# Techniques for Large Data Visualization

Dan R. Lipşa[1], Robert S. Laramee[1], R. Daniel Bergeron[2], and Ted M. Sparr[2]

[1]Visual and Interactive Computing Group, Department of Computer Science, Swansea University, Swansea, UK
[2]Computer Science Department, University of New Hampshire, Durham, NH, USA

February 18, 2011

**Abstract**

Often scientific datasets are several times larger than the main memory of a computer. The size of datasets, in general, has exceeded that of main memory for several decades and will continue to do so for the foreseeable future. Because of large disk-drive latency, visualization algorithms designed to process data from main memory can rarely be directly applied to data stored on disk without modification. In this paper we review current methods and techniques designed to deal with large data, often larger than the computer's main memory. Our goal is to provide a student or researcher with understanding of fundamental concepts and knowledge of the most important techniques in the current research literature for visualizing large scientific data. The most important terminology related to out-of-core visualization is identified and discussed as well as the fundamental challenges faced by this class of techniques. We provide a valuable starting point for readers interested in gaining a concise introduction of techniques for large data visualization.

**Keywords:** Out-of-core/external algorithms; caching and prefetching; multiresolution and adaptive resolution; chunking.

## 1 Introduction

A scientific dataset consists of spatial data values either collected from the real world or produced by a computer simulation. Several different attributes (for instance the concentration of several chemical elements) can be acquired or produced for each point in space and these measurements can be repeated over a period of time.

As the processing power of CPUs continues to increase and GPUs start to provide parallel processing capabilities to applications, computer scientists have made progress in dealing with the large processing requirements of scientific applications. However, large disk-drive latency, and very large data sets continue to be major problems in processing scientific data.

Often scientific datasets are very large, much larger than the main memory of a computer. Because of large disk-drive latency, visualization algorithms designed to process data from main memory can rarely be applied to data stored on disk directly without modification. A visualization algorithm which does not preload all the data to be visualized in main memory is called an *out-of-core* or an *external memory* visualization algorithm. These algorithms usually require special optimization to circumvent the ill effects of large hard-drive latency.

This paper focuses on the most important methods and techniques used in visualizing large scientific data. It provides a tutorial for those less familiar with the topic, as such it is not an exhaustive overview of the research area. Our focus is on presenting the most important techniques in greater detail for readers new to this topic, rather than exhaustive coverage of all recent techniques.

This manuscript serves as a very good starting point for those that may be interested in the subject. A reader will gain a good understanding of the most important techniques and not be overwhelmed by the number of techniques presented. References to further literature, including surveys are given for the interested reader in studying the topic in more breath.

The rest of this manuscript is organized as follows: algorithms often used as a basis for more complex external memory techniques are presented in Section 2. Section 3 reviews the most important prefetching and caching techniques. Section 4 reviews multiresolution and adaptive resolution techniques and Section 5 reviews other ways of preprocessing data which can speed up visualization of scientific data.

## 2 External memory algorithms

External memory (out-of-core) algorithms process data that cannot fit into the main memory of the computer. This data is stored on an external memory device, commonly a disk drive, and pieces are brought to memory as they need to be processed. In contrast with the main memory, disk drives have very long access time. To reduce the ill effects of this long access time, the operating system reads and writes data from disk in blocks. This is important when estimating the running time of external memory algorithms.

In this section we review important external memory algorithms [21] that provide the basis of more complex visualizations.

To model complexity of out-of-core algorithms, Silva et al. [21] use I/O complexity which is defined as the number of I/O operations performed. I/O complexity is used in place of regular complexity (the number of operations performed) because an I/O operation (typical disk seek time is 10 ms) is several orders of magnitude slower than a memory access or a CPU operation (typical CPU speed is 2 GHz). In expressing I/O complexity of algorithms we use the following parameters: $N$ is the number of items in the problem instance; $M$ is the number of items that fit into main memory; $B$ is the number of items per disk block.

External memory algorithms can use one of two major computational paradigms for out-of-core processing of data [21]: *batched computation*, where no preprocessing is done and the program streams all data through the main memory only keeping a small portion of the data in main memory at any given time; *on-line computation* for which processing is done on the result of a query operation. In this case, data is often preprocessed such that the query operation examines only a small portion of the data.

## 2.1 Algorithms for batched computations

In this section we review four external memory techniques presented by Silva et al. [21]: external merge sort, the join operation, out-of-core pointer dereferencing and the meta-cell technique.

Sorting is a fundamental operation used in many out-of-core algorithms. External merge sort is a $k$-way merge sort, where $k$ is chosen to be $M/B$, so we can fit $k$ blocks in main memory at a time. We start with an unsorted list of $N$ items. If the list fits in memory, we just load it, sort it there and output the sorted list. If it does not fit, we split the list into $k$ sub-lists, we sort those sub-lists recursively and then merge the $k$ sorted lists. To merge $k$ sorted lists, we read one block from each list into memory, and we do a $k$-way merge. During the in-memory merge process, once a block finishes, we read in the next block from the same list. The I/O complexity for the external merge sort is $O(\frac{N}{B} \log_k \frac{N}{B})$.

