EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

Streaming and Data Enrichment

M. J. McDerby, W. T. Hewitt, M. J. Turner

University of Manchester, UK

Abstract

Data is being created at a continuously increasing rate. Scientists are "drowning in their data". So much so they cannot visualize the data to see the big picture. This work-in-progress paper describes a data capture and visualization problem, and the steps being undertaken to solve the problem. An overview is given of the various approaches to dealing with these large data sets that have been previously proposed and the application of such methods within a well known scientific visualization tool. The paper then goes on to propose a method that deals with large data visualization by addressing the bottlenecks in the visualization pipeline, and combining some of the approaches described herein with parallel techniques on a high performance visualization system.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation I.3.4 [Computer Graphics]: Graphics Utilities I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling I.3.6 [Computer Graphics]: Methodology & Techniques

1. Introduction

Increasingly there is a need for the development of tools to visualize extremely large datasets. Advances in computing technology are making simulation as important to science today as theory and experiment have been in the past. However, the amount of data that can be produced by these technologies is overwhelming, in fact "Scientists are drowning in their data" and whilst scanning and raw computing technology provide the means to acquire or generate datasets, other technologies such as storage, networking and graphics have not kept up - creating "technological bottlenecks". Due to these bottlenecks more data is produced than can be analyzed or visualized. The main aim of this project is to address these bottlenecks by reviewing the standard Haber & McNabb Reference Model [HM90], and proposing enhancements to handle datasets more effectively so as to develop multi-resolution software tools which will facilitate the understanding of large-scale simulation datasets.

This paper describes the work-in-progress to date and the initial evaluation of the problem. Section 2 covers previous work in the area of large data visualization and approaches to streaming. Section 3 discusses the development of streaming in AVS/Express, and how data is currently handled. Section 4 gives an overview of streaming and the visualization pipeline. In Section 5 we present the proposed approach and Section 6 goes on to discuss future work.

2. Previous Work

Over the last decade extensive work has gone into out-of-core algorithms to work on large datasets. Out-of-core is essential for data that will not fit into memory, so methods have been devised to simplify/compress meshes in order to visualize them. Such methods include, mesh cutting, dereferenced triangle soup, clusters and more recently streaming representation. However, special attention has been given to the initial format of the input mesh as this can complicate the subsequent methods.

Levoy et al. [LPC*00] used mesh cutting in the Digital Michelangelo project. Michelangelo's David when scanned consisted of about 2 billion polygons, the data was stored as range images (2D) instead of polygon meshes because of the size of the 3D mesh (36 gigabytes). The authors [LPC*00] designed their own file formats to map the range values to 3D. However, problems still lay ahead as the authors report that no matter how efficient their storage mechanism was - they could still not display the complete model. The form of mesh cutting used was to construct range pyramids. However, to display a range image, or to merge multiple range images or to perform geometric operations on the data all need the 3D points to be converted to a triangle mesh.

Ho et al. [HLK01] took mesh cutting one step further by compressing the sub-meshes using mesh compression techniques. However, they go onto report that existing mesh compression and decompression algorithms are only effective if a representation of the mesh's entire topological and geometric structures (and some other attributes) are small enough to fit in-core. For the mesh compression stage (which involves 3D geometry), the relationship between the vertices and faces is multi-directional. Mesh compression algorithms aim to create a linear ordering on the mesh (an ordering based on spatial proximity which aims to reduce the vertex or face traversals). However, the authors admit that they have completely ignored the geometry of the simplified mesh, and that it would be interesting to see if the geometry of the simplified mesh can be used to further reduce the compressed geometry of the input model. In fact, to proceed further in their direction they say that a more refined and geometry-oriented partition scheme is probably required.

