Giga-Scale Multiresolution Volume Rendering on Distributed Display Clusters

Sebastian Thelen[1], Joerg Meyer[2], Achim Ebert[1], and Hans Hagen[1]

[1] Technical University Kaiserslautern, Germany
{s_thelen, ebert, hagen}@cs.uni-kl.de
[2] University of California, Irvine, U.S.A.
jmeyer@uci.edu

Abstract. Visualizing the enormous level of detail comprised in many of today's data sets is a challenging task and demands special processing techniques as well as a presentation on appropriate display devices. Desktop computers and laptops are often not suited for this task because data sets are simply too large and the limited screen size of these devices prevents users from perceiving the entire data set and severely restricts collaboration. Large high-resolution displays that combine the images of multiple smaller devices to form one large display area have proven to be an adequate solution to the ever-growing quantity of available data. The displays offer enough screen real estate to visualize such data sets entirely and facilitate collaboration, since multiple users are able to perceive the information at the same time. For an interactive visualization, the CPUs on the cluster driving the GPUs can be used to split up the computation of a scene into different areas, where each area is computed by a different rendering node.
In this paper we focus on volumetric data sets and introduce a dynamic subdivision scheme incorporating multi-resolution wavelet representation to visualize data sets with several gigabytes of voxel data interactively on distributed rendering clusters. The approach makes efficient use of the resources available on modern graphics cards which mainly limit the amount of data that can be visualized. The implementation was successfully tested on a tiled display comprised of 25 compute nodes driving 50 LCD panels.

Keywords: High-Resolution Displays, Large-Scale Data Sets, Volume Rendering

1 Introduction

Many data sets that are acquired in today's applications exceed the computational capabilities of desktop PCs or workstations. The Visible Human Project [34], for instance, provides 40GB of data for the female cadaver, thus asking for techniques to process an enormous amount of information. A lot of research has focused on developing methods for handling such large data sets. Solutions include the parallelization of the render process or the invention of level-of-detail or data

compression techniques. The benefits of these methods are undeniable, however, one limitation remained for a long time: the visualization of large amounts of information and the advantage of having high-resolution data sets also require a large amount of screen real estate.

The pixel count of regular computer monitors is usually too low to display complex data sets at their full level of resolution. For three-dimensional data sets, the gap between the amount of information contained in a partially transparent volume and the available screen space becomes even more obvious.

Zooming+panning (e.g., Google Earth navigation) or focus+context (i.e., a high-resolution focus view and a low-resolution context view) approaches offer ways to deal with this problem but for some data sets these methods are not appropriate. For instance, such data sets can only be fully perceived when they are displayed entirely. In the last couple of years tiled high-resolution displays have become more and more popular due to advances in display and hardware technology. Tiled high-resolution displays combine the resolution of multiple devices to form one large display area. Two basic approaches have emerged: *projector-based* and *monitor-based* tiled displays. Multiple projectors can be combined to form a projector-based tiled display. The challenge is to calibrate the system, as projector images are usually distorted and non-uniform in terms of color and luminance. LCDs represent the most affordable way to build a large high-resolution display. LCD-based systems are easier to set up since they usually require less space, and problems like lens distortions or mismatches of color temperature and luminance are limited to a minimum. Further, the resolution of today's average LCDs is higher than the one of most projectors, so that it is relatively easy to build systems with a resolution of several hundred megapixels.

In this paper we focus on the development of a volume rendering application for large scalar data sets on tiled displays. The display system we work with is a tiled wall consisting of fifty 30 inch LCDs arranged in a 10×5 grid. Instead of the traditional isosurfaces, i.e., regions representing a constant scalar value within the data set, we use a direct volume rendering approach based on 3D texture mapping for visualization. The size of data sets that can be visualized on a computer is mainly limited by the available amount of video RAM on the graphics card. Therefore, the main contribution of this paper is the description of a technique that fully exploits the resources of a render node in the display cluster. We combine octree-based out-of-core frustum clipping with a wavelet-based multi-resolution representation of the data to increase the visual quality of the renderings.

The rest of the paper is structured as follows: Section 2 gives a brief overview of current literature relevant to our work. Section 3 describes the data structure we employ, while section 4 deals with software- and hardware related aspects. In section 5, we introduce the actual method that allows us to visualize high-resolution data sets on large displays, followed by a description of implementational aspects in section 6. Section 7 discusses the results we achieve on our system that are more deeply analyzed in section 8. Section 9 compares our system to other dis-

tributed rendering systems. The paper ends with a conclusion and an outlook on future work.