*The join operation* takes as input two lists that contain related elements. Elements in the two lists are related through a field *key* that is contained in each element. Join produces a list that contains related elements from both input lists, that is, elements for which the key value is the same. To perform join in an I/O efficient way we external sort both lists on *key*, and then traverse both lists in parallel and write the output list. Considering the size of both lists to be $N$, the I/O complexity of this operation is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. This is much smaller than (denoted with $\ll$) $O(N)$ which is the I/O complexity of doing the same operation by directly traversing one list and accessing information referred to in the second list. To see that, notice that $\log_{\frac{M}{B}} \frac{N}{B} \ll B$ which, by using each side as an exponent to $\frac{M}{B}$, is equivalent with $\frac{N}{B} \ll (\frac{M}{B})^B$. This inequality is true for typical values for $N$, the number of elements, $M$ the number of items that fit into memory and $B$ the number of items in a disk block (for instance, for $N = 10^9$, $M = 10^6$ and $B = 10^3$ we have $\log_{\frac{M}{B}} \frac{N}{B} = \log_{10^3} 10^6 = 2 \ll B = 10^3$). Both out-of-core pointer dereferencing and the meta-cell technique, the two algorithms we present next, use join operations.

**Out-of-core pointer dereferencing**. Typically, irregular scientific data in index cell list format (ICS) stores a vertex list, with each vertex containing its x, y, z coordinates and attribute data, and a cell list, with each cell containing references to vertices that make up the cell. A common operation with scientific datasets is to process all cells in the dataset and access the information stored at each vertex of the cell. While this can be done very efficiently if the dataset is loaded entirely in memory, the same operation is very inefficient if the data resides on disk. In this case, traversing all cells in the dataset and accessing all vertices in a cell leads to random access in the disk file that stores the data with I/O complexity of $O(N)$. *Out-of-core pointer dereferencing* is an I/O efficient technique that stores the information associated with each vertex in a cell together with the cell itself. So, instead of a vertex list and a cell list, the dataset is stored as a cell list, with information about each vertex in a cell stored together with the cell itself. This technique duplicates information about vertices, but reduces the I/O complexity of traversing all the cells in the dataset to $O(\frac{N}{B})$. The transformation itself can be achieved with I/O complexity $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ which is much faster than $O(N)$ the time required to traverse all cells of the unprocessed dataset.

Let's see how we can perform out-of-core pointer dereferencing, that is, to obtain a cell list with each cell storing in-place information about all vertices in that cell, in an I/O efficient way. We start off with an irregular dataset in index cell list (ICS) format. First, we sort the cells in the cell list using as a key the index of the first vertex in the cell. This way, we obtain at the beginning of the cell list all cells that have vertex one as a first vertex in the cell, then we have all cells that have vertex two as a first vertex in the cell and so on. By traversing in parallel the vertex list and the cell list, we can fill in the information about the first vertex for all cells in the dataset. Note that this is a join operation between the vertex list and the cell list on the vertex index of the first vertex in the cell. We continue by sorting the cells in the cell list on the second vertex in a cell. We then do another traversal of both cell list and vertex list and fill in the information about the second vertex for all cells in the dataset (join operation on the second vertex index in a cell). We apply the same procedure for the third and the fourth vertices in a cell. The I/O complexity for this procedure is going to be $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ for a sort and $O(\frac{N}{B})$ for a traversal of the cell list and vertex list. So the overall I/O complexity is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$.

**Meta-cell technique**. The meta-cell technique [5, 21] spatially clusters neighboring cells together to form a meta-cell. Each meta-cell contains approximately the same number of vertices and is small enough to fit into main memory. Each meta-cell stores cells and vertices in index cell set (ICS) format. Each meta-cell is self-contained, which means that vertices that are shared between cells in different meta-cells are duplicated in those meta-cells. A cell is part of a unique meta-cell that is decided using a voting system based on the number of vertices in a cell that is part of a certain meta-cell. To be useful, meta-cell size has to be a multiple of the disk block size. This technique processes an irregular dataset in index cell set (ICS) format and produces a list of $k^3$ meta-cells each containing spatially neighboring cells together, where $k$ is an input parameter. The technique starts with an external sort of vertices on the $x$ coordinate and then partitions the vertices in $k$ chunks that contain an equal number of vertices. Each chunk is then externally sorted on the $y$ coordinate, and partitioned into $k$ chunks that contain the same number of vertices. Finally, each of the $k^2$ chunks is sorted on the $z$ coordinate and partitioned in $k$ chunks that contain the same number of vertices. This process produces $k^3$ chunks, each containing the same number of vertices. This partition is similar to the partition induced by a kd-tree [3, 12]. Each of these chunks corresponds to a meta-cell. To find out the cells and the shared vertices that are part of a meta-cell, further external sort and join operations are needed, so the I/O complexity is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. Full details for the meta-cell construction are presented by Chang et al. [5].