External memory data structures is another way of partitioning a mesh, but the pieces are a lot smaller and are referred to as clusters. Cignoni et al. [CMRS03] presents a data structure called an Octree-based External Memory Mesh (OEMM) that is based on a hierarchical geometric partitioning of the data set with no vertex replication. The aim is to have consistent vertex indexing between leaf nodes which share a reference to the same vertex. The hierarchy is coupled with a partial knowledge of the geometry and topology which allows for a form of element tagging strategy. The mesh format is considered in the simplification of the mesh, but no re-ordering of the mesh is undertaken to reorganise the data so as to make it easier to process. Hence, many passes have to be done over the data to finalize triangles which involves an extra time overhead.

So far we have considered indexed meshes - a mesh is called indexed if all the triangles are encoded by storing a triple of references to their vertices. Conversely, if the data only consists of a list of the triangles and the sharing of vertices is not considered then this is called a triangle soup. In [Lin00] Lindstrom describes a method of out-of-core simplification for triangle soups. The algorithm is based on uniform sampling via vertex clustering, and is enhanced by the use of error quadrics. The use of a triangle soup dataset enhances the simplification speed, but roughly uses twice the disk space as compared with a mesh dataset. This they reported can be solved by compressing the data on disk and then decompressing it on the fly during simplification. The algorithm unfortunately does not perform adaptive sampling, which means that often models have to be further coarsened.

One of the most recent surveys of streaming techniques was published in 2005 [IL05]. This refers to the current mesh formats and the failings with regards to storage and retrieval of large data sets. Mesh formats are limited and so is the common desktop, which people have to work with where, the main memory limits are often set to a maximum 4 gigabytes. This paper presents a streaming format for polygon meshes that is more suitable for large meshes and still works within the visualization pipeline.

In order to process geometric data sets that do not fit in main memory it is essential to use out-of-core algorithms to arrange the mesh so that it does not need to be kept in main memory in its entirety, and adapt their computations to operate only on the loaded parts [IL05]. Most commonly they create a streaming mesh – which is a mesh that is a logically interleaved sequence of indexed vertices and triangles with extra or meta-information about when vertices are introduced and when they are finished with.

In most of the research covered a lot of the work as shown in [LPC*00] ignored the problem with meshes and used an alternative method rather than tackle the problem of large meshes head on. Hence, they still end up with the problem of converting back and forth from 3D range data to mesh data when they want to do some form of geometric operations. In [IL05] Isenburg et al. reports that the main problem of most of these algorithms is the initial format of the input. This is because the mesh formats being used were designed many years ago when meshes were smaller and less complex. One common mesh format - PLY - uses an array of floats to specify vertex positions, followed by a second array of indices into the vertex array to specify the polygons. The order in which the vertices and polygons are arranged is varied upon the methods used to create the mesh. This Isenburg et al. say was convenient when meshes were much smaller, but as meshes have become larger processing of such meshes i.e., dereferencing - resolving triangle-to-vertex references can slow down. Therefore, to visualize a polygon, a search has to be done throughout the data for the vertices that make up the polygon - if the data is effectively random and in no subsequent order, then this can substantially reduce performance. Figure 1 shows a scan of the Nukhul mountain range in Egypt - the dataset of this scan is a typical problem for Earth Scientists at The University of Manchester. The scan of the mountain range consists of 16 separate scans totalling a size of 19 gigabytes which cannot be easily viewed all at once without using visualization techniques such as downsizing. But researchers need to see all the information from the scan in order to analyse the rock content. The data resultant from the scan is a perfect example of an incoherent mesh and is shown in Figure 1. The different colours show the way in which the data is visualized to the screen, and demonstrates it's incoherence.

Figure 1: *Incoherent Nukhul Mesh*

Figure 2 shows a coherency graph of the same data mesh. This graph is a layout diagram (the triangles and vertices are laid out in the order they occur in the original data file) which connects triangles that share the same vertex with horizontal line segments (green), and vertices referenced by the same triangle with vertical line segments (grey). The further the green and grey line segments are from the major diagonal, the less coherent the layout is.