2 Related Work

In its simplest form, 3D texture mapping is limited by the fact that the original data has to fit entirely into the texture memory of the graphics card. To overcome this limitation, LaMar et al. [20] describe a multi-resolution approach that is based on octree subdivision. The leaves of the octree define the original data and the internal nodes define lower-resolution versions. Artifacts are minimized by blending between different resolution levels. Weiler et al. [48] extend this approach. Their hierarchical level-of-detail representation allows consistent interpolation between resolution levels. The described methods are relevant to our work since we also produce multi-resolution images of the data set. However, in our case multi-resolution refers to the detail levels of tiles within the display wall and not to regions of different resolution within the resulting picture. The techniques above are able to handle data sets that do not fit into the texture memory of the graphics hardware. Nonetheless, data still has to fit into the computer's main memory.

The data structure we exploit in this paper has been described by Meyer et al. [31]. The authors use a dynamic subdivision scheme that incorporates space-subdivision based on octrees and wavelet compression. It has been used to implement a network-based rendering application that is able to visualize data sets between 20MB and 76GB size by first transferring low-resolution versions of the data via network and then gradually refine them [30]. Plate et al. [38] developed the Octreemizer, a hierarchical paging scheme that uses octrees to deal with data sets that exceed the size of RAM. The Octreemizer however only exploits the octree subdivision characteristics without incorporating any data compression scheme.

Various attempts have been made to parallelize rendering on a cluster of computers by image-order [37, 2, 3], object-order [13, 16] or hybrid approaches [14]. The aim of these methods is to achieve a speed-up in the rendering process by combining the computational power of a set of computers. Far less research though has focused on utilizing clusters to generate high-resolution results of volume rendered images on high-resolution displays.

Vol-a-Tile [46] was able to visualize seismic data sets on the 5×3 GeoWall2 [15] display grid at EVL. The authors present a master-slave prototype that uses an MPI-based rendering library and an OptiStore data streaming server. Application specific implementation details are not given. The TeraVoxel project [23] founded at the California Institute for Technology was able to interactively render a $256 \times 256 \times 1024$ data set on a 3840×2400 display using four VolumePro 1000 cards by employing pipelined associative blending operators in a sort-last configuration, contrary to our sort-first approach. Schwarz et al. [45] describe an octree approach for frustum clipping to make better use of system resources without exploiting the power of a multi-resolution wavelet representation and

give a comparison of various parallel image- and object-order as well as hybrid volume rendering techniques. QSplat [43] is a multi-resolution rendering system for displaying isosurfaces out of point data sets. It also uses a progressive refinement technique but generates images using a splatting approach (see section 5.1). Despite the increasing popularity of high-resolution displays, until today the ultimate general purpose software solution for driving all different types and configurations of displays is still missing. Various commercial and non-commercial software solutions are available, differing mainly in the way they perform distributed rendering and the display configurations they can handle. Distributed rendering strategies were first analyzed and classified by Molnar et al. [33]. The authors identified three classes of rendering algorithms, characterized by the stage in which geometry is distributed among the nodes (sort-first, sort-middle and sort-last). Alternatively, distributed rendering can be classified by the execution mode of applications on the cluster [9]. In master-slave mode, an instance of the application is executed on each node of the cluster, while in client-server mode a client node issues rendering tasks to the render servers via network. A good survey on distributed rendering software was given by Ni et al. [36] and Raffin et al. [39]. Some of the best known software packages include CAVELib [8], VRjuggler [5], Syzygy [44], AnyScreen [10], OpenSG [40], Chromium [17], NAVER [18], Jinx [47] , and CGLX [11].

3 Data Structures

Rendering of large data sets in real time requires data reduction techniques and hierarchical subdivision of the data set. This chapter describes a space- and time-efficient, combined space subdivision and multi-resolution technique for volumetric data [30].

3.1 Adaptive Octree

Large volumetric data sets, which are typically arranged in a regular Cartesian grid, are often too complex to be rendered in real time on today's hardware for cluster nodes and with current 3D texture-based rendering techniques. Therefore, they must be reduced to a reasonable size, so that each node can quickly access the appropriate section of the 3D volume and render this portion of the data almost instantaneously. The slice format in which volume data often comes is not suitable for this purpose, because too many