne laser scanning point clouds," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 59, pp. 85–101, 2004.

[12] Regina Estkowski and Joseph S. B. Mitchell, "Simplifying a polygonal subdivision while keeping it simple," in *Proc. 17th ACM SCG*, 2001, pp. 40–49.

[13] J. Wu and L. Kobbelt, "A stream algorithm for the decimation of massive meshes," in *Graphics Interface'03 Proceedings*, 2003, pp. 185–192.

[14] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink, "Large mesh simplification using processing sequences," in *Visualization'03 Proceedings*, 2003, pp. 465–472.

[15] David F. Maune, *Digital elevation model technologies and applications : the DEM users manual*, chapter 13, American Society for Photogrammetry and Remote Sensing, 2001.