Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2002)
D. Ebert, P. Brunet, I. Navazo (Editors)

Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes

John Plate Michael Tirtasana Rhadamés Carmona[1] Bernd Fröhlich[2]

Virtual Environments, Fraunhofer IMK, St. Augustin, Germany
[1] Lab. Computacion Grafica, Facultad de Ciencias, Universidad Central de Venezuela
[2] Media Faculty, Bauhaus-Universität Weimar, Weimar, Germany

Abstract

We have developed a hierarchical paging scheme for handling very large volumetric data sets at interactive frame rates. Our system trades texture resolution for speed and uses effective prediction strategies. We have tested our approach for datasets with up to 16GB in size and show that it works well with less than 500MB of main memory cache for 64MB of 3D-texture memory. Our approach makes it feasible to deal with these volumes on desktop machines.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image generation – Display algorithms

Additional Key Words and Phrases: Volume rendering, texture caching, out-of-core rendering

1. Introduction

Increasingly sophisticated data acquisition techniques and complex simulations produces larger and larger volumetric datasets. The oil and gas industry in particular acquires enormous amounts of seismic data for the exploration of potential new reservoirs. This data has to be sighted by geologists and geo-physicists to discover the precious oil and gas containing subsurface structures. Dealing with these multi-gigabyte volumes has been really slow or even impossible in most applications.

We have developed a system that allows users to roam through multi-gigabyte volumetric data sets in real-time with low memory requirements. In our application domain, users typically explore their datasets using a set of slices through the volume and local volume rendering lenses as shown in Figure 1. Users browse through data sets by moving around slices and volume rendering lenses at different speeds. They quickly move through some areas and slow down or pause in other regions to take a closer look at local phenomena. Interactive frame rates are an important issue in this context and our system gives users the possibility to dynamically trade resolution for speed as necessary.

Our multi-resolution approach focuses on exploring only parts of a large volumetric dataset at a time and roaming through the volume. Other approaches, e.g. LaMar et al. [8] and Weiler et al. [11], use multi-resolution techniques to render large volumetric datasets as a whole with the best possible quality. Our work is based on similar multi-resolution data structures and also uses 3D-texture hardware. We augment these multi-resolution data structures with a sophisticated caching scheme and predictive paging techniques.

Our main contribution is the development of a hierarchical paging scheme that guarantees interactive frame rates for very large volumetric data sets by trading texture resolution for speed. Our approach handles paging from main memory into texture memory and paging from hard disk into main memory efficiently. We describe our paging techniques in detail and introduce effective prediction strategies. We have tested our approach for datasets with up to 16GB in size and show that it works well with less than 500MB of main memory, which makes it feasible to deal with these volumes on desktop machines.

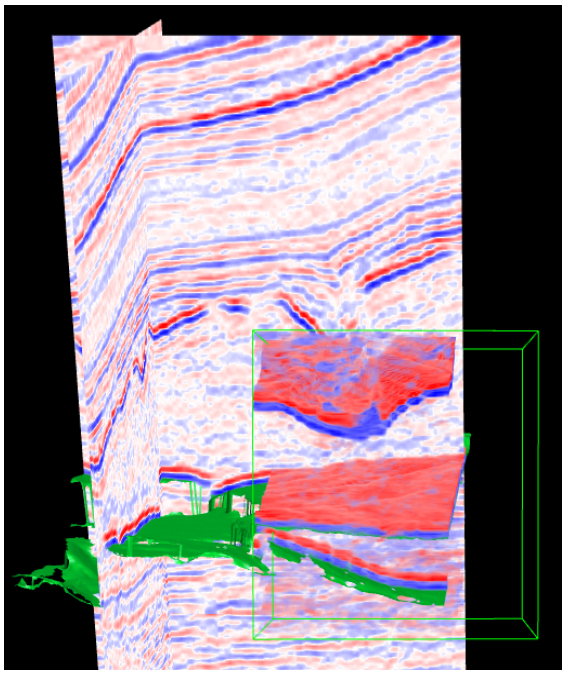


Figure 1: *A seismic dataset. Two orthogonal slices are shown and a volume rendering lens with a green frame. The blue and red stripes show strong reflections of the acoustic shock waves used to acquire this data. The green, polygonal surface is a horizon, which separates two earth layers.*

2. Related Work

3D-texture based volume rendering techniques and the appropriate sampling schemes were introduced by Cullip and Neumann[4]. Cabral et al.[1] were the first to show interactive volume rendering of medical datasets on graphics hardware with 3D-texturing capabilities. Westermann and Ertl [12] developed these ideas further to support efficient handling of clipping geometries and non-polygonal surface rendering.

