Book chapter :
Lipsa, D., Bergeron, R., Sparr, T. & Laramee, R. (2009). *Dynamic Chunking for Out-of-Core Volume Visualization Applications.* Advances in Visual Computing, (pp. 117 Springer.

http://dx.doi.org/10.1007/978-3-642-10520-3_11

Dynamic Chunking for Out-of-Core Volume Visualization Applications

Dan R. Lipsa[1], R. Daniel Bergeron[2], Ted M. Sparr[2], and Robert S. Laramee[3]

[1] Armstrong Atlantic State University, Savannah GA 31411, USA,
[2] University of New Hampshire, Durham, NH 03824, USA
[3] Swansea University, Swansea SA2 8PP, Wales, United Kingdom

Abstract. Given the size of today's data, out-of-core visualization techniques are increasingly important in many domains of scientific research. In earlier work a technique called *dynamic chunking* [1] was proposed that can provide significant performance improvements for an out-of-core, arbitrary direction slicer application. In this work we validate dynamic chunking for several common data access patterns used in volume visualization applications. We propose optimizations that take advantage of extra knowledge about how data is accessed or knowledge about the behavior of previous iterations and can significantly improve performance. We present experimental results that show that dynamic chunking has performance close to regular chunking but has the added advantage that no reorganization of data is required. Dynamic chunking with the proposed optimizations can be significantly faster on average than chunking for certain common data access patterns.

1 Introduction

Disk drive size and processor speed have increased significantly in the last couple of years. This enables scientists to generate ever larger simulation data and to acquire and store ever larger real world data. While visualization is one of the most effective techniques to analyze these large data sets, visualization usually requires loading the entire data into the main memory. Often, this is impossible given the size of data. This problem has led to a renewed interest in out-of-core visualization techniques, which execute the visualization by loading only a small part of the data into the main memory at any given time.

While multidimensional arrays are usually stored in a file using *linear storage*, a common way of improving access to them is to reorganize the file to use *chunked storage*. That happens when data is split in chunks (cubes or bricks) of equal size and the same dimensionality as the original volume, and each individual chunk is stored contiguously in the file using linear storage. Chunks are stored in the file in linear fashion by traversing the axes of the volume in a certain order using nested loops. The order of traversing the axes and the size of the chunks is determined by the expected access pattern. However, chunked file storage typically results in better file system performance (compared to linear storage) even when the access pattern does not match the "expected" one.

Dynamic chunking views a dataset stored using *linear storage* as if it were chunked into blocks of configurable size and shape. As soon as an item in any block is accessed, the entire block is read. Loading a block in memory may require many read operations.

Previously, dynamic chunking was described in the context of a slicer visualization application [1] and it was shown that it provides some of the benefits of file chunking without having to reorganize or maintain multiple copies of the file. In this paper we extend this work to apply to an arbitrary direction slicer and to ray casting. We propose an optimization that can take advantage of further knowledge about the iteration pattern and significantly improve performance. We propose an optimization that can take advantage of knowledge about the behavior of previous iterations to improve performance. We show that dynamic chunking has performance close to that of regular chunking without the need to reorganize the data, and it can be significantly faster in the average with the proposed optimizations.

2 Related Work

The field of out-of-core visualization has a large body of work. Engel et al. [2] present techniques used to render large data on the GPU while Silva et al. [3] present a survey of external memory techniques. In this section we summarize work that is most closely related to ours.

Chunking and related techniques. Sarawagi et al. [4] introduced chunking as a way to improve access to multidimensional arrays stored in files. Many other techniques use chunks as a unit of storage for data [5] or as a unit of communication between a server that contains the data and clients that visualize the data [6]. A technique similar with chunking but applied to irregular grids is the meta-cell technique [7, 8, 9]. Meta-cells contain several spatially close cells, have fixed size (several disk blocks) and are loaded from disk as a unit which enables them to be stored in a space saving format.

Pascucci and Frank [10] describe a scheme to reorganizes regular grid data in a way that leads to efficient disk access and enables extracting multiple resolution versions of the data.

While chunking is very effective and widely used, it has the disadvantage that data needs to be reorganized using chunked storage. Dynamic chunking works directly with a linear storage file, so no reorganization is required. While reading a dynamic chunk from a linear storage file is less efficient than reading a chunk from a chunked file, dynamic chunking can benefit from choosing the dynamic chunk size based on the available memory and iteration parameters.