## 2.2 Algorithms for on-line computations

On-line computation algorithms are designed to use search data structures to facilitate fast queries through scientific data. A preprocessing step is necessary to build these data structures. In this section we present the B-tree, a well known external search data structure and a general technique designed to make it efficient to use a binary search tree larger than can fit into the main memory of the computer.

The best known out-of-core search data structure is the B-tree [9], which is a balanced, multiway search tree. Each node but the root has between $t$ and $2t$ keys and between $t + 1$ and $2t + 1$ children. The root of the tree can have between 1 and $2t$ keys and between 2 and $2t + 1$ children. The B-tree has the property that all the keys in the sub-tree attached between keys $k_i$ and $k_{i+1}$ have values between $k_i$ and $k_{i+1}$. The maximum number of keys in each node $2t$ is chosen such that a node has the same size as a disk block or is a multiple of disk block size.

Although binary search trees are common search data structures for in memory algorithms, they are not very effective for external use since they tend to be inefficient with respect to disk I/O. Silva et al. [21] present a general method for transforming binary trees into out-of-core search data structures. The main goal is to have a node in the tree occupy one or several disk blocks. Lets denote with $B$ the number of keys that can fit in a disk block. If we start with a balanced binary tree we can transform each sub-tree of height $\theta(\log B)$ into a node on disk. This node will have $\theta(B)$ keys and children. This operation will speed up searches through the tree by a factor of $B$.

# 3 Caching and Prefetching Techniques

Many of today's I/O optimization techniques are not well suited for scientific data processing. In particular, I/O system *prefetching* and *caching* algorithms treat every data file as a one dimensional sequence of data items and the effectiveness of the algorithms is based on the assumption that the file is likely to be read (either entirely or in large sections) in the order in which the data items are stored in the file. This assumption is very often false when applied to scientific data that represents data in a *spatial* domain. Such data is usually most effectively processed by being stored in a multidimensional array, which then must be mapped to the linear physical storage of a disk file. In this context, data values that are close neighbors in the multidimensional array (and in the visualization) can be very far apart on the disk. Consequently, interactive out-of-core visualization requires better tools for accessing today's I/O systems.

We review here techniques based on chunking the dataset, reordering the chunks and choosing the chunk size and shape such that the time needed to access data is minimized [18], choosing the right order for prefetching and discarding pages by using compression techniques [11], and building an application controlled paging system [10].

**Chunking**. Sarawagi and Stonebraker [18] present four techniques for improving access to multidimensional arrays stored on secondary or tertiary memory. These techniques are chunking, reordering of chunks, creating redundant copies of data with different chunk order and partitioning. The authors model access to data by defining an access pattern as a set of block shapes together with the probability of access for each block shape.

Basic *chunking* decomposes a multidimensional array into chunks of the same dimensionality, each chunk having the same size as the unit of transfer used by the file system. The shape of a chunk can be determined by exhaustive search such that for a given access pattern the number of blocks read is minimized. Chunks are laid out on disk by traversing the chunked array in axis order.

*Chunking with optimal chunk order* changes the axes traversal order for laying out chunks such that the number of blocks read for certain access pattern is minimized. The authors deduce a formula to determine the axis ordering that minimizes the time to access data for a certain access pattern.

*Redundant chunking* stores redundant copies of the array organized with different axis orderings for storing chunks. When a request arrives it can be routed to the replica with the smallest estimated access cost.

*Partitioning* is designed for tertiary memory devices, which consists of a large number of storage media and a few drives. A robot arm switches the media between the shelves and the drives in typically ten seconds. This technique consists of chunking an array with a chunk size that is the same size as the storage media.

**Compression techniques**. Curewitz et al. [11] use data compression techniques for prefetching. They view the sequence of page requests as a file to be compressed. Each page request is seen as a character from a finite alphabet which corresponds to the set of all pages in the database. A compression algorithm determines substrings that occur often in the file and encodes them with short codes and substrings that occur rarely in the file and encodes them with longer codes. The corresponding prefetching algorithm uses the calculated probability for page requests (characters) to prefetch the most probable k pages. The authors use three data compression algorithms to create three universal prefetchers, which make no assumption about the application or data representation. They simulate the use of their prefetchers on sequences of page accesses derived from OO1 and OO7 benchmarks and from CAD applications and they observed significant improvement in hit rate in comparison to using

LRU cache.

**Application controlled paging**. Cox and Ellsworth [10] use visualization of Computational Fluid Dynamics to test various ways of managing data that is larger than physical memory. They compare and contrast three ways of managing data that cannot fit in the available memory:

*Application controlled segmentation.* The application loads a small number of segments into memory at a time and processes them before moving on to a new set of segments. The size of each segment is application and data dependent which means that the size of a segment may be larger than physical memory. If that is the case, the application uses virtual memory and its performance degrades precipitously with the increase in the segment size and the decrease in available physical memory [10].