LaMar et al.[8] describe an octree-based multi-resolution approach for interactive volume rendering. They filter the volume to create levels-of-detail in an octree structure. They propose the use of spherical shells to reduce visual artifacts for 3D-texture based volume rendering. Based on this work, LaMar et al. [9] introduce an adaptive scheme that renders the data along a cutting plane at different resolutions depending on the distance to a given center of interest. The required volume tiles are loaded for each frame. There is no explicit paging strategy introduced. Artifacts are limited by blending between different levels of resolution.

Weiler et al. [11] carefully address the avoidance of interpolation errors in their multi-resolution model for volumetric datasets. Their approach allows consistent interpolation between levels even for adaptive slice distances.

Cline and Egbert [2] also apply a two-level caching mechanism for dealing with large two-dimensional textures. They use a quadtree hierarchy to store their two-dimensional terrain textures at different levels of detail. The main differences to our approach are the following: they deal only with two-dimensional textures, their geometry is tiled according to the size of the texture tiles in a pre-process, and they do not use any prediction schemes.

Our approach focuses on roaming at interactive frame rates through three-dimensional volumes that do not fit into texture memory or even into main memory. Our hierarchical approach leads to paging and prediction strategies that make effective use of frame-to-frame coherence.

3. Volume Texture Paging

Our basic data structure for storing volumetric datasets is an octree similar to the one used by LaMar et al. [8]. Original volumes are divided into bricks of a certain size, typically in the range of 32x32x32 to 64x64x64 voxels each. These bricks create the finest level in our octree structure. Eight neighboring bricks are filtered into a single brick of the next coarser level until only a single brick remains on the top of the octree. We also deal with non-powers of two volumes, since our octree structure does not require the existence of all children for each node. For bricks, which are only partially filled with voxels, we maintain the boundaries of the original volume and clip sampling geometries against these boundaries. We are able to handle arbitrary polygonal meshes as sampling or proxy geometries for volume rendering as well as for cutting and slicing geometries.

Our approach is divided into two separate tasks, which are typically performed in parallel: paging from main memory into texture memory and paging from the hard disk into main memory.

3.1. Paging From Main Memory Into Texture Memory

We adopt a two-step approach for finding the desired bricks, which need to be loaded into texture memory for each frame. First we create a list of bricks, which would display the sampling geometry at the best possible resolution under the constraint of the texture memory size. In a second step, we optimize this list such that only a given number of bricks gets loaded into texture memory per frame. This step guarantees that the texture reload takes only a certain amount of time and therefore avoids slowing down the rendering process.

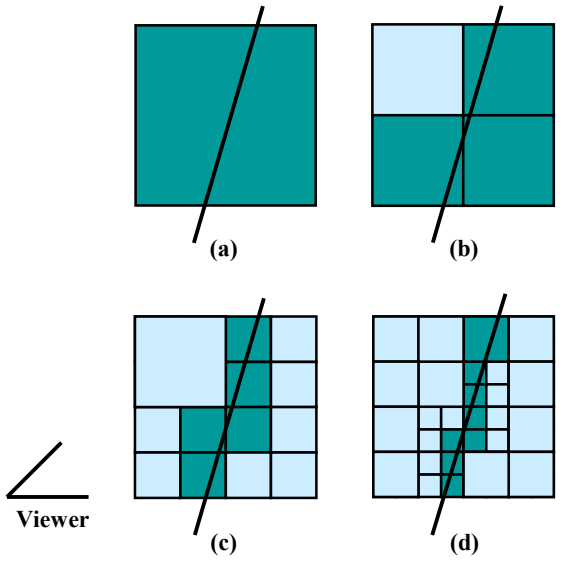


Figure 2: *(a) – (d) show the hierarchical insertion process of a slice into the octree structure. The dark tiles are required for displaying the slice, while the light colored tiles are ignored. The process stops, when the texture memory limit is reached or no further refinement is possible. In this example, we limit the amount of available texture memory to 9 bricks. The viewer is assumed to be on the lower left side of the dataset. Bricks closest to the viewer are refined first, which can be seen in (d), when the process stops.*