Caching and prefetching. Common ways to mediate the effect of slow I/O are to use prefetching when the user is thinking what to do as in [11] or to use a separate thread to overlap rendering with data I/O as in [12].

Brown et al. [13] use a compiler to analyze future access patterns in an application and issue prefetch and release hints and use the operating system to manage these requests for all applications. Rhodes et al. [14] use information

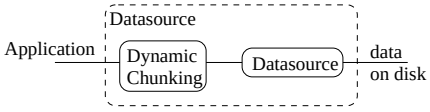

Fig. 1. The Dynamic Chunking module provides the same interface as the Datasource module. This allows for easy integration into an application.

about the access pattern provided by an iterator object to calculate a cache block shape that reduces the number of reads from the file. Chisnall et al. [15] use data inferred from previous accesses to an out-of-core octree, to improve out-of-core rendering of a point dataset using discrete ray tracing.

Cox et al. [16] speed up out-of-core visualization of Computational Fluid Dynamics using application controlled demand paging. This works similarly with memory mapped files but with the additional advantage that the application can specify the page size and it can translate from an external storage format. Dynamic chunking differs from this work in how we decide which data is chosen to be prefetched and cached. The dynamic chunking module prefetches blocks with the same dimensionality as the data, regardless of where that data is stored on disk. Multiple application-level read operations are required to load such a block, but most of those reads would have occurred eventually anyway.

3 Dynamic Chunking Validation

We use dynamic chunking [1] to speed-up out-of-core execution for two visualization applications using four different data access patterns. One pattern is implemented in an arbitrary direction slicer application that reads slices from a volume of data and composes them to build a maximum intensity projection (MIP) [2] representation of the volume. A ray casting application supports three different access patterns: it reads individual voxels, blocks of size 2^3 voxels and blocks of size 4^3 voxels along rays and builds either a MIP representation or a volume rendering [17] representation of the volume.

Our applications access 3D data through the Granite library [14] datasource which hides the location of data from its users. Our applications can access data from main memory or from disk without any change in the source code. A datasource is conceptually an n-dimensional volume of voxels where each voxel can store one or more attributes. Our dynamic chunking module uses a datasource to read data from disk and the module, together with the datasource it reads from, is encapsulated behind a datasource interface as showed in Figure 1.

This enables us to switch between accessing data from main memory, accessing data from disk and accessing data from disk through dynamic chunking without changing the application source code or the data representation. An application could easily be converted to use dynamic chunking by using a datasource for all data accesses.

Lipsa et al. [1] describe dynamic chunking in detail. We summarize the the main idea here. We view the entire volume as composed of blocks of configurable size and shape and we create a block table that stores a reference to each of these blocks. Each reference can point to a block from the volume which has the same number of dimensions as the original volume or it can be nil. Loading a block is done on demand, as soon as any element from the block is needed. We use the Least Recently Used (LRU) block replacement algorithm to maintain cache relevance. We apply dynamic chunking to regular grid data stored using conventional linear storage. We use dynamic chunking to speed up out-of-core execution for two visualization applications: a slicer application and a ray casting application.

Slicer application. The slicer application builds a MIP representation of a subvolume by using a *slice iterator*. This iterator allows a user to specify a subvolume of the data and a vector which represents a plane normal. We move the plane along its normal, using a unit step, for as many iterations as the plane touches the subvolume. At every step, we determine the voxels in the intersection of the subvolume and the plane. This intersection polygon is computed using a 3D scan-conversion algorithm for polygons [18]. We use nearest-neighbor interpolation to determine voxels that form the intersection polygon. We read them from the file and we store them as a texture. Textures are composed using graphics hardware to obtain a final image which is the MIP representation of the volume viewed from the direction specified by the vector normal to the plane.

Ray casting application. We implemented a ray casting application and used it to generate a MIP image or a volume rendering [17] representation of a subvolume. The subvolume is an arbitrarily oriented cuboid (rectangular parallelepiped) subset of the entire dataset. The ray casting application accesses the data file by reading equally spaced voxels along each ray such that the distance between any of these voxels is equal to the distance between two voxels.