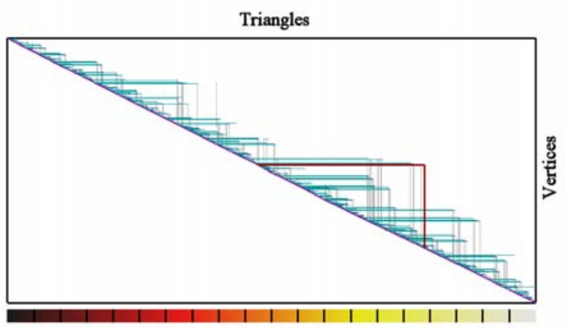

Figure 2: *Incoherent Layout Graph of Nukhul Data (Consisting of n vertices and m triangles).*

An easier way of explaining incoherence is shown using a simple test example in Figures 3 and 4. Figure 3 shows a small subset of the incoherent Nukhul mesh in 2D.

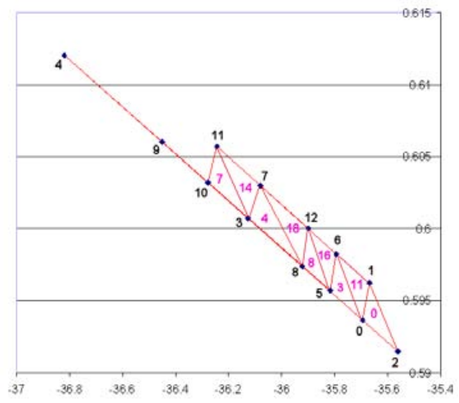

Figure 3: *Small Mesh*

Figure 4 shows the incoherence layout diagram for this subset. Vertices are shown along the y axis and triangles along the x axis. For each vertex of a triangle a purple point is drawn, and a vertical line connects all the vertices that make up that triangle. For each vertex a horizontal line is drawn which shows all the triangles that reference it.

In [IL05] a characterization of a layout of an indexed mesh is made to define it's "streamability". The triangle span

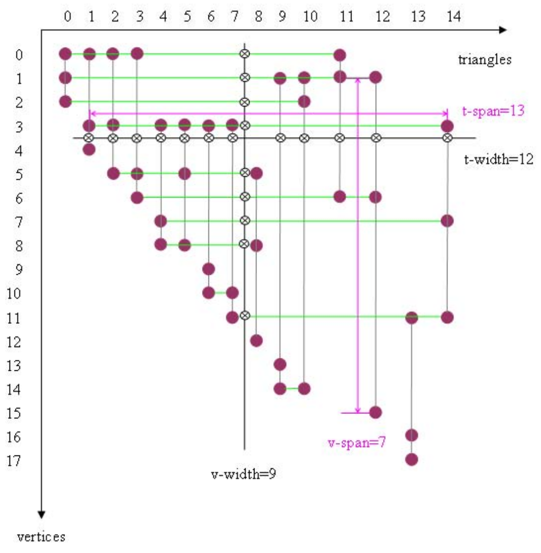

Figure 4: *Incoherence Graph*

(t-span) in the layout diagram (shown as a horizontal line with arrows at either end) is the number of triangles between and including the first and last reference to the vertex. The vertex span (v-span) represented by the vertical double sided arrow is the maximum index difference (plus one) of its vertices. The vertex span of a layout is the maximum span of all triangles (vertices). The vertex width of a layout is the maximal number of horizontal line segments that can be cut by a new vertical line - 9 in Figure 4; the triangle width is the maximum number of vertical line segments that can be cut by a new horizontal line - 12 in Figure 4. Finally, the skip of a layout is the maximum number of "concurrently" skipped vertices. There is no skip in the Nukhul test example shown.

A major issue in large meshes is de-referencing which is basically resolving all triangle-to-vertex references [IL05]. This slows down streaming, as algorithms have to search about finding the vertices of the triangles in question. The algorithms presented in the paper resolve this issue by finalizing the triangle-vertex relationships by interleaving indexed vertices and triangles, and provides extra information about when vertices are last referenced.