The first step starts with the top-down insertion of the sampling geometry into the octree. Figure 2 shows an example for a single slice. This process stops, when the finest level is reached or the number of required bricks equals the texture memory size (or a given fraction thereof). The refinement process works level by level and prefers bricks closer to the viewer over those farther away. This way we end up with only two different refinement levels. The sampling geometry is clipped on the fly at the boundaries of each brick. We call the list of bricks resulting from this refinement process our wish list, since it contains those bricks, we would like to have in our texture memory to display the sampling geometry at its best resolution. The brick wish list plus the path through the octree leading to the bricks on the wish list is called wish list tree.

The second step takes the wish list and compares it with the list of currently loaded texture bricks in texture memory. Those bricks, that are loaded into texture memory and do not belong to the wish list, are added to the list of unused bricks. This list sorts bricks by their last access time and does not remove bricks from texture memory. The remaining bricks form the reload list. If the number of bricks, which need to be reloaded, is larger than a given reload limit, we need to collapse refinement levels. Here we trade speed for resolution. We start the collapse process with the currently finest level and farthest away from the viewer. When the collapse process requests brick texture memory from the list of unused bricks, it is first checked if this brick is still in texture memory and unused, otherwise the oldest brick from the list of unused bricks is assigned. Figure 3 shows an example.

3.2. Adding Prediction

The described paging on demand strategy works quite well in most cases. By trading resolution for speed, this strategy will never stop the rendering process for longer than it takes to download the required bricks, but in some cases the displayed texture resolution can be quite low. Since sampling geometries typically do not jump around in texture space, a predictive paging strategy seems quite promising. Since texture memory and main memory to texture memory bandwidth are very scarce resources, we need to carefully plan how much memory and bandwidth we invest for predictive texture loading. One option is to just add all the neighbors of the current wish list to the wish list and performing the collapse operation based on this

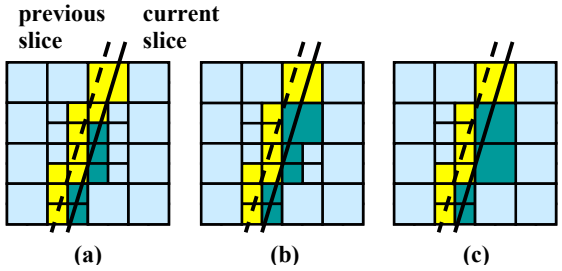


Figure 3: *(a) – (c) show the process of collapsing bricks to satisfy a given reload limit of 4 bricks. The dashed line shows the position of a cutting plane during the previous frame. The bricks that were resident in texture memory during the previous frame are shown in yellow. The dark bricks are those bricks that need to be reloaded for the current position of the slice. (a) 5 bricks need to be reloaded, which is more than our reload limit. (b) the brick that is farthest away from the viewer, which is assumed to be on the lower left corner of the dataset, is collapsed. Unfortunately, we still need to reload 5 bricks. (c) we then collapse the next brick and his neighbor and this time we reduce the number of new bricks to 4, which is our limit.*

extended list. Unfortunately, this requires such a large neighborhood that we often achieve only really coarse resolutions even when the sampling geometry is not moving at all. It soon became clear that we need a more precise estimate of the potentially needed bricks for the next frames. We decided to use linear extrapolation of the positions of the current sampling geometry based on their motion from the previous frame to the current frame. Copies of the current sampling geometry moved to the predicted locations are also inserted into the octree to create a separate wish list. The refinement and collapse processes work level by level as described in the previous chapter and alternate now between the two wish lists. For a given level in the octree, the refinement process takes the regular wish list first and refines it. Then the predicted wish list is refined. This process continues until the texture memory limit is reached. The collapse process works in the reverse order. It collapses the predictive wish list first and then the regular wish list until the reload limit is reached.