For the MIP image we use either nearest neighbor interpolation or trilinear interpolation to determine data values along the ray. These three possibilities: MIP with nearest neighbor interpolation, MIP with trilinear interpolation and volume rendering require three different access patterns through the data file: reading voxels, reading blocks of size 2^3 voxels and reading blocks of size 4^3 voxels along each ray. For each of the three access patterns, the distance between two sample points along a ray is the same as the distance between two neighboring voxels. The resolution of the image produced is determined by the resolution of the data, for instance a volume of size 256^3 voxels will produce an image of size 256^2 pixels.

Volume rendering [17] is implemented by compositing the color and opacity of equally spaced points along a viewing ray. For each point along a ray we read a 4^3 block of voxels from the data file, which is used to calculate the color and opacity at the current position in the ray.

While the slicer and the ray casting applications do not load into the main memory the visualized subvolume, they cache blocks of data through the dynamic chunking module. Denning [19] defines the *working set* of a program as

"the smallest collection of information that must be present in main memory to assure efficient execution" of the program. For the slicer application, the working set consists of all blocks that cover the current slice in an iteration. For the ray casting application, the working set consists of all blocks that completely enclose all ray segments along one rectangular side of the iteration cuboid.

4 Dynamic Chunking Optimizations

Dynamic chunking is a general technique because it does not know about a particular data access pattern the application may want to use. If the application makes this information known, or if the module can infer this information, further optimizations can be applied that can improve performance.

These optimizations are based on the observation that a larger block size results in better performance for dynamic chunking because of larger reads from disk. At the same time, increasing the block size results in a larger cache memory required to store the working set of the application and a block size too large can result in cache thrashing. Our goal is to find the maximum block size that allows us to store the working set of the application in the available physical memory. Our optimizations work when the iteration subvolume is aligned with the principal axes. If the iteration subvolume is not aligned with the principal axes, larger cache blocks result in reading more data that is not used by the application. In this case increasing the cache blocks size may not increase performance of the iteration.

We present two block size optimizations: analytical and adaptive. The analytical optimization uses information provided by the application to calculate the optimal block size for certain iteration patterns while the adaptive optimization uses information gathered from previous iterations to optimize the block size used by the dynamic chunking module.

Analytical block size optimization. We present an algorithm that finds the maximum block size for the dynamic chunking module such that we can store all blocks that cover a slice in memory. While our algorithm works with either a slicing application or a ray casting application for which the iteration subvolume is aligned with the principal axes, the iteration direction and the view is arbitrary.

For simplicity we present our algorithm in 2D, but the same reasoning can be applied in 3D. A slicing application provides two extra parameters to the dynamic chunking module: the iteration subvolume and the orientation of the iteration slice. Our module then calculates the maximum block size that can be used by the dynamic chunking module that avoids cache thrashing.

Suppose that the iteration area (see Figure 2) has an edge size of l and that the iteration area is partitioned by the dynamic chunking module into $n \times n$ squares. Suppose that the angle between the iteration line and the horizontal axis is α and that the amount of available memory is M. We want to minimize n (this in turn will maximize the square size) such that the working set of the application (the squares that cover the iteration line) still fits in the amount of

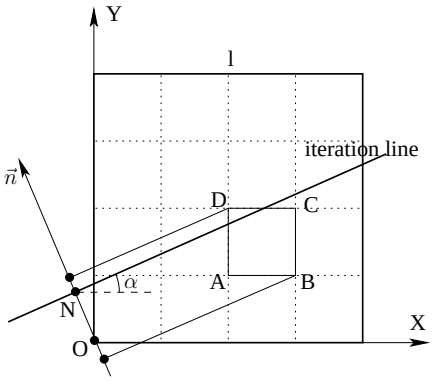

Fig. 2. An area with edge length l partitioned in n^2 squares where $n = 4$. The intersection between square $ABCD$ and the iteration line can be determined by looking at the projections of the corners of the square on the normal to the iteration line.

available memory M. We can assume that α is between $0°$ and $45°$; all other angles can be treated similarly through symmetry.