3. Streaming in AVS/Express

To support streaming the visualization software must support breaking the data set into pieces and correctly processing those pieces. Ahrens et al. [ABM*01] stated that the data must be separable, mappable and result invariant.

- *Separable*. Breaking the data into pieces - each piece

should be coherent in geometry, topology, and/or data structure. Separating the data should also be both simple and efficient. In addition, the algorithms in this architecture must correctly process pieces of data.
- *Mappable*. Controlling the data streaming through the pipeline - we must determine what portion of input data we need to generate a given portion of the output. Allowing control of the size of the data through the pipeline.
- *Result invariant*. Results should be independent of the number of pieces and the execution mode (single or multi-threaded).

Most visualization tasks within AVS/Express are carried out using chunks e.g., an isosurface is comprised of many chunks of a chosen number of triangles. Lever et al. [LPL05] reports that such chunking of data improves handling of memory and rendering speed. Under asynchronous behaviour, chunks of renderable data are rendered and hence cached by the graphics hardware. Chunking has two benefits:

1. Users can see the visualization progressively appear as it is produced, which is particularly useful for large datasets.
2. The graphics pipes may cache more efficiently when geometry is streamed through them.

For example, taking the isosurface algorithm chunking can be carried out by taking strips of the data. The data must already be de-referenced, but it is not important that there is any pre-intelligent processing.

This simple chunking system has the following drawbacks:

- Triangles are taken in chunks, with no regards for finalization.
- Re-calculation upon boundary measures to ensure triangles line up (result invariant).

However, Law et al. [LMST99] reports that commercial visualization systems such as AVS and IBM Data Explorer fail in two important ways when dealing with large data:

1. Interactive control of the visualization process is lost when the time to transmit, process, or render data becomes very long.
2. The physical or virtual memory address space of the computer is overwhelmed, and the system thrashes ineffectively or crashes catastrophically.

The paper goes on to say that the typical response to this is to go out and buy bigger computers such as a graphical supercomputer, and that such solutions are only available to wealthy users but this still may not solve the problem.

4. Streaming and the Visualization Pipeline

In [HM90] Haber and McNabb present a generic and very popular model of scientific visualization as a complex set of generalized mappings of data obtained from simulation, observation or experiment. The mappings transform raw data into a geometric abstraction of the scientific information content, which can then be rendered to a displayable image using computer graphics or image processing. Any specific sequence of mappings constitutes a visualization idiom.

The authors take the view that visualization is a series of transformations that convert raw data into a displayable image. The goal of these transformations is to convert the information to a format that is understandable by the human perceptual system while maintaining the integrity of the information. Three major transformations occur in most visualization procedures:

- Data Enrichment/Enhancement
- Visualization Mapping
- Rendering

Bottlenecks from large data sets in the visualization pipeline usually occur at each of these transformations.

Following on from Law et al. [LMST99] the key to a structured data streaming pipeline is the ability to break the data into pieces. By controlling the size of the pieces of data, we can avoid memory swapping. The size of the piece of data needs to be controlled according to the size of the computer memory and the number of filters in the visualization pipeline. Therefore, there are two benefits from streaming data through the visualization pipeline.

1. We can run visualization that wouldn't normally fit into memory or swap.
2. We can run visualizations with a smaller memory footprint.

5. Proposed Approach

One of the aims of this research is to address the bottlenecks in the traditional visualization pipeline using AVS/Express PST as the scientific visualization tool. This combines streaming of data as outlined in [IL05] with a new approach to the visualization pipeline as detailed below:

1. Pre-processing the mesh format to allow for a more economic distribution across the processors involved.
2. Distribute the data following the work in [IL05], but parallelised over multiple processors.
3. Processing the data using parallel techniques inside of PST.
4. Rendering the data using compression techniques [IL05].