This linear prediction method avoids loading of large neighborhoods when the sampling geometry moves slowly or stops. In such cases pre-paging is effectively turned off and the maximum available detail for the actual sampling geometry is displayed. There are two trade-offs implied by this method. The displayed texture resolution is lower while moving a sampling geometry through the volume than without prediction and coarse texture bricks might be displayed, when the sampling geometry starts to move. Figures 10a and 10b show snapshots for texture memory paging with and without prediction.

3.3. Paging From Hard Disk Into Main Memory

We allocate a certain amount of main memory for texture caching from the hard disk. A separate process takes over the wish list generated by the actual sampling geometry and basically computes a hull around the bricks contained in the wish list tree. This hull contains all the neighbors, fathers, and neighbors of neighbors and so on of the current wish list. Basic priorities for loading these bricks are assigned based on distance to the bricks contained in the wish list tree and on the level in the hierarchy. The distance is measured in bricks. There are first order neighbors, 2nd order neighbors and so on. The elements of the wish list tree have neighborhood distance zero. An inner brick has 26 neighbors with a distance of one, 98 bricks with a distance of two, and so on. The scheme is shown in Figure 4. It gives priority to coarser levels, which should always be present before adding in finer detail.

Many bricks might have the same priority, because they have the same distance to a brick on the wish list. Here we prioritize those bricks, which were created by the most recent entries on the wish list. This means essentially that we are assigning higher priorities to bricks that we are moving towards than to those we are moving away from,

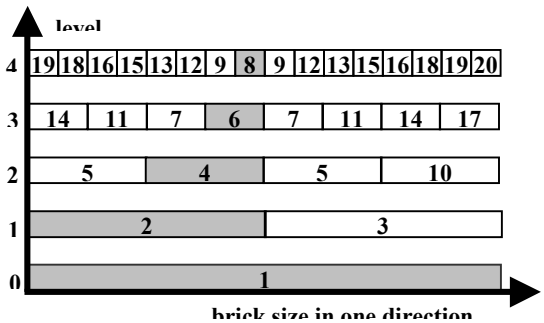


Figure 4: *Paging priorities given to bricks based on their level in the octree and their neighborhood distance to bricks in the wish list tree. The bricks with a gray background are elements of the wish list tree.*

even if they have the same neighborhood distance. This is a very important point, since it adds direction-dependent prediction to the paging process. Figure 10c shows an example.

When the available main memory cache is full, we need to decide which bricks should be discarded. We adopted a modified LRU strategy, which is based on the following discard priority p:

$$p = f(a,l,d)$$

a denotes the number of frames since the brick has been accessed, l is the octree level of the brick, and d is the neighborhood distance to the wish list tree. Currently we use a weighted sum of the parameters:

$$f(a,l,d) = k_a a + k_l l + k_d d$$

with the following parameter settings:

$$k_a = 1; k_l = 16; k_d = 1$$

This discard function and the empirically chosen parameters prefer to discard finer levels for bricks with the same age as well as bricks with higher neighborhood distance and the same age. The discard priority does not seem to have a really crucial effect on the performance of the system as long as there is a reasonable amount of main memory cache allocated, but nevertheless there might be better discard priority functions, which reduce the main memory requirements.

4. Implementation

The Octreemizer code has been developed in C++ and on top of OpenGL. There are only very few graphics calls in the core library, which could be easily ported to support

other graphics libraries. The original development took place on SGI Onyx2 machines with Infinite Reality graphics running SGI's IRIX operating system. In the meanwhile, Octreemizer has been ported to Windows and Linux platforms.

The latest version of our software has been tightly integrated with our virtual environment framework Avango (Tramberend [10]), which is based on SGI's Performer graphics tool kit. Performer uses multi-processing to pipeline application processing, view frustum culling, and rendering. Our software makes effective use of this process model to avoid stalling the graphics pipeline by performing too much computation during the rendering process. Our most time consuming task, the creation of brick wish list, happens in the application process, which runs in parallel with the culling and drawing process. This list is handed over to the drawing process, which optimizes the list to satisfy the reload limit, loads the required bricks (manual texture paging), and draws the geometry. Another separate process handles paging from the hard disk into main memory.