We denote with I_B the maximum number of squares intersected by the iteration line as it traverses the whole subvolume. Note that the number of squares intersected by the iteration line in Figure 2 varies through the iteration. It starts with one square at the beginning of the iteration, then it increases and then it decreases back to one square at the end of the iteration. We denote with M_B the size (area) of a square where $M_B(n) = \frac{l^2}{n^2}$. Our goal is to find the maximum square size, or equivalently to find the minimum n, such that all squares intersected by the iteration line can be stored in the available memory M. That can be done by using a `for` loop that starts with $n = 1$ and tests at each iteration if $I_B * M_B < M$. If the test is true, then we have found n and, in turn, the square size; if it is not true, we continue by doubling n (which means that we decrease the square size by a factor of 4) until a minimum square size when we give up. In that case we do not have enough memory to optimize the execution of the application through dynamic chunking.

So, the only problem left is to find I_B the maximum number of squares intersected by the iteration line as it traverses the subvolume. We start by looking at the intersection between the iteration line and a square from the partition of the subvolume. We can deduce that an iteration line sliding along its normal $\boldsymbol{n}$ (which corresponds to angle $\alpha \in [0, 45]$ degrees) intersects the square with the left-lower corner at position (i, j) if

$$d \in (\frac{l}{n}(n_x(i+1) + n_y j),\ \frac{l}{n}(n_x i + n_y(j+1)))$$

where d is the distance from the origin to the line $d = \|ON\|$, n is the number of squares per edge of the iteration area, l is the iteration area edge size, n_x and

n_y are the components of the normal to the iteration line $\boldsymbol{n}$. A similar result can be deduced in 3D.

For each of the n^3 blocks, we have an interval that gives us the position of the slice that intersects the block. Overlapping intervals give us positions of the slice that intersects several blocks. We can find the maximum number of overlapping intervals by creating a sorted list with the left and right ends of all intervals. We initialize I_B, the maximum number of blocks intersected by the slice, to 0. We traverse the sorted list of interval ends and we execute the following: if we see a left end of an interval we increment I_B, if we see a right end of an interval we decrement I_B. While we do that we keep track of the maximum value for I_B. That maximum value is the maximum number of blocks intersected by the slice as it traverses the subvolume.

To overcome large time requirements ($n^3 \log n$) of this algorithm, especially for partitions with more blocks than 64^3, we note that the maximum number of blocks intersected by a slice depends only on the direction of iteration and on the number of blocks in the partition of the subvolume, and it does not depend on the data itself. In our implementation the maximum number of blocks intersected by a slice are calculated and stored in the application for expected directions of iteration and number of blocks in the partition of the subvolume.

Adaptive block size optimization. For iteration patterns for which the working set is not a slice but a more complex shape, such as a part of a sphere, or for applications that do not provide the required additional information, it is not possible to analytically calculate the biggest block size that does not cause cache thrashing for a certain memory size. In this case the dynamic chunking module can learn from the behavior of past iterations and adjust the block size for best performance.

The application provides a unique identifier for the iteration pattern used. This identifier allows the dynamic chunking module to keep track of previous iterations and decide if a certain block size worked well for a certain iteration and memory size. The identifier differentiates between different working set shapes, working set sizes and cache memory sizes.

The dynamic chunking module starts with a default block size, eventually provided by the application. While the application executes an iteration, the dynamic chunking module tests for cache thrashing by keeping track of the blocks used. If a block is discarded, and then reused we conclude that the working set does not fit in memory and so the cache is thrashed.

If an iteration thrashes the cache for a certain block size, the iteration is restarted with a smaller block size that has half the edge size of the original block. We continue this process until we complete an iteration without thrashing or a minimum block size is reached. If the minimum block size is reached, we have too little memory to use dynamic chunking. If we complete an iteration without thrashing, the dynamic chunking module stores the block size for the current identifier together with an *optimal* flag which signals that no other block size besides the one stored will be tried for future iterations.

If an iteration completes without thrashing for a certain block size, the dynamic chunking module stores the block size for the current identifier. A future iteration tries to use a larger block that has an edge size double the edge size of the stored block, and if that completes without thrashing, the block size will be stored for the current identifier. The process continues until a block size results in cache thrashing, in which case the block size with edge size half the edge size of the original block is stored for the current identifier together with the *optimal* flag.