5.1. Pre-processing the Data

The first task was to implement a streaming mesh format which would allow for coherency in the data. Figures 1 to 4 show the initial investigation and the coherency of the meshes that were obtained from the scan of the Nukhul region.

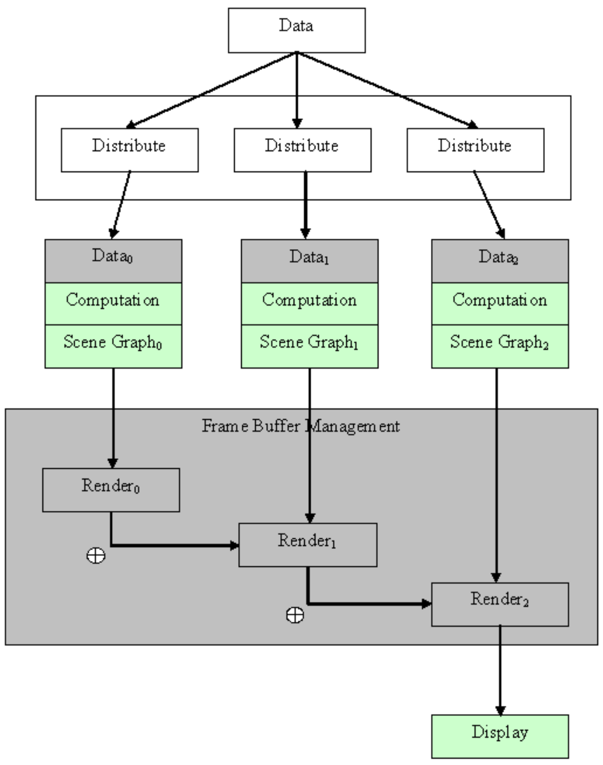

Figure 5: *AVS/Express Parallel Support Toolkit (PST) Visualization Pipeline. Data is distributed as it is read in, so as to speed up the reading of large datasets. The main bottleneck is at the rendering stage.*

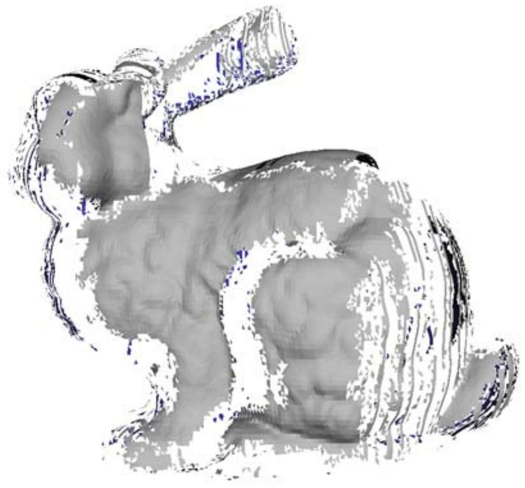

Figure 6: *Original Stanford Bunny Data Order Streaming*

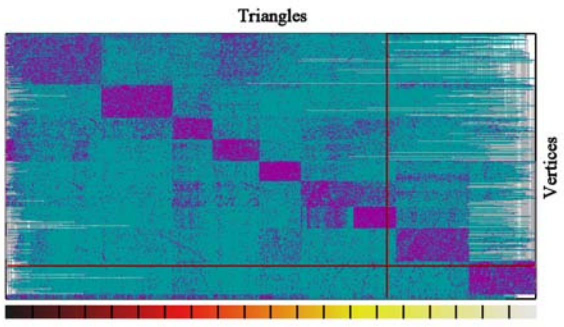

Figure 7: *Coherency Layout of the Original Bunny Data*

This data is unstructured so by sorting we can increase the coherency.

To demonstrate increasing coherency and streaming with a well known dataset - the Stanford Bunny is shown in it's original data format in Figures 6 and 7 with final results shown in Figures 8-10.

To make the data streamable a series of tasks has to be undertaken to create a vertex layout, a triangle layout, and finalization information.