We have integrated the described algorithms into our geo-scientific prototype (Fröhlich et al.[5]). We use the system to drive stereoscopic display devices like workbenches [6,7], surround screen systems [3], and large multi-projector wall displays. Users interactively manipulate cutting planes and volume lenses as well as other data types and visualization primitives.

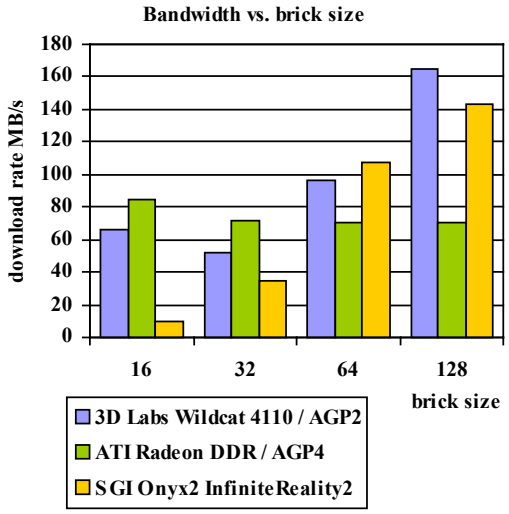


Figure 5: *The download rates for texture bricks with different sizes. Cubic bricks were used with the same number of voxels in each direction.*

5. Results

One of the most important parameters for texture paging is the texture download rate. Figure 5 shows the results of our benchmarks. We compared a SGI Onyx2 system with IR2 graphics with currently available PC graphics boards, which support 3D-textures. The results show a strong dependency of the download rate from the brick size. The Onyx2 in particular reaches only a very small percentage of the maximal download rate of 320 MB/s. The start up time and the general overhead for block transfers seems quite large and gets amortized only with larger blocks. The PC boards perform quite well for smaller block sizes and the dependency on the block size is less pronounced. These results suggest to use large block sizes, but the trade-off is that the actual number of bricks fitting into texture memory gets less and less. The texture memory range from approximately 45 MB on the ATI Radeon, which has a combined frame buffer and texture memory of 64MB, to actual 64MB on the Wildcat and InfiniteReality2 boards. For practical applications, we found that brick sizes of 32x32x32, 64x32x32, 64x64x32, and 64x64x64 voxels are a decent trade off between download rate and the partitioning of the texture memory.

Another important factor is the bandwidth from the hard disk into main memory. We performed some experiments to assess which structure would be best for storing the octree data structure on the hard disk. Our first option was to store the whole octree in a single random access file. The second option was to store each brick separately in a file. It turned out that we got much better performance for random brick access with the single octree file under the SGI IRIX 6.5 operating system. We stored the octree file on a 4-way stripe set and on a single hard disk. The stripe set delivered random bricks at about 18MB/s and the single disk at about 7 MB/s. Disk caching was turned off. The main memory to texture memory bandwidth is much higher than the hard disk to main memory bandwidth, but in typical applications we do not want to spend the entire frame time with downloading textures. This means that the effective bandwidth for both transfers is approximately the same, which keeps the main memory cache size within a reasonable range.

We have done a set of performance tests with data sets of 120MB (64x64x32 bricks), 2.1GB (64x64x64 bricks), and 16GB(64x64x64 bricks). We moved a volume lens on a circular track through the volume. There were 64 sampling planes inside the volume lens for the first two tests (figure 6 and 7). The size of the volume lens varied between 0.15x0.15x0.15 and 0.3x0.3x0.3. The size of the volume was 1.0x1.0x1.0 in each case. The texture memory reload limit was set to 0.5 MB/frame. The window size was 600x600. We recorded the frame time for the static lens and for the moving lens on an SGI Onyx2 system with IR2 graphics.

