Spatial Prefetching for Out-of-Core Visualization of Multidimensional Data.

Dan R. Lipsa[a] and Philip J. Rhodes[b] and R. Daniel Bergeron[c] and Ted M. Sparr[d]

[a]Department of Information Technology, Armstrong Atlantic State University, Savannah, GA;
[b]Department of Computer and Information Science, University of Mississippi, Oxford, MS;
[c,d]Department of Computer Science, University of New Hampshire, Durham, NH;

ABSTRACT

In this paper we propose a technique called *storage-aware spatial* prefetching that can provide significant performance improvements for out-of-core visualization. This approach is motivated by file *chunking* in which a multidimensional data file is reorganized into multidimensional sub-blocks that are stored linearly in the file. This increases the likelihood that data close in the n-dimensional volume represented by the file will be closer together in the physical file. Chunking has been demonstrated to improve the typical access to such data, but it requires a complete re-organization of the file and sometimes efficient access is only achieved if multiple different chunking organizations are maintained simultaneously. Our approach can be thought of as *on-the-fly* **chunking**, but it does not require physical re-organization of the data or multiple copies with different formats. We also describe an implementation of our technique and provide some performance results that are very promising.

Keywords: out of core visualization, volume visualization, chunking

1. PROBLEM AND MOTIVATION

We define scientific data as multidimensional data obtained either from the real world or from a simulation. Two characteristics which are important for our research are that this data is large in size and the retrieval is often "volumetric",[1] meaning that data retrieval is done by querying a sub-volume of data.

As computer processing power and memory size have increased dramatically in recent years, scientists are able to generate ever larger data sets. Today's scientific data are measured in terabytes or larger. The sensors used to acquire data are becoming more and more sophisticated and the computers used to generate simulation data are becoming more and more powerful. Therefore, the size of the data that we need to process is likely to keep increasing.[2] It is still the case that visualization is one of the most effective techniques for analyzing these data sets and understanding the physical phenomena that the data represents. Unfortunately, however, the huge size of the data creates significant bottlenecks in the visualization process. Very often, the data to be rendered is much too large to fit into main memory at once. This phenomenon has lead to increasing interest in *out-of-core visualization* algorithms. *Interactive* out-of-core visualization is particularly challenging since the generation of the next image may have to wait for a significant amount of time for needed data to be read from disk.

The past twenty years has seen a thousand-fold increase in processor speed,[3] memory size and hard disk size. Disk I/O performance improvements, however, lag far behind the progress made in all other areas of scientific data processing. In addition, 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.

Scientific data can be stored in files using a variety of methods, the most common being linear storage and chunked storage. A multidimensional array is stored in a file using *linear storage* by traversing its indexes (axes)

in predefined order using nested loops.[4] A multidimensional array is stored in a file using *chunked storage* when data is split in chunks (cubes or bricks) of equal size, 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.

The file system cache usually prefetches and stores data that is nearby *in the file*, to the data retrieved by a "read" command. However, that data might not be nearby *in the volume* represented by the file. Therefore the file system prefetching and caching does not prefetch or store data immediately required by a volumetric retrieval method. Due to this fact, prefetching and caching methods used by the file system are not usually effective for volumetric scientific data retrieval.

2. BACKGROUND AND RELATED WORK

2.1. Chunking

Sarawagi et al. have explored ways to improve the access to multidimensional arrays stored in files.[4] They reorganize the files using chunked storage and maintaining several copies of the data that would match each access pattern. While this method is very effective and widely used, it has the disadvantage that data needs to be reorganized or copies of the data need to be made with different organization. Because today's scientific data is measured in gigabytes or even terabytes, it is often impractical to store multiple copies of the data with different organizations.