*Memory mapped file.* An improvement to the use of virtual memory for large data is the use of a memory mapped file. The improvement occurs when the traversal is sparse [10] and it happens because only the data actually needed is read from disk. The authors identify two problems with this method. First, there is no control over page size. Second, if data is stored in chunks in the file, it cannot be translated into linear storage in memory. Another drawback of this method is not mentioned by the authors. Namely, the operating system has a one dimensional view of n-dimensional data. This means points nearby in the volume may be stored in different pages in memory. For sparse traversal of data and adequate physical memory this does not slow down the application as only a few pages are needed to store the data to be processed. For algorithms that need to access the whole volume of data (such as volume visualization algorithms) and data that cannot fit in physical memory, this additional drawback causes the operating system to make poor decisions for loading and discarding pages.

*Application controlled paging.* In this case the paging system is managed by the application. The page size can be varied, and the application can translate from the storage format (compressed or chunked) in the file into linear format in memory. For three dimensional data stored in a linear file, this method suffers from the same problem as memory mapped files. That is, the file is split into pages that are uni-dimensional so the three dimensional nature of the data is ignored.

## 4 Multiresolution and Adaptive Resolution Techniques

A common technique to deal with large scientific datasets is to use a lower resolution version of the dataset for query, visualization and exploration. This can be done by using multiresolution (MR) and adaptive resolution (AR) representations of the dataset.

A multiresolution (MR) representation of the dataset consists of a hierarchy of versions of the original dataset, each having a different resolution. Typically a user starts by exploring the lowest resolution version, which is much smaller than the original dataset. Higher resolution versions of the dataset are used either when the user zooms in locally to view a region of interest or when the user stops the exploration so there is more time to render a higher resolution version of the entire dataset.

An adaptive resolution (AR) representation partitions a dataset into regions with different resolutions. Ideally, the resolution for each region is determined such that the error for that region is smaller than a user specified *error* for the dataset. A region of similar values will have a low resolution while a region with larger variations will have a high resolution such that the error for both regions will be smaller than the error specified for the dataset. The error for a region at the original resolution is zero. The error for a region at a lower resolution than the original is calculated by comparing the region at the original resolution with the region at the lower resolution.

The techniques we review cover both regular [4,22,23] and irregular datasets [8, 17]. All techniques use a preprocessing step to build the multiresolution hierarchy and use a low resolution to allow interaction with the user and use a higher resolution when the interaction with the user stops. Reviewed techniques for regular datasets deal with main memory size limitation by using compression [23], loading in main memory only a portion of the original volume [4] or loading in main memory low resolution levels and only a window of higher resolution levels [4,22]. Techniques for irregular datasets use points [17] or tetrahedrons [8] as rendering primitives.

While certain ways of storing data such as those in [8, 17] could be used for adaptive resolution data management, that was not investigated by the authors. No other references to adaptive resolution were found in the literature we reviewed.

**In memory compressed data**. Wetzel et al. [23] describe a system designed to help train students in an anatomy laboratory. The system stores Visible Human data on a server machine and uses inexpensive computers to view the data. The system implements a search facility that associates an anatomical feature name with a location and a bounding box, and inverse, a location in data with the feature name and bounding box. Since most anatomical features are not aligned with the orthogonal data axes, the system provides arbitrary direction slicing and the ability to show a flattened image of a spline surface.

The authors determine that storing data on the hard drive using chunked storage and $64^3$ chunks is not fast enough to serve the desired number of users (40 users) so they decide to store the entire data in the server's memory. They deal with the large amount of data in the data set in two ways: by using a hierarchy of resolutions built using wavelet transforms and by using compression.

The coarsest level in the hierarchy of resolutions uses a voxel which corresponds to a $64^3$ chunk. A small tree is produced for each voxel that provides fast access down to the level which corresponds to a $8^3$ chunk. The $8^3$ chunks are compressed and stored as variable length, entropy encoded bit strings. The authors use a lossless compression which achieves 3:1 compression ratio and investigate a lossy compression which has the potential to achieve a much better compression.

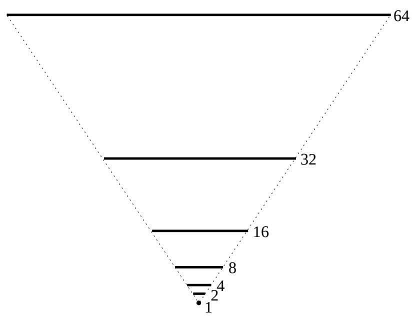

Figure 1: Side view for a mipmap for a texture of size $64^2$

**Clipmaps**. Tanner et al. [22] present the clipmap which is an extension of the mipmap texture representation to support arbitrarily large texture sizes. The mipmap [20] stores a pyramid of resolutions for a texture as shown in Figure 1. The mipmap stores at level zero the texture with the highest resolution which has a size multiple of two along each direction. At level one the mipmap stores a texture half the size level zero, along each direction. The process continues until only one pixel is stored at the apex of the pyramid. Each texture level is usually a lower resolution representation of the previous level, with level zero being the highest resolution.