Data set size	120 MB	2.1GB	16GB
Texture memory size	10 MB	64MB	64MB
Main memory cache	120 MB	400MB	400MB
Time per frame static	3.7ms	5.3ms	10ms
Time per frame moving	35ms	40ms	41ms

Figure 6: *Performance measurements for a volume rendering lens of size 0.15x0.15x0.15.*

Data set size	120 MB	2.1GB	16GB
Texture memory size	10 MB	64MB	64MB
Main memory cache	120 MB	400MB	400MB
Time per frame static	7ms	15ms	14ms
Time per frame moving	48ms	45ms	46ms

Figure 7: *Performance measurements for a volume rendering lens 0.3x0.3x0.3.*

Data set size	120 MB	2.1GB	16GB
Texture memory size	10 MB	64MB	64MB
Main memory cache	120 MB	400MB	400MB
Number of slices	128	47	56
Time per frame static	13ms	13ms	13ms
Time per frame moving	56ms	43ms	45ms

Figure 8: *Performance measurements for a volume rendering lens 0.3x0.3x0.3 and a variable slice number to keep the time per frame constant for the static case.*

The texture memory size was artificially reduced for the smallest volume with 120MB to force texture paging. Hard disk paging was only necessary at the beginning, because the whole volume fit into the main memory cache. For the 2.1GB and 16GB volumes the measurements were pretty similar as expected. The time per frame for the moving volume rendering lens is dominated by the texture downloads and the overhead associated with collapsing the wish list. We found that 400MB of main memory seem to be enough for manual movement of the lens through the volume. Our system did not produce any cache misses when running off a local hard disk. This is mainly due to the limited bandwidth into the graphics card, which needs to be shared between downloading triangles and volume bricks.

We have just received a new computer which allowed us to provide some more interesting and detailed results. The hardware is a dual processor Xeon4 1.7GHz computer equipped with 2GB RAM, and an ATI FireGL4 graphics board with 128MB shared video and texture memory. As a test case we used our 2.1GB seismic data with a resolution of 1250x1300x1400 voxels (8 bit). The total size of the octree is about 11000 bricks of size 64^3, which results in a data set size of 2.7 GB. We used a window size of 600x600 with the view of the whole volume just fitting into the window.

The volume lens had a size of 0.2x0.2x0.2 or 0.8% of the total volume. We used 256 slices inside the volume lens. The main memory cache was set to 1500MB. We reserved 90MB of the 128MB memory on the graphics board for our volume bricks. The reload limit from main memory into texture memory was set 2 MB / frame, since this graphics card provides more than 300MB/s download from main memory into texture memory. We achieved an average frame rate of about 10 frames per second with this configuration when moving the lens through the data set. The frame time of 100ms consists of about 20ms for sorting the 256 slices into the octree structure, 8ms for reloading the bricks, and about 75ms for drawing the slices. All the other operations necessary require less than 1ms in total. An interesting observation was the following: We could reduce the download time to 4ms, when we were alternating between drawing slices and reloading bricks instead of loading all the new bricks before drawing the slices. This clearly shows that this graphics card is able to interleave these operations and perform them in parallel.

For providing quantitative information about the influence of our prediction strategy, we used the following two test cases with and without prediction: the volume lens was moved at two speeds through the volume on a circular track taking 10 seconds respectively 40 seconds for one round. Figure 9 shows a table with the minimum, maximum, average, and standard deviation of the octree depth of all the bricks contained in texture memory. It can be clearly seen that the average brick depth is higher with prediction, but more importantly the standard deviation is much smaller resulting in a much more even appearance.

	min	max	average	dev
slow / no P	3.6	5.0	4.85	0.18
slow / P	4.6	5.0	4.99	0.017
fast / no P	3.2	4.5	4.18	0.15
fast / P	4.1	4.4	4.22	0.05

Figure 9: *The minimum, maximum, average, and standard deviation of the octree depth of all the bricks contained in texture memory for four test cases: the slow and fast moving volume lens with and without prediction (abbreviated as P). The maximum octree depth was 5.*

Another convenient side effect of the system is that start up times have been reduced to a minimum. Without the main memory caching, the whole volume was loaded at start up time, which took about two minutes for the 2GB volume. Now the application starts up immediately and loads the volume on demand.