1. In an initial pass over the input mesh write the vertices and triangles to separate temporary files.
2. Store with each vertex its original index (so it can be identified by its triangles).
3. Compute vertex degrees (this is a count of the lifetime of the vertex and is decremented each time the vertex is used). This is used in the finalization step.
4. Specify explicit/fixed layouts as required.
5. For a specified triangle layout, sort keys (k) are assigned to vertices (v) based on their new incident triangle indices (t).
6. The sort keys and vertex degrees are merged with the triangles and vertices in their temporary files. The result is a file with vertex records (k_v, v, d, x, y, z) and a file with triangle records ($k_t, v1, v2, v3$).
7. An external sort is then performed on the two files (on increasing key value) to bring the elements into their desired order.
8. A streaming mesh is then output by scanning these files in parallel.

Explicit/fixed layout is mentioned in item 4, this really depends on the data. A dataset may have coherent layouts for both triangles and vertices, but they may not be compatible in that they are sorted along different axes. According to Isenburg et al. [IL05] it may be sensible then to keep one of the original layouts fixed and to re-arrange the other layout so that it is compatible. An explicit layout directly gives you a set of unique sort keys. This could be as simple as taking spatially sorted vertices, and using the x-coordinate as the sort key, or the same as in item 6 above.

De-referencing is accomplished by performing external sorts on each vertex field in an explicit vertex order (item 7 in the list) and updating their vertex indices. Finalization is achieved using the vertex degree and decrementing it each time a vertex is used.

The vertex file contains (k_v, v, d, x, y, z) where:

- k_v is the sort key based on the new incident triangle indices t;
- v is the original vertex order;
- d is the vertex usage/degree;
- x, y, z, is the vertex.

The triangle file consists of: $(k_t, v1, v2, v3)$ where:

- k_t is pre/post order triangle key (based on the indices in the reordered layout);
- and $v1, v2, v3$ are the vertices which make up the triangle.

Pre/Post-order is defined in [IL05] and is the order upon which you choose to interleave the triangles and vertices - pre-order is where each vertex precedes all triangles that reference it and post-order is where each vertex succeeds all triangles that reference it.

Figure 8 shows the data sorted by Y with colour mapping representing the streaming order and Figure 9 shows the path of the data streaming for a possible single chunk. The coherency graph of the Stanford Bunny data sorted by Y is shown in Figure 10. This sort has had a tremendous effect on the coherency of the data - but may not be economical enough as there is still some fuzziness around the diagonal line in Figure 10. Section 5.2.1 discusses this further.

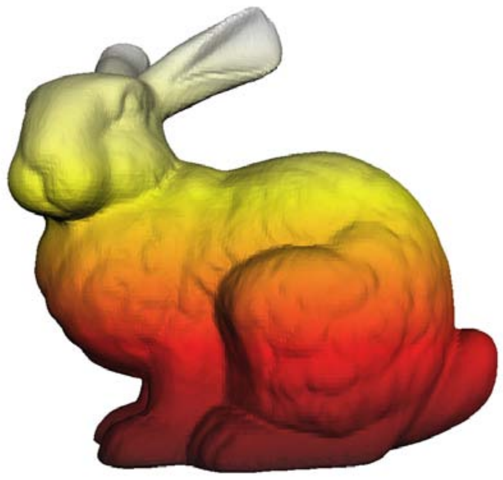
Figure 8: *Stanford Bunny Sorted By Y*

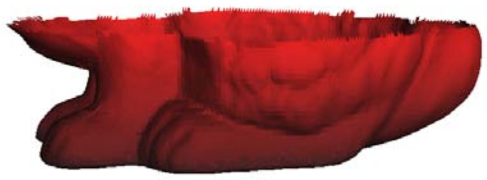
Figure 9: *Stanford Bunny Sorted by Y Rendering*

5.2. Streaming

The next phase then is to implement a streaming process:

- Large data is read streamed in.
- Data is distributed using PST.
- Overheads with respect to streaming the data across many processors have to be taken into account at this point.
- The data is then piped to a parallel visualization module such as isosurface.
- This in turn is streamed to a simplification process.
- Which then streams it to a compression engine.
- The resulting bit stream is then encoded and transmitted to a remote location, where the triangles are rendered as they decompress.
- Parallel rendering will be undertaken using compositing.