Following this work, many other reorganization methods have been developed. More and Choudary reorganize their data according to the expected query type, and the likelihood that data values will be accessed together.[5] The Active Data Repository uses chunking to reduce overall access costs and to achieve balanced parallel I/O.[6,7]

The Volume Server/Browser from the Pittsburgh Supercomputer Center provides a multi-user remote visualization service for the Visible Human data sets.[8] The 40-60GB data sets are pre-processed in a variety of ways to extract multiple resolution levels and to compress the remaining data so that most of the needed data can be held in main memory in the data server.

2.2. Prefetching

Prefetching has also been an active area of research for some time. Cao et al. allow applications to have control of data cache replacement strategy in their share of cache blocks.[9] Brown et al. describe a hint based method that effectively accelerates paged virtual memory performance using an operating system that takes advantage of compiler generated hints and multiple disks.[10] Kotz describes disk directed I/O, a method for aggregating and prefetching data requests in a parallel environment.[11] Mowry presents software controlled prefetching for hiding or reducing the latency experienced by a processor accessing memory.[12] Gao et al. describe a prefetching cache designed for large volume visualization in a distributed context.[13]

Rhodes et al. use *iteration aware prefetching* to speed-up a point or block iteration along one of the principal axes.[14] The authors introduce the idea of an n-dimensional cache block that has the same number of dimensions as the volume represented by the file. Using the information about the access pattern provided by an iterator object, and the fact that the iterator progresses along one of the principal axes, the authors can calculate a cache block shape that reduces the number of reads from the file and contains all the data needed in the current and future iteration steps. The time required to perform the iteration is greatly reduced because the number of disk read operations is reduced.

2.3. Out-of-core visualization

Cox et al. use visualization of Computational Fluid Dynamics to test various ways of managing data that is larger than physical memory.[15] 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.[15]

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[15] 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. 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 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 volumetric nature of processing in not exploited.

3. APPROACH

3.1. Storage-aware spatial prefetching

The goal of our research is to provide some of the benefits of file chunking without having to reorganize or maintain multiple copies of the file. Our approach is to dynamically simulate the read behavior that would occur with a chunked file. In other words, assume that we have a three-dimensional data set stored in conventional linear order that we want to treat as if it had been chunked into 16x16x16 blocks. When the application initially reads a single data record, our spatial prefetching module actually reads the entire block that contains the record and saves the entire block in its local cache storage. In order to read the block, we issue 256 read requests for 16 data records each. Because of the effectiveness of today's file system caching algorithms, there will typically be many fewer physical read operations to satisfy the 256 application-level read invocations.

We call this approach *storage aware spatial prefetching*, where *spatial* refers to the multi-dimensionality of the blocks and *storage aware* refers to the fact that the n-dimensional blocks are loaded from disk in a way that most closely matches the ordering of the data in the file.[14] That is, we read blocks from the file by reading along the most frequently varying axis to the least frequently varying axis.

We built our module on top of the *datasource* component of the Granite system.[14] A datasource is conceptually an n-dimensional volume of voxels where each voxel can store one or more attributes. Our spatial prefetching module splits the entire datasource into blocks of configurable size and creates 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 a voxel from the block is needed. We use the LRU block replacement algorithm to maintain cache relevance.

3.2. Test application

To test the effectiveness of our approach, we created an application that builds an image of a volume using the maximum intensity projection (MIP) algorithm.[16] Our application was built in Java, using JOGL library, which is a Java binding to OpenGL.

We implemented the MIP algorithm by using a *slice iterator*. This iterator allows a user to specify a sub-volume of the data and a vector which represents a plane normal. We step the plane along its normal for as many iterations as the plane touches the sub-volume. At every step, we determine the voxels in the intersection of the sub-volume and the plane. This intersection polygon is computed using a 3D scan-conversion algorithm for polygons.[17] We use nearest-neighbor interpolation to determine values that form the intersection polygon, and we store the intersection polygon as a texture. Intersection polygons are composed using graphics hardware to obtain the final image that represents the volume to be viewed from the direction specified by the vector normal to the plane.

To speed up our application, the memory available to the spatial prefetching module needs to be able to contain enough blocks to completely enclose the intersection polygon.