6. Conclusions and Future Work

We have presented efficient paging and caching techniques for dealing with multi-gigabyte volumes on small and medium scale computers. Fill and download rates of current and next generation PC graphics cards are very encouraging and reach far beyond those available on the very high end graphics engines today. In addition, full support of 3D-texturing on these cards is just becoming available. This makes porting to Linux systems and IA32 platforms worthwhile. Current and next generation high end hardware provide larger texture memory in the range of 256MB to 1GB, which allows more detailed presentations.

Our current system does not provide an explicit frame rate control mechanism. Users need to specify a fixed limit for the texture download in megabytes per frame. By dynamically updating this limit depending on the current frame rate, we could provide a frame rate control mechanism, which keeps the frame rate stable. This is an important feature for interactive virtual environment applications.

Currently, we have only done tests, where the volume data resides on a local hard disk. We need to investigate how our prediction techniques and caching strategies work for larger data sets over network connections, since data bases are often maintained on large file servers.

For real world applications, it is often necessary to support multi-attribute volumes and several independent volumes at once. These volumes need to share the available memory and bandwidth. We are extending our caching and paging strategies to be able to handle these cases efficiently. Visualization techniques for multi-attribute data and several volumes also need further development.

We are currently working on an interactive volume editing system based on the described multi-resolution structure. In this system, we support typical image processing operations such as delete, enhance, filter, and shade. These are very important operations to facilitate the exploration and understanding of complex volumetric datasets.

Acknowledgments

This work was partially supported by the VRGeo consortium. We thank the members of the consortium for their valuable feedback during our meetings.

References

1. Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. 1994 Symposium on Volume Visualization, 91-98, October 1994.

2. David Cline and Parris K. Egbert. Interactive display of very large textures. Proceedings of IEEE Visualization '98, 343-350, October 1998.

3. Cruz-Neira, C., Sandin, D.J., and DeFanti, T.A. Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE. Proceedings of SIGGRAPH '93, 135-142, 1993.

4. T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture mapping hardware. Technical Report TR93-027, Department of Computer Science,University of North Carolina, Chapel Hill, 1989.

5. B. Fröhlich, S. Barrass, B. Zehner, J. Plate, M. Göbel: Exploring Geo-Scientific Data in Virtual Environments, Proceedings IEEE Visualization 1999, 169-173, October 1999

6. Krüger, W., Bohn, C.-A., Fröhlich, B., Schüth, H., Strauss, W., and Wesche, G. The Responsive Workbench. IEEE Computer, 42-48, July 1995