The clipmap is a partial mipmap (see Figure 2), in which each level is clipped to a specified maximum size, which is usually the screen resolution. This clipping determines two kinds of levels: clipped levels which represent only a region of the entire surface and levels which were not clipped which represent lower resolutions of the entire surface. The region represented in a clipped level can be chosen to be anywhere in the original surface and it can be moved dynamically in response to user actions. For each clipped level, the original surface is stored on disk and is used to update that level as a response to user actions. The clipmap level 0 position inside the surface is specified by specifying the position of its center inside the surface. The positions of all other clipmap levels are calculated from the center for level 0 and the depth of the level (which depends on the level number).

To update the region stored in a clipped level, the authors use *toroidal addressing*, a clever addressing method that avoids the need to move overlapping data between the old and new regions. As the level number increases, the textures in different non-clipped levels in a mipmap cover the same surface at decreasing resolution whereas the textures in different clipped levels in a clipmap cover concentric regions of increasing size and decreasing resolution. The authors describe an SGI system with hardware support for clipmaps that can render a 170 GB texture at 60 Hz with 16 MB clipmap cache.

**Volume boundary approaches**. Bhaniramka and Demange [4] present techniques for dealing with large data used in OpenGL Volumizer, a commer-

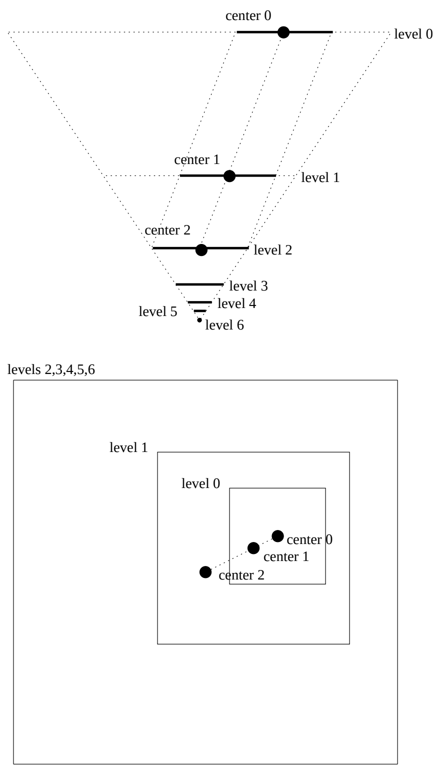

Figure 2: The top figure shows a side view of a clipmap. The thick lines show the window from each clipped level that is stored in the main memory. Non clipped levels (2,3,4,5,6) represent lower resolutions of the entire surface, clipped levels (0,1) represent rings of resolution surrounding level 0 as is shown in the bottom figure.

cial product created by SGI. The authors identify the following factors which can limit processing of large data: graphics card limitations such as fill-rate (textured pixels rendered per second) or size of texture memory, main memory size, data transfer limitations (bandwidth and latency between main memory and graphics card or between hard drive and main memory). Volume bricking is used as a basis for other techniques to insure efficient communication between disk and the main memory and between main memory and the graphics card memory.

To address graphics card limitations, the authors propose using several graphics cards and using either screen space decomposition (deals with the fill rate and geometry rate limitations) or data decomposition (deals with fill and geometry rate and texture memory limitations). The authors address main memory size and data transfer limitations through volume roaming, multiresolution volume rendering and 3D clip-textures (an extension in three dimensions of the clipmap technique of Tanner et al. [22]).

*Volume roaming* allows the user to explore the entire volume by interactively moving a volumetric probe anywhere in the original volume. The probe sits in the texture memory which allows the entire power of the GPU to be used to render it. A larger sub-volume that includes the probe sits in the main memory. This allows fast update of the texture memory if the user moves the probe within that sub-volume. The original volume sits on disk. This technique overcomes graphics card, main memory and data transfer limitations but the user can see only a sub-volume of the original volume and the frame rate may not be constant if the user moves the probe too fast through the volume.

*Multiresolution volume rendering* allows applications to interactively render a large volume by using a low resolution when the user interacts with the volume and using progressively higher resolution when the interaction stops. An octree is used to maintain the resolution levels of the data. The main problem with this technique is that the entire data has to be loaded into main memory, which limits the size of data that can be rendered.

*3-D Clip-Textures* were designed to combine the advantages of volume roaming and multiresolution volume rendering. This technique is an extension in three dimensions of the clipmap technique [22] covered earlier in this section. The library maintains a pyramid of resolution levels, but keeps in main memory only low resolution levels and a window of fixed size of the high resolution levels. As the user navigates the volume, the window can be updated from disk, and only new data needs to be read by using toroidal addressing. The different resolution levels are stored on disk using bricked storage.

The authors rendered a 6.7 GB volume with 1 GB resident in main memory and 256 MB texture memory at between 2 and 30 fps on a SGI Onyx system.