In our work we use multidimensional prefetch blocks to store blocks read from a multidimensional file. Unlike the work by Cox et al.[15] our blocks are not linear, one dimensional pieces of the file, but they are sub-blocks of the volume represented by the file. Note that several read operations might be needed to read the data stored in a memory block.

Unlike the work by Rhodes et al.[14] we don't use advanced knowledge about the iteration to calculate and prefetch a unique cache block that contains data for the current and future iteration steps. While the knowledge about the iteration is very useful for improving a point or block iteration parallel to one of the axes,[14] we present a general method for improvement when the iteration pattern is too complex for computing in advance the size and shape of the prefetch block.

4. RESULTS AND CONTRIBUTIONS

For our tests we traversed a sub-volume of size 256^3 voxels with three bytes per voxel (the sub-volume has 48 MB) located inside a data volume of size $1024 \times 1216 \times 2048$ voxels (the data volume has 7.2GB). The sub-volume has its lower corner at $256 \times 512 \times 768$. We traversed the sub-volume by sliding a plane along its normal. At each step we calculated the current slice through the volume, and we composed that slice with the current MIP image of the sub-volume.

The data volume is stored in a file using linear storage. If we denote with 0, 1 and 2 the orthogonal axes of the volume, data is laid out in the file using three nested loops, the outer loop along axis 0, then along axis 1 and the inner loop along axis 2. Rhodes et al. define a *rod* as a one dimensional sequence of elements that are contiguous in both the n-dimensional index space and the 1-dimensional file space.[14] Using this terminology we can describe the file as storing rods parallel with axis 2. Several of these rods form planes orthogonal to axis 0. Several of these planes form the entire volume.

We obtain the direction of iteration (the normal to the iteration plane) by using two Euler angles: heading and pitch as described by Dunn and Parberry.[18] We start with an identity direction (heading,pitch)=(0,0) in the direction of axis 2. From there we apply heading rotation and then pitch rotation to get to the direction we are describing. Heading measures the amount of rotation about axis 1, positive rotation is counter-clockwise when axis 1 points toward the viewer. Pitch measures the amount of rotation about axis 0, counter-clockwise when axis 0 points toward the viewer. For example (heading, pitch) (0,0) represents axis 2, (90,0) represents axis 0, (-90, 0) represents a vector parallel with axis 0 but in opposite direction, (0, -90) represents axis 1.

Our tests measured traversal time for 40 plane orientations specified by the following pairs of heading and pitch angles: $\{0, 45, 90, 135, 180, 225, 270, 315\} \times \{-90, -45, 0, 45, 90\}$. Before each run of a traversal for a particular plane orientation, we clear the file system cache by running a separate program called *thrashcache*. This program allocates a block as big as the amount of available physical memory, loads that memory with data from a file different than the file used for our tests and then frees the memory. This ensures that we start each test with a file system cache that does not contain any data left over from previous test runs using the same data set.

The test machine is a 2.5GHz dual processor Power Macintosh G5 with 2GB of RAM running OS X 10.4 and Java 1.5.0. The hard disk is a 7200 RPM SATA model with a 9 ms average seek time and 8 MB of cache.

4.1. File system cache versus spatial prefetching

Figure 1 displays the time we need to build a MIP representation of a 48MB sub-volume from a 7.2GB data volume. This is approximately the same as the time required by the slice iterator to traverse the sub-volume, because composing the slices in hardware takes negligible time. Note that we set the Java Virtual Machine memory to be 20MB (using `-Xms` and `-Xmx` switches), less than the size of the sub-volume traversed.