With regards to the reading and distribution of the data the aim is to partition the mesh into sub-meshes, ensuring that the overlapping partitions are minimised such that the amount of information that must be shared between processors is minimised.

5.2.1. Overheads

The Stanford Bunny data has 35947 vertices and 69452 triangles. When sorted in Y and divided across 4 processors (17363 triangles per processor) the overhead which will restrict the visualization time can be calculated via the following performance model summising the Overlap between p1 and p2 + Overlap between p2, p1 and p3 + Overlap between p3, p2 and p4 + overlap between p4 and p3 divided by number of processors:

$$\frac{2363+4354+5513+2554}{4} = 3696 \qquad (1)$$

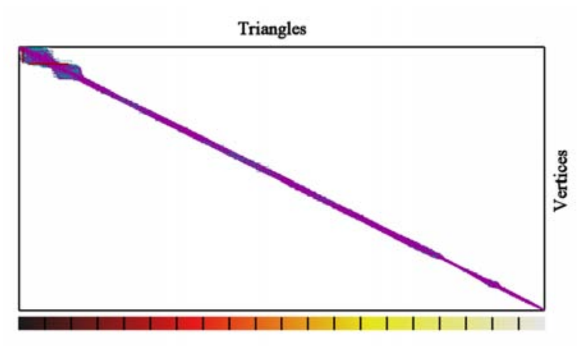

Figure 10: *Coherency Layout of the Stanford Bunny Data sorted by Y*

Similarly, the average overlap for 2 CPUs (34726 triangles per processor) is 2294 vertices, for 6 CPUs (11576 triangles per processor) it is 3417, and the result for 8 CPUs (8682 triangles per processor) is 3959 vertices per CPU.

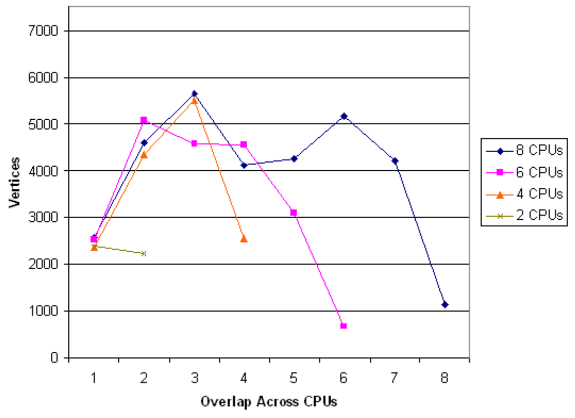

Figure 11: *Overheads across CPUs*

Figure 11 shows the series of experiments with the sorted Bunny data visualized across a number of processors. Most of the work is performed by the processors where there is a bigger overlap due to the fact that the data is still not coherent enough i.e., all triangles and vertices which rely on each other should be dealt with on the same processor (or the same chunk of data). As mentioned in Sections 2 and 5.1 the aim is to remove de-referencing by means of sorting the data in such a way that the vertices and triangles that rely on each other are interleaved and compact. With regards to the way the Bunny data is sorted, both Figure 11 and the performance model above show that the data is still not in a truly economical ordering, and that the more processors that are used results in a steady increase in the average overlap. Which shows that the triangles and vertices that rely on each other are still largely distributed throughout the data. Such large amounts of vertices in the overlaps can (or cell sets in AVS/Express) cause problems at the rendering time in that if a large cell set resides on a single processor, then the processing is essentially serialized as all other nodes are idle until that processing node has completed it's work.