**Multiresolution point rendering**. Rusinkiewicz and Levoy [17] present QSplat, a system designed for representing and displaying meshes created by 3D scanning devices. These devices usually produce large meshes (hundred of millions of polygons) with relatively imprecise location for vertices. Other mesh simplification algorithms, which focus on vertex and edge placement and spend a large amount of effort per vertex, are too slow for this kind of data. QSplat achieves interactive frame rates by using a multiresolution hierarchy of bounding spheres to represent data at different levels of detail. The hierarchy is created through a preprocessing step, and is used for visibility culling, level of detail selection and rendering. Each node of the tree contains a sphere center position and radius, a normal, the width of the normal cone [19] and a color. Sphere center position and radius are quantized to 13 values and their values are relative to the parent sphere. Normals, colors, normal cone widths are also quantized to save space. The tree is built bottom up through a recursive process. Initially a sphere is created for each vertex. In the recursive step the current group of spheres is split in two along the longest diagonal of the bounding box for the group of spheres, a tree is built recursively for each of the two groups and then the two trees are used as children of a new node that stores the enclosing sphere for the two trees.

**Multiresolution tetrahedron rendering**. Cignoni et al. [8] present TAN, a system that stores and visualizes data at multiple resolutions. They use a model based on tetrahedral meshes, which is built off-line through data simplification techniques and is stored in a file using *index cell set* format (ICS) with additional information for each cell (tetrahedron). Irregular volume data is usually stored using index cell set (ICS) as a list of vertices and a list of tetrahedral cells. For each vertex we store the x, y, z coordinates in space of that vertex and all data attributes. Each cell contains four indexes (references) to vertices in the vertex list.

The authors use two measures to estimate the difference between an approximated mesh and the original mesh: *warping* and *error*. Warping of the domain of the dataset is an estimation of the difference between the volume spanned by the original mesh and the volume spanned by the approximated mesh. The error is defined as the difference between scalar values in vertices in the original mesh and scalar values calculated in vertices of the original mesh by using interpolation in the approximated mesh. Both warping and error are defined for convex, non-convex curvilinear and non-convex irregular datasets. For building an approximated model, the authors present both a refinement heuristic to be used for convex and non-convex curvilinear datasets and a decimation heuristic to be used for non-convex irregular datasets.

A refinement heuristic starts with a mesh that has as vertices a small subset of the original set of vertices, and then the mesh is iteratively refined by inserting vertices from the original mesh into it. The algorithm continues until its accuracy satisfies the required threshold. The refinement procedure starts with a Delaunay tetrahedrization of the convex hull of the original mesh. An online algorithm for Delaunay tetrahedrization is used together with a selection criteria for choosing the next vertex that is added for refining the mesh. The next vertex is chosen by the selection criteria such that it has the maximum error or warp from all vertices in the original mesh. The refinement heuristic assumes that adding a vertex that has maximum error or warp to the approximated mesh will yield the best mesh that can be obtained by adding only one vertex.

A decimation heuristic starts from the original mesh and iteratively removes vertices until the desired accuracy is achieved. The heuristic removes the vertex with the (estimated) minimum error or warp from all vertices. A vertex is removed by collapsing one of its incident edges to its other endpoint.

Each of the two algorithms produce an historical sequence of meshes from coarse to fine (increasing accuracy) for the refinement heuristic and from fine to coarse (decreasing accuracy) for the decimation heuristic. Two accuracies are stored for each tetrahedron: $\mu_b$ called the birth accuracy and $\mu_d$ called the death accuracy. For the refinement heuristic, these accuracies correspond to the worst and the best accuracies of the mesh containing the tetrahedron. To retrieve a mesh with accuracy $\mu$, all tetrahedrons with $\mu_b \leq \mu \leq \mu_d$ are retrieved.

## 5 Other Techniques Requiring a Preprocessing Step

This section reviews other preprocessing techniques that speed up visualization of large data. One group of techniques reviewed [1, 6, 17] speed-up visualization of a large volume of data by splitting it into sub-volumes and storing summary information together with each sub-volume. That enables the application to skip over sub-volumes that are not used in the visualization. We review three papers Rusinkievicz and Levoy [17] simplify processing of data by reading it along the main axes and use compression to reduce its size and skip over unimportant regions, Ahrens et al. [1] split the data in pieces, using a preprocessing step, and build summary information about each piece and use that information to cull out or to change the order of processing of individual pieces and Childs et al. [6] allow individual processing modules to specify what optimization can be done on data before the data reaches the processing module.

Another group [5, 7] speed-up visualization by building a fast search data structure that is used for identifying the sub-volumes used in the visualization.

**Shearwarp**. Lacroute and Levoy [14] present a set of fast volume rendering algorithms based on shear-warp factorization of the viewing transformation. This factorization splits the viewing transformation into a shear parallel to the data slices, a projection to form an intermediate image and a 2D warp to form an undistorted final image (see Figure 3).