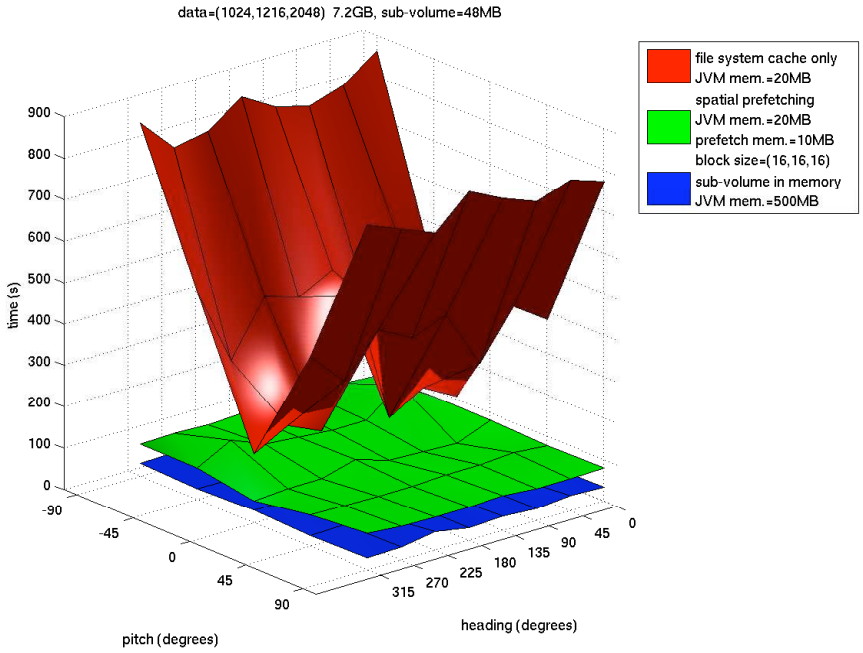

Figure 1. File System Cache Versus Spatial Prefetching

We tested three cases:

file system cache only In this case traversal of the sub-volume is done without spatial prefetching (only the file system cache is used)

spatial prefetching In this case traversal of the sub-volume is done with 10MB allocated to the spatial prefetching module and blocks of size (16,16,16).

sub-volume in memory In this case traversal of the sub-volume is done with the entire sub-volume in memory. We do this by using a unique block as large as the sub-volume and by setting Java Virtual Machine memory large enough so that the sub-volume fits in memory (500MB). This option represents the absolute best possible performance.

As expected, traversal time is worst when only the file system prefetching and caching is used. This is shown in the graph in Figure 1 labeled "file system cache only". The directions for which the file system cache performs well are around (heading,pitch)=(90,0) and (270,0). These directions correspond to a plane orthogonal to axis 0. In this case, data prefetched by the file system is part of the next iteration, so this data is read from cache, and few read operations are issued to the hard disk.

The graph in Figure 1 labeled "spatial prefetching" displays the running time of a sub-volume traversal when spatial prefetching is used. On average, traversal time is 6.9 times better when spatial prefetching is used than traversal time when only the file system cache is used.

The graph in Figure 1 labeled "sub-volume in memory" displays the running time of a sub-volume traversal when the sub-volume is loaded in memory. This is done by using spatial prefetching with a block size equal to the sub-volume size.

This "optimal" performance is about 2.5 times better on average than spatial prefetching with a block size (16,16,16).

4.2. Block shape variation

Figure 2 shows how traversal time of a linear file with spatial prefetching varies with the shape of the block.

