Streaming Formats for Geometric Data Sets

Martin Isenburg*
Max-Planck-Institut für Informatik
Saarbrücken

Peter Lindstrom
Lawrence Livermore
National Laboratory

Stefan Gumhold
Max-Planck-Institut
für Informatik

Jack Snoeyink
University of North Carolina
at Chapel Hill

Abstract

Recent years have seen an immense increase in the complexity of geometric data sets. Today's gigabyte-sized 3D models can no longer be completely loaded into the main memory of common desktop PCs. Unfortunately, most storage and exchange formats for geometric data do not account for this. They were designed years ago when models were orders of magnitudes smaller. Using these formats to store and distribute giga-byte sized data sets is inefficient and unduly complicates all subsequent processing.

In this talk we will describe streaming formats for geometric data that are basically as simple as existing formats but more suitable for storing large data sets than all current alternatives. Such formats contain tiny bits of additional information that "finalize" previously read data. This information specifies which elements of a mesh or which areas in space have already been completely traversed. This gives the necessary guarantees to safely process these parts of the data and deallocate the corresponding data structures without first parsing the entire data set. While the focus of this talk is mainly on "topological streaming" of unstructured meshes, we will also motivate "spatial streaming" of meshes and point clouds.

1 Motivation and Overview

Modern scientific technologies enable the creation of digital 3D models of incredible detail and precision. These geometric data sets easily reach sizes of several gigabytes, making subsequent processing a difficult task. The sheer amount of data may not only exhaust the main memory resources of common desktop PCs, but even exceed the address space limit of a 32-bit machine. To process such data sets, one resorts to *out-of-core* algorithms that arrange the data so that it does not need to be kept in memory in its entirety, and adapt their computations to operate mainly on the loaded parts.

But for unstructured surface or volume meshes, already the way the raw input data is stored can turn the simplest pre-processing into a highly IO-inefficient operation. Current mesh formats use an array of floats to specify the vertex properties followed by an array of indices into the vertex array to specify the polygons or polyhedra. Storing large meshes in such a format means that one gigabyte-sized array of data is indexed by another gigabyte-sized block of data. Since the order in which the mesh elements appear in these arrays is left unspecified even simple de-referencing (i.e. resolving all vertex references) can potentially thrash the memory.

The ineffiency of indexed mesh input has been addressed in large mesh papers for the last eight years. [Chiang and Silva 1997] write that "Unfortunately, the datasets are often given in a format that contains indices to vertices. Thus we have to de-reference the indices before actually building the interval tree." and propose to use external sorting for this. Despite requiring large amount of scratch space and multiple passes over the data, this has since become the standard mechanism for dealing with large indexed meshes. Recent works often try to abandon indexed meshes altogether. [Cignoni et al. 2004], for example, assume that "the mesh is represented as a triangle soup, i.e., a list of triangles with direct vertex information". But as most their data sets are originally stored as indexed meshes, like the 3D scans of Michelangelo's statues [Levoy et al. 2000], they still need to de-reference in a pre-processing step.

*isenburg@cs.unc.edu http://www.cs.unc.edu/ ~ isenburg/sm

We will try to convince the audience that *streaming mesh* formats are much better suited for storing and distributing large meshes than current alternatives. First, they do not have the problem of in-efficient dereferencing, second, they are a more "natural" output format for memory-limited applications that generate large meshes, and third, they are an ideal input and ouput format for I/O-efficient algorithms that perform out-of-core stream processing.

The basic idea is to logically interleave vertices and the mesh elements that reference them and to provide explicit information about when vertices are "finalized" or "referenced for the last time". While the required changes to go from existing formats to streaming formats are minimal, the payoff can be substantial. Because the format tells us which of the previously read vertices to keep in memory, we can trivially de-reference such meshes in an IO-optimal manner—the problem of repeated, possibly incoherent look-up of vertex data in a gigantic array does not exist. And because the format tells us which vertices can safely be deallocated because they are no longer needed, we can do this for meshes of practically arbitrary size while requiring only moderate amounts of memory.

But a streaming mesh format is not only a better input format for large meshes—it is also a more natural output format for most mesh generating applications. Given limited memory resources, it is in fact quite *difficult* to output meshes into standard indexed formats. A mesh generating application that can only hold and work on small pieces of the data at any time will need to store vertices and triangle into seperate temporary files and concatenate them later. Memory mapping the vertex and triangle arrays is not possible without knowing the exact size of the vertex array in advance. For example, an out-of-core marching cubes iso-surface implementation that processes the volume layer by layer will naturally output vertices and triangles in the same order. And vertices from the last layer can trivially be finalized before moving on to the next layer.

Furthermore, a streaming mesh format is the ideal input and output for stream processing. In this model, the mesh streams through an in-core buffer, which is large enough to hold all active mesh elements. For straight-forward tasks, such as rendering a flat shaded mesh, a minimal stream buffer is needed. For more elaborate processing tasks, a larger stream buffer may hold as many additional mesh elements as there are memory resources, allowing random access to a localized but continuously changing subset of the mesh.

Streaming meshes allow pipelined processing, where multiple tasks run concurrently on separate pieces of the mesh. One module's output then serves as the input for another module. Envision a scenario where one algorithm extracts an isosurface and pipes it as a streaming mesh to a simplification process, which in turn streams the simplified mesh to a compression engine that encodes it and immediately transmits the resulting bit stream to a remote location where triangles are rendered as they decompress. In fact, we now have all components of this pipeline—and it is the streaming format that makes it possible to pipe them all together.

References

CHIANG, Y.-J., AND SILVA, C. T. 1997. I/O optimal isosurface extraction. In *Visualization '97*, 293–300.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2004. Adaptive tetrapuzzles - efficient out-of-core construction and visualization of gigantic polygonal models. In *SIGGRAPH 2004*, 796–803.

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The Digital Michelangelo Project. In *SIGGRAPH 2000*, 131–144.