This algorithm results in progressively larger block sizes used for the dynamic chunking module, which results in progressively better performance. If the initial block size provided by the application does not result in thrashing, the only price paid for this improved performance is one iteration that will thrash the cache, which will be quickly detected by the fact that a block discarded is being reused. If the initial block size provided by the application results in thrashing, several iterations that result in thrashing are possible until a block size that does not result in thrashing is found or the dynamic chunking module gives up. If a block size that does not result in thrashing is found, that block size is optimal.

5 Results

To test the dynamic chunking module we traversed a subvolume of size 256^3 voxels with three bytes per voxel (the subvolume has 48 MB) located inside a data volume of size 1024x1216x2048 voxels (the data volume has 7.2GB) with 25 MB allocated to the dynamic chunking module for caching. The subvolume is centered at $(256, 256, 256)$ and we vary either the orientation of the slice for the slicer application or the orientation of the subvolume for ray casting.

Our tests measure traversal time through the subvolume for 84 orientations specified by the following pairs of heading and pitch Euler angles [20]: $\{0, 30, \ldots, 330\} \times \{-90, -60, \ldots, 90\}$. We summarize all times required for a traversal for all possible orientation angles using a box plot. For the ray casting application, the orientation of the iteration cuboid uses a bank angle of 20 degrees. For the slicer application, the orientation of the normal to the slice can be specified only using heading and pitch angles.

Dynamic chunking improves the time required to read data for various access patterns used in visualization algorithms. Our tests measure only the time required to read the data when using the same access pattern as the visualization application. Note that we set the Java Virtual Machine memory to 30MB (using `-Xms` and `-Xmx` switches) and we set the cache memory to 25MB, both less than the size of the subvolume traversed, so our visualization applications and our tests run with data out-of-core.

Before each run of a traversal for a particular orientation, we clear the file system cache by running a separate program called *thrashcache*. For the Linux operating system, this program `umount`s and then `mount`s the file system that contains the data.

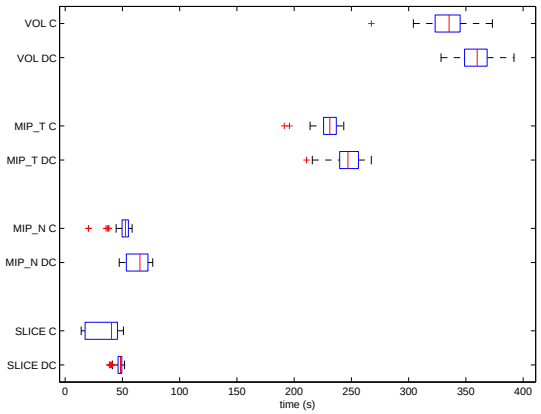

Fig. 3. Data read time for dynamic chunking (DC) versus chunking (C) optimizations for four access patterns: slicing (SLICE), ray casting with nearest neighbor interpolation (MIP_N), ray casting with trilinear interpolation (MIP_T), and volume rendering (VOL).

In the graphs presenting our test results, we use acronyms for the data access patterns tested: SLICE for the slicer application, MIP_N for the ray casting application that calculates a MIP image and uses nearest neighbor interpolation, *MIP_T* for the ray casting application that calculates a MIP image and uses trilinear interpolation and *VOL* for volume rendering. We also use acronyms for the optimization technique used when the test is run: *DC* for dynamic chunking, *C* for chunking, *FS* for file system cache only.

Our test machine has a dual Intel Xeon at 3GHz processor, with 512KB L2 cache and it has 1GB of RAM. The disk drive has rotational speed of 7200 RPM, it has 8.5 ms average seek time and 2 MB of cache. The machine is running Fedora Core 5 GNU-Linux Operating System and Java 1.5.0. All our applications are built in Java, using Java binding to OpenGL (JOGL) [21] for rendering. This makes our application platform independent.

The results presented report only the time required to read the data, the time needed for the actual visualization is not included in the results. This makes our results relevant to any implementation of a visualization algorithm that accesses data using one of the patterns tested.

Dynamic chunking versus chunking. Figure 3 displays the time to read a 48MB subvolume from a 7.2GB chunked file and from a 7.2GB file with data stored using linear storage. The box plots labeled C shows the time to read the subvolume from a chunked file. The graphs labeled DC shows the time to read the subvolume from a linear file using dynamic chunking with block size 16^3 voxels.