An alternative sorting method for the streaming of the data and a possible solution to the problem mentioned above is Spectral Sequencing [KCH02, IL05]. According to Koren the spectral approach has two distinct advantages that make it very attractive:

1. Mathematically-sound formulation leading to an exact solution to a layout problem, improving on other formulations that result in an approximation.
2. Computation speed. This is very important as we have repetitively said in this paper that the amount of information to be visualized is growing rapidly.

However, it can have drawbacks for large data sets since it relies on eigenvector computation, so as an alternative Liu et al. [LJZ06] reports that Nystrom approximation (a subsampling and reconstruction technique orginating from integral calculus) can be used. Isenburg et al. [IL05] pre-simplify vertices into clusters using streaming edge collapses. These clusters are kept in a memory-mapped array as circular linked lists of vertices. The ACE (Algebraic multi-grid Computation of Eigenvectors) algorithm by Koren et al. [KCH02] is then applied to order the clusters in-core, and then to order the mesh cluster by cluster - with no particular vertex order within each cluster.

6. Future Work

The ACE [KCH02] method is used to order clusters in-core, with no particular vertex order within each cluster. This can be improved upon by ordering the vertices within each cluster to minimise the streamability measures and further ensure coherence.

As mentioned above the aim is to develop this work with AVS/Express MPE/PST, thereby ending up with a suite of tools that allow the streaming, and compression of large meshes in a parallel manner. Future work could be to develop the approach even further and allow distributed streaming visualization. Using several machines one for each phase of the pipeline i.e., data enrichment phase - sorting and streaming of data undertook on one machine, data is then streamed to a visualization supercomputer whereby the visualization mapping phase is undertaken using PST, with the displayable object being displayed in a multi-pipe environment.

Going one step further, the streaming and sorting of the various scans (note that the Stanford Lucy model was made up of originally 47 scans, and the Nukhul mountain scan is made up of 16 scans) can also be distributed across machines to speed up the computation.

Some other areas that are still to be looked at in the research arena is multi-resolution streaming of structured and unstructured meshes.

7. Acknowledgements

The authors wish to thank Peter Lindstrom, Lawrence Livermore National Laboratories, for his advice on streaming. David Hodgetts, School of Earth, Atmospheric and Environmental Sciences, University of Manchester for the dataset and time. Many thanks also go to the following staff in the Manchester Visualization Centre; Robert Frank, Yien Kwok, George Leaver, Louise Lever, Tobias Schiebeck.

References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-scale data visualization using parallel data streaming. In *IEEE Computer Graphics and Applications* (2001), pp. 34–41.

[CMRS03] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: External memory management and simplification of huge meshes. In *IEEE Transactions on Visualization and Computer Graphics* (2003), pp. 525–537.

[HLK01] HO J., LEE K., KRIEGMAN D.: Compressing large polygonal models. In *IEEE Visualization* (2001), pp. 357–262.

[HM90] HABER R. B., MCNABB D. A.: Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing* (1990), IEEE, pp. 74–93.

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *Visualization* (2005), IEEE, pp. 231–238.

[KCH02] KOREN Y., CARMEL L., HAREL D.: ACE: A fast multiscale eigenvector computation for drawing huge graphs. In *IEEE Visualization* (2002), pp. 137–144.

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *Siggraph* (2000), pp. 259–262.

[LJZ06] LIU R., JAIN V., ZHANG H.: Subsampling for efficient spectral mesh processing. In *Computer Graphics International* (2006), pp. 172–184.

[LMST99] LAW C. C., MARTIN K. M., SCHROEDER W. J., TEMKIN J.: A multi-threaded streaming pipeline architecture for large structured data sets. In *Visualization 1999* (1999), IEEE, pp. 37–44.

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital michelangelo project: 3D scanning of large statues. In *Siggraph* (2000), pp. 131–144.

[LPL05] LEVER L. M., PERRIN J. S., LEAVER G. W.: Integrated parallel rendering for AVS/Express. *CSAR Focus*, 14 (2005), 19–21.