The algorithms presented are fast because of two main reasons. First, in a shear-warp factorization rows of voxels in the volume are aligned with rows of pixels in the intermediate image. This is used to create a scan-line based processing of data in which traversal of volume and of intermediate image is done in parallel. Second, spatial coherence in the data and in the intermediate image is exploited by run-length encoding of both data and the intermediate image. This encoding allows the algorithm to skip over transparent voxels in the current data scan-line, and over opaque pixels in the current intermediate image scan-line. This yields an algorithm that renders a $256^3$ volume data in one second.

The algorithm presented by the authors can be used to render a perspective

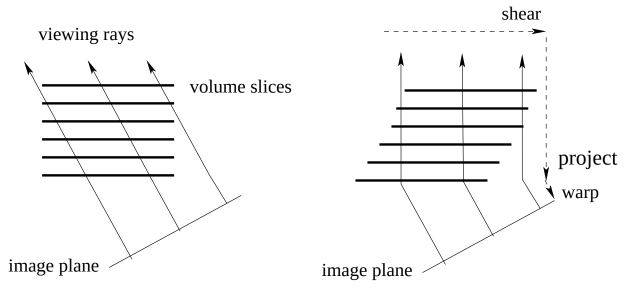

Figure 3: Shear-warp factorization: shear of volume's slices then a projection to form an intermediate image and then a warp to form the final image

view of data, with a small penalty determined by the fact that scan-lines in the intermediate image can cover several scan-lines in the sheared slices of data. Both algorithms for parallel and perspective projections work on classified data, that is, on data where the mapping between an attribute value and the opacity of the data point was already done [15].

A second extension presented is a fast classification algorithm that allows the shear warp algorithm to work directly with the unclassified volume data. The classification algorithm is based on a min-max octree used to encode spatial coherence in unclassified volumes. This data structure and the shear-warp rendering algorithm allows the authors to classify and render a $256^3$ in three seconds.

**Fast search data structures**. Cignoni et al. [7] present an optimal time isosurface extraction algorithm for irregular volume data. The algorithm runs in $O(k + \log h)$ where $k$ is the number of active cells (cells that intersect the isosurface) and $h$ is the number of distinct extrema values (minimum and maximum) associated with cells of the dataset. If $m$ is the number of cells in the dataset, we have that $h \leq 2m$ because each cell has a maximum and a minimum, but different cells may have the same maximum or minimum. The algorithm is based on a data structure called an interval tree which stores intervals of real numbers and supports efficient retrieval of intervals that contain a given value. Each cell of the data set can be associated with an interval $[min, max]$ where $min$ and $max$ is the minimum respectively the maximum value of the cell. Finding all cells that intersect an isosurface with value $q$ is translated into finding all intervals $[min, max]$ that contain $q$. Tests performed by the authors show a speedup of 5 to 10 times for finding the cells that are intersected by the isosurface when compared with the Marching cubes [16] algorithm. Note that Marching cubes has a running time of $O(m)$ where $m$ is the number of cells in the dataset.

Chiang and Silva [5] present an I/O optimal technique for extracting an isosurface from irregular volume data. They combine the ideas of Cignoni et al. [7] for building an interval tree that allows efficient search of all cells intersected by the isosurface with the external-memory interval tree of Arge and Vitter [2]. The authors implement their methods as an I/O filter for VTK's [13] isosurface extraction routine.

**Chunking with summary information**. Ahrens et al. [1] describe a modular visualization architecture designed to speedup visualization of large datasets or enable visualization of out-of-core datasets. The main idea is to apply a preprocessing step that breaks the dataset into smaller pieces and calculates summary information about each piece. Individual pieces are streamed through memory, they are processed through the original visualization steps and the outputs are composed to get the final image. This approach has two main advantages when compared with loading the entire dataset into main memory. First, it enables out-of-core processing of data that does not fit into the main memory. As data is processed a piece at a time, the need to load the entire dataset into main memory is eliminated. The second advantage is that many of the pieces may not be processed by all steps in the visualization process. This is because many pieces may be culled out because they are out of the viewing frustum or they are occluded by other objects in the scene. The authors also describe an improvement to this architecture that is obtained by adding a priority to each individual data piece. This priority can be based on either spatial location of the piece or on data values within the piece. By rendering data pieces with higher priority first, early feedback can be given to the user before the rendering of all pieces finishes.

**Application controlled optimizations**. Childs et al. [6] describe a technique for allowing individual processing modules in a visualization pipeline, to specify optimizations that can be done on the data before it reaches that module. A *contract*, a data structure that travels through the visualization pipeline is used to specify what optimizations can be performed on data. This technique was implemented in VisIt, a visualization tool built by the Department of Energy. VisIt uses a data flow network design for its visualization pipeline. *Data objects* traverse a network of *components* or *processing objects*. The components can be *filters*, *sources* and *sinks*. A source produces a data object, a filter reads a data object, processes it and produces a data object and sinks just read a data object and usually renders it on the screen. Data processing starts at a sink with an update request that propagates up the pipeline through the filters and to a source. The source reads the data and produces a data object which is processed through the filters and is rendered by a sink. An update request can have an associated contract, a data structure that allows a component to specify optimizations that can be done on the data before it reaches it.