The file is chunked with chunks of size 16^3 voxels, the same as the size of pages in the paging module. Paging over a chunked file works about 17% faster

than dynamic chunking over a linear file, but it requires reorganization of the file. The big speed advantage that would be expected from chunking, is not seen because of the file system and disk drive cache which are able to avoid many actual hard drive operations. For example, when the dynamic chunking module loads a block of size 16^3 voxels, from 16x16 read operations sent to the file system many of them are served from the file system or hard drive cache.

Block size optimizations. To test the block size optimizations we ran the same tests as before for the SLICE access pattern, but we turn on either the analytical block size optimization or the adaptive block size optimization.

For the analytical optimization the block size used by the dynamic chunking module is determined before the iteration based on the information provided by the application (iteration subvolume and normal to the iteration plane), and information determined from the system (the amount of available memory). Figure 4 shows that we get about 25% better performance than chunking in the average, while chunking is still better for certain traversal angles that match the way chunks are stored in the file.

For testing the adaptive optimization we still use the SLICE access pattern but we do not provide the extra information needed to calculate the best block size for the given amount of cache memory. In this case, the block size used by the dynamic chunking module is determined from knowledge gathered from previous iterations. The graph for the adaptive block size optimization shows the mean of ten iterations for each iteration angle. For iterations that are interrupted because of cache thrashing we add the iteration time but we do not count the iteration when we calculate the mean.

As an example, we present the sequence of block sizes used for iteration direction (heading, pitch) = (0,-90), in the ten iterations tested. The first iteration uses block size 16^3 (supplied by the application), and then the adaptive optimization adjusts that to 32^3 and then 64^3. For block size 128^3 cache thrashing occurs which signals that the optimal block size is 64^3. Cache thrashing is quickly detected from the fact that a block discarded is reused by the application. The rest of 6 iterations are run with the optimal block size 64^3. Other iteration directions behave similarly the only difference being the block size where thrashing occurs, which in turn determines the optimal block size.

6 Conclusions and Future Work

We have presented performance tests for dynamic chunking, a technique that can speed-up common data iterations used in volume visualization algorithms. Our test results show that dynamic chunking performs 5.3 times better than file system cache alone and that by using dynamic chunking over a linear file yields about 83% performance of that of paging over a chunked file.

We presented optimizations that can improve performance further if additional information is either provided by the application or infered by the dynamic chunking module.

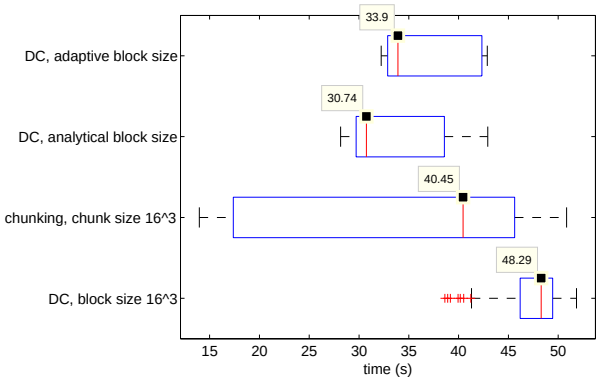

Fig. 4. Data read time for traversing a 48MB subvolume from a 7.2GB volume with a cache of 25MB. We test the SLICE iteration pattern for four optimizations: dynamic chunking (DC) with fixed blocks of size 16^3 voxels, chunking with chunks of size 16^3 voxels, dynamic chunking with analytical block size optimization and dynamic chunking with adaptive block size optimization. For adaptive block size optimization we show the average of ten iterations.

In the future, we plan to explore other possible dynamic chunking optimizations that can be applied to different iterations and visualization applications. We also plan to investigate different ways to get more information from the application about the access pattern.

7 Acknowledgments

We would like to thank Radu G. Lipsa for a valuable discussion on the block size optimization. This research was partially funded by the Welsh Institute of Visual Computing (WIVC).