7. Krüger, W., and Fröhlich B. The Responsive Workbench. IEEE Computer Graphics and Applications, 12-15, May 1994

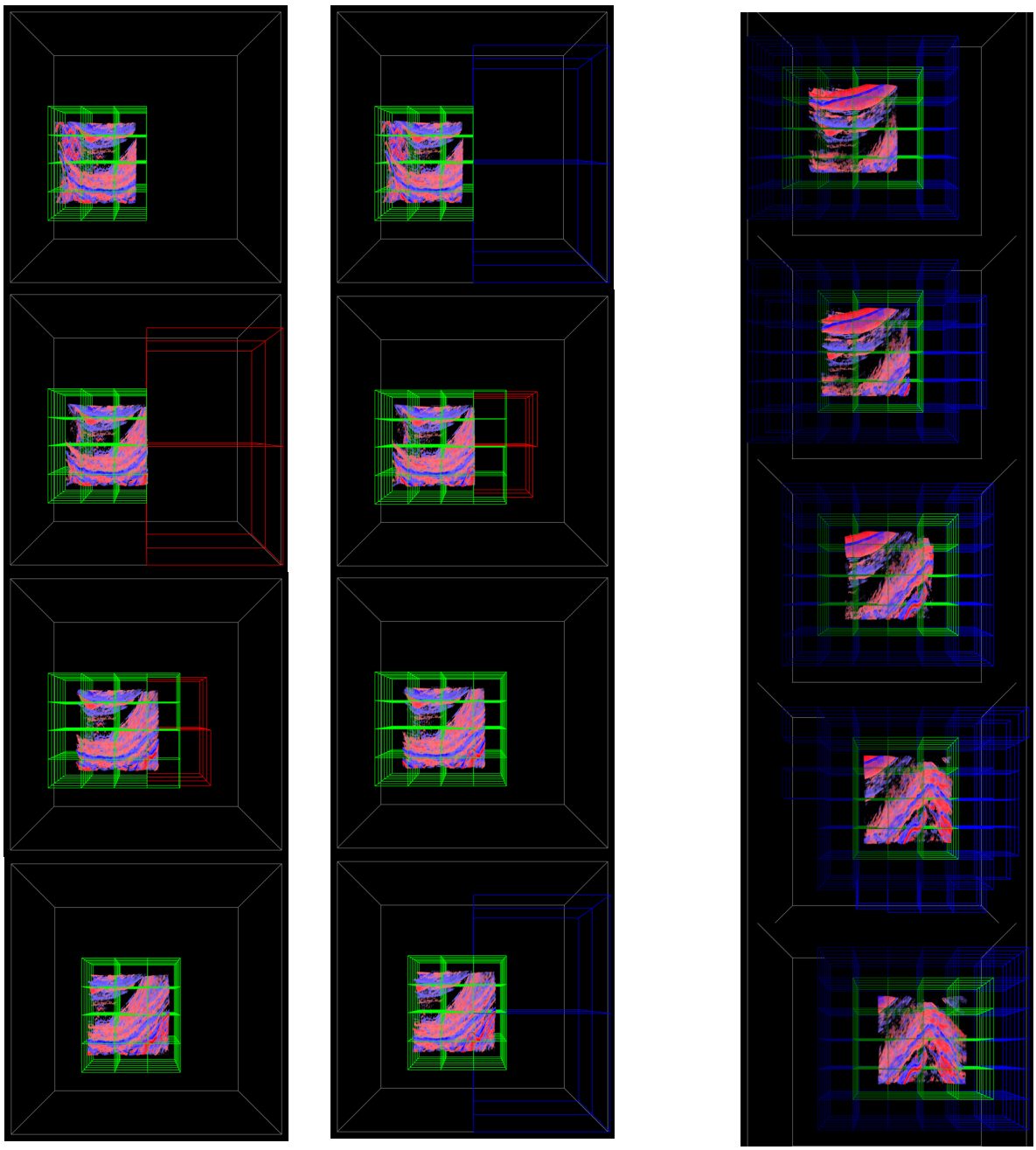
8. Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution Techniques for Interactive Hardware Texturing-based Volume Visualization. In IEEE Visualization 99, 355-361, November 1999.

9. Eric LaMar, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Multiresolution Techniques for Interactive Texture-based Rendering of Arbitrarily Oriented Cutting Planes, VisSym2000, 105-114, 2000.

10. Tramberend, H. Avocado: A Distributed Virtual Reality Framework. Proceedings of VR'99 Conference, Houston, Texas, 14-21, March 1999.

11. Manfred Weiler, Ruediger Westermann, Chuck Hansen, Kurt Zimmerman, Thomas Ertl. Level-Of-Detail Volume Rendering via 3D Textures, IEEE Symposium on Volume Visualization '00, 7-13, 2000

12. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In Computer Graphics (SIGGRAPH 98 Proceedings), 291–294, 1998



(a) **(b)** **(c)**

Figure 10: *This figure shows snapshots from moving a volume rendering lens from left to right through a larger volume. Texture paging from main memory into texture memory is shown in (a) and (b). No prediction is applied for (a). Prediction is applied for (b). The green and red bricks are requested by the current wish list. Green bricks show the finest level. Red bricks show a coarser level. The blue bricks are requested by the predictive wish list. It is easy to see that prediction provides more detail and the pre-paging happens only in the direction of the movement. (c) shows the texture paging from the hard disk into main memory. Green bricks contain actual geometry. Green and blue bricks are in main memory. Blue bricks show the hull generated around the wish list. The brightness of the lines shows the age of the bricks. Bright colors show younger bricks, darker colors show older bricks.*