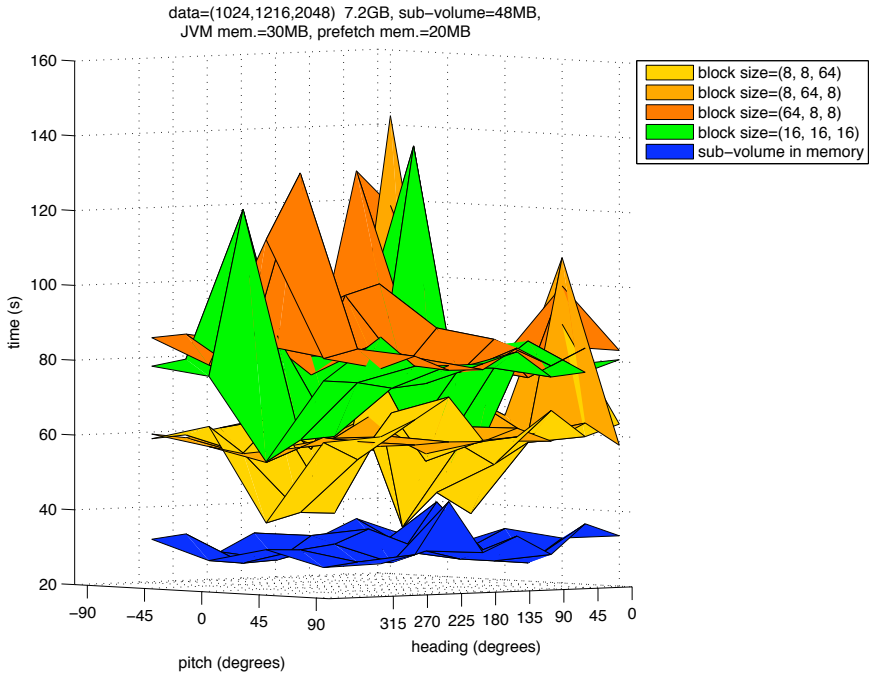

Figure 2. Block shape variation

We tested three block shapes (8,8,64), (8,64,8), (64,8,8) such that the volume of a block is the same as the volume of a block used in the previous tests: (16,16,16). Different block shapes yield different traversal times, with the best shape being (8,8,64) which results in 22% better performance than block shape (16,16,16). The reason for this improvement is the fact that these blocks are longer in the rod direction. To read a block of size (8,8,64) we issue $8 * 8 = 64$ read operations where each operation reads 64 bytes. To read a block of size (16,16,16) we issue $16 * 16 = 256$ read operations where each operation reads 16 bytes. Since fewer reads are issued in the first case, the cost of positioning the head over the hard disk is payed fewer times, so performance is improved.

The relationships of the performance of the other shapes is a bit more complicated. Shapes (8,64,8) and (64,8,8) both result in $8 * 64 = 512$ reads of 8 bytes each. However, the file system prefetching is very aggressive, so that a read for a block of shape (8,64,8) is very likely to result in a file system read that is large enough to contain 1 or more rods that are requested in subsequent reads. This data gets cached by the file system resulting in future application reads that do not result in a physical read. In the (64,8,8) case, however, the needed rods are so far apart in the physical file that many fewer application reads will hit the file system cache.

4.3. Spatial prefetching versus chunking

Table 1 shows how spatial prefetching over a linear file compares with access to a chunked file and to spatial prefetching implemented on top of a chunked file. Note that spatial prefetching over a chunked file has a similar performance advantage as a scenario where the chunked file is memory mapped and the chunk size is the same as the page size.

	mean time (s)	min time (s)	max time (s)
sub-volume in memory	30.7	25.7	44.4
block size=(8, 8, 64)	60.6	35.1	112.2
block size=(8, 64, 8)	63.2	53.9	142.4
chunked file and spatial prefetching	75.3	54.5	83.6
block size=(16, 16, 16)	78.3	50.3	135.5
block size=(64, 8, 8)	87.1	75.3	130.9
chunked file	505.4	169.2	817.1
file system cache only	540.4	158.5	860.9

Table 1. Linear Storage Versus Chunked Storage

We measured the time required to traverse the same $48MB$ sub-volume of data from a data volume stored using chunked storage. We used 16^3 as both the chunk size and block size. We obtained a small improvement of 4% for spatial prefetching over a chunked file than the performance for spatial prefetching over a linear file with block of size (16,16,16).

For spatial prefetching over a linear file with block size (8,8,64) we obtained an improvement of about 20%. when compared with the performance for spatial prefetching over a chunked file.