References

1. Lipsa, D.R., Rhodes, P.J., Bergeron, R.D., Sparr, T.M.: Spatial Prefetching for Out-of-Core Visualization of Multidimensional Data. In: Proc. of SPIE, Visualization and Data Analysis. Volume 6495–0G., San Jose, CA, USA (2007) 1–8
2. Engel, K., Hadwiger, M., Kniss, J.M., Lefohn, A.E., Salama, C.R., Weiskopf, D.: Real-Time Volume Graphics, Course Notes. In: Proc. of ACM, SIGGRAPH, New York, NY, USA, ACM Press (2004) 29
3. Silva, C., Chiang, Y., El-Sana, J., Lindstrom, P.: Out-of-Core Algorithms for Scientific Visualization and Computer Graphics, Course Notes for Tutorial 4. In: IEEE Visualization, Boston, MA, USA, IEEE Computer Society Washington, DC, USA (2002)

4. Sarawagi, S., Stonebraker, M.: Efficient Organizations of Large Multidimensional Arrays. In: Proc. of the Tenth International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (1994) 328–336
5. Chang, C., Kurc, T., Sussman, A., Saltz, J.: Optimizing Retrieval and Processing of Multi-Dimensional Scientific Datasets. In: Proc. of the Third Merged IPPS/SPDP Symposiums, IEEE Computer Society Press (2000)
6. Wetzel, A., Athey, B., Bookstein, F., Green, W., Ade, A.: Representation and Performance Issues in Navigating Visible Human Datasets. In: Proc. Third Visible Human Project Conference, NLM/NIH. (2000)
7. Chiang, Y.J., Silva, C.T., Schroeder, W.J.: Interactive Out-Of-Core Isosurface Extraction. In: IEEE Visualization. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (1998) 167–174
8. Chiang, Y.J., Farias, R., Silva, C.T., Wei, B.: A Unified Infrastructure for Parallel Out-of-Core Isosurface Extraction and Volume Rendering of Unstructured Grids. In: Proc. of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics, Piscataway, NJ, USA, IEEE Press (2001) 59–66
9. Farias, R., Silva, C.T.: Out-Of-Core Rendering of Large, Unstructured Grids. IEEE Computer Graphics and Applications **21** (2001) 42–50
10. Pascucci, V., Frank, R.J.: Global Static Indexing for Real-Time Exploration of Very Large Regular Grids. In: Supercomputing '01: Proc. of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), New York, NY, USA, ACM Press (2001) 2–2
11. Doshi, P., Rundensteiner, E., Ward, M.: Prefetching for Visual Data Exploration. Proc. Eighth International Conference on Database Systems for Advanced Applications **8** (2003) 195–202
12. J.Gao, J.Huang, Johnson, C., S.Atchley: Distributed Data Management for Large Volume Visualization. In: IEEE Visualization. (2005)
13. Brown, A., Mowry, T.: Compiler-Based I/O Prefetching for Out-of-Core Applications. ACM Trans. on Computer Systems **19** (2001)
14. Rhodes, P.J., Tang, X., Bergeron, R.D., Sparr, T.M.: Iteration Aware Prefetching for Large Multidimensional Scientific Datasets. In: SSDBM'2005: Proc. of the 17th International Conference on Scientific and Statistical Database Management, Berkeley, CA, US, Lawrence Berkeley Laboratory (2005) 45–54
15. Chisnall, D., Chen, M., Hansen, C.: Knowledge-Based Out-of-Core Algorithms for Data Management in Visualization. In: EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization. (2006) 107–114
16. Cox, M., Ellsworth, D.: Application-Controlled Demand Paging for Out-of-Core Visualization. In: IEEE Visualization, Los Alamitos, CA, USA, IEEE Computer Society Press (1997) 235–ff.
17. Levoy, M.: Display of Surfaces from Volume Data. IEEE Computer Graphics and Applications **8** (1988) 29–37
18. Kaufman, A., Shimony, E.: 3D Scan-Conversion Algorithms for Voxel-Based Graphics. In: SI3D '86: Proc. of the 1986 Workshop on Interactive 3D Graphics, New York, NY, USA, ACM Press (1987) 45–75
19. Denning, P.J.: The Working Set Model for Program Behavior. Commun. ACM **11** (1968) 323–333
20. Dunn, F., Parberry, I.: 3D Math Primer for Graphics and Game Development. Wordware Publishing Inc (2002)
21. java.net: Java Bindings for OpenGL (JSR-231) (2008) online document, `https://jogl.dev.java.net/`.