Many optimizations depend on data being split into pieces called *domains* and on meta-data for the domains, such as spatial extent and attribute bounds. Each component in the pipeline can specify one of the following optimizations: reading the optimal subset of data, choosing the execution model, generation of ghost data, and sub-grid generation. The contract technique is extensible; more optimizations could be added to those already implemented. Reading the optimal subset of data can be specified by a component such as one that calculates the intersection between the volume of data and a plane. Only the sub-domains intersected by the plane need to be read, and those domains can be determined by using the meta-data (which is always in memory). The execution model specifies how the domains are distributed to processors: one approach called *streaming* processes one domain at a time through the entire processing pipeline; a second approach called *grouping* has one component process all domains before the processing goes to the next component. Ghost data deals with artifacts that may appear because of the splitting of the data volume in domains: one case is when external faces to a domain are internal to the whole volume - these faces are marked as ghost data and should not be rendered; a second case is when interpolation is done at the boundaries of domains - ghost data has to be generated at the boundaries using data from neighboring domains for interpolation to yield correct results. Often, visualization algorithms run faster on rectilinear grids than on unstructured grids, and with the same data, rectilinear grids occupy less memory than unstructured grids. Using these observations, sub-grid generation optimization identifies rectilinear grids in filters' output and separates those grids from the rest of the output which may be an unstructured grid.

## 6 Conclusions

In this paper we review the most important methods and techniques designed to deal with large scientific data. We focus on the most fundamental and persistent themes designed to visualize scientific data. We also identify and define the relevant terminology used in the research literature on this topic. We start with external memory algorithms that can be used as a basis for more complex out-of-core visualization applications. We cover caching and prefetching techniques, multiresolution and adaptive resolution techniques and other techniques requiring a preprocessing step.

Our goal is to provide an introduction or tutorial for those less familiar with the subject and provide references for further study for readers interested in pursuing this subject in more detail.

## 7 Acknowledgments

This research was supported in part by the Research Institute of Visual Computing (rivic.org) Wales.

## References

[1] J.P. Ahrens, N. Desai, P.S. McCormick, K. Martin, and J. Woodring. A Modular Extensible Visualization System Architecture for Culled Prioritized Data Streaming. *Visualization and Data Analysis (VDA), Proceedings of SPIE*, 6495:0I.1–12, 2007.

[2] L. Arge and J.S. Vitter. Optimal Dynamic Interval Management in External Memory. *Foundations of Computer Science, 1996. Proc., 37th Annual Symposium on*, 00:560–569, 1996.

[3] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[4] P. Bhaniramka and Y. Demange. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets. *Proc. of the IEEE Symposium on Volume Visualization and Graphics*, pages 45–54, 2002.

[5] Yi-Jen Chiang, Cludio T. Silva, and William J. Schroeder. Interactive Out-Of-Core Isosurface Extraction. In *IEEE Visualization (VIS)*, volume 0, pages 167–174, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[6] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A Contract Based System For Large Data Visualization. *IEEE Visualization (VIS)*, pages 25–25, 2005.

[7] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal Isosurface Extraction from Irregular Volume Data. In *Proc. of the IEEE Symposium on Volume Visualization*, pages 31–38, Piscataway, NJ, USA, 1996. IEEE Press.

[8] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Multiresolution Representation and Visualization of Volume Data. *Visualization and Computer Graphics, IEEE Transactions on*, 3(4):352–369, 1997.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 18. MIT Press, 2001.

[10] Michael Cox and David Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *IEEE Visualization (VIS)*, pages 235–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[11] K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical Prefetching via Data Compression. *ACM SIGMOD Record*, 22(2):257–266, 1993.

[12] M. de Berg. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[13] Kitware Inc. *The VTK User's Guide Version 5 (Paperback)*. Kitware Inc., 2006.

[14] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. *Computer Graphics*, 28(4):451–458, 1994.

[15] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[16] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[17] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[18] Sunita Sarawagi and Michael Stonebraker. Efficient Organizations of Large Multidimensional Arrays. In *Proc. of the Tenth International Conference on Data Engineering*, pages 328–336, Washington, DC, USA, 1994. IEEE Computer Society.

[19] L.A. Shirmun and S.S. Abi-Ezzi. The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum*, 12(3):261–272, 1993.

[20] Dave Shreiner, Mason Woo, Jachie Neider, and Tom Davis. *OpenGL Programming Guide, Fifth Edition*, chapter 9. Addison Wesley, 2006.

[21] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-Core Algorithms for Scientific Visualization and Computer Graphics, Course Notes for Tutorial 4. In *IEEE Visualization (VIS)*, Boston, MA, USA, 2002. IEEE Computer Society Washington, DC, USA.

[22] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap: A Virtual Mipmap. In *Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 151–158, New York, NY, USA, 1998. ACM.

[23] A.W. Wetzel, B. Athey, F. Bookstein, W. Green, and A. Ade. Representation and Performance Issues in Navigating Visible Human Datasets. In *Proc. Third Visible Human Project Conference, NLM/NIH*, 2000.