5. CONCLUSIONS AND FUTURE WORK

We presented a spatial prefetching module that helps speed up an iteration used in a volume visualization algorithm. Our tests showed an average speedup of 8.9 for a traversal that uses spatial prefetching with block shape (8,8,64) when compared with a traversal that uses only the file system cache. We showed that spatial prefetching over a chunked file is only about 4% better than spatial prefetching over a linear file and that we could even improve on performance obtained with chunking and spatial prefetching if we shape the blocks used in spatial prefetching such that the longer side is along the rod direction. We see spatial prefetching as an attractive alternative to chunking when the space required for replicating the file is not available.

In the future, we intend to extend our tests to cover several platforms and larger data sets and we plan to explore the benefits of spatial prefetching to other types of visualization applications. We are interested in testing the benefits of virtual memory (memory mapped files) on applications that work with out-of-core multidimensional data and compare that to spatial prefetching.

6. ACKNOWLEDGMENTS

This work was supported in part by Armstrong Atlantic State University under Research and Scholarship Grant 727186 and by the National Science Foundation under grants CCF-0541239, IIS-0082577 and IIS-9871859.

REFERENCES

1. J. L. Pfaltz, R. F. Haddleton, and J. C. French, "Scalable, parallel, scientific database," in *Proceedings 10th International Conference on Scientific and Statistical Database Management*, IEEE, (Los Alamos, CA), 1998.
2. S. Prohaska, A. Hutanu, R. Kahler, and H. Hege, "Interactive Exploration of Large Remote Micro-CT Scans," *Proceedings of the IEEE Visualization 2004 (VIS'04)* **0**, pp. 345–352, 2004.
3. computerhistory.org, "Computer history museum - timeline," 2006. This is an electronic document. Date retrieved: July 1, 2006. http://www.computerhistory.org/timeline/.

4. S. Sarawagi and M. Stonebraker, "Efficient organizations of large multidimensional arrays," in *Proc. of the Tenth International Conference on Data Engineering*, pp. 328–336, IEEE Computer Society, (Washington, DC, USA), 1994.
5. S. More and A. Choudhary, "Tertiary storage organization for large multidimensional datasets," in *8th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2000.
6. C. Chang, T. Kurc, A. Sussman, and J. Saltz, "Optimizing retrieval and processing of multi-dimensional scientific datasets," in *Proc. of the Third Merged IPPS/SPDP Symposiums*, IEEE Computer Society Press, May 2000.
7. C. Chang, T. Kurc, A. Sussman, and J. Saltz, "Active data repository software user manual."
8. A. 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.
9. P. Cao and E. Felten, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Transactions on Computer Systems* **14**(4), 1996.
10. A. Brown and T. Mowry, "Compiler-based i/o prefetching for out-of-core applications," *ACM Transactions on Computer Systems* **19**(2), 2001.
11. D. Kotz, "Disk-directed i/o for mimd multiprocessors," *ACM Trans. on Computer Systems* **15**(1), 1997.
12. T. Mowry, *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.
13. J.Gao, J.Huang, C. Johnson, and S.Atchley, "Distributed data management for large volume visualization," in *Proc. IEEE Visualization*, 2005.
14. P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr, "Iteration aware prefetching for large multidimensional scientific datasets," in *SSDBM'2005: Proc. of the 17th international conference on Scientific and statistical database management*, pp. 45–54, Lawrence Berkeley Laboratory, (Berkeley, CA, US), 2005.
15. M. Cox and D. Ellsworth, "Application-controlled demand paging for out-of-core visualization," in *VIS '97: Proceedings of the 8th conference on Visualization '97*, pp. 235–ff., IEEE Computer Society Press, (Los Alamitos, CA, USA), 1997.
16. K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf, "Real-time volume graphics," in *GRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, p. 29, ACM Press, (New York, NY, USA), 2004.
17. A. Kaufman and E. Shimony, "3d scan-conversion algorithms for voxel-based graphics," in *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics*, pp. 45–75, ACM Press, (New York, NY, USA), 1987.
18. F. Dunn and I. Parberry, *3D Math Primer for Graphics and Game Development*, ch. 10. Wordware Publishing Inc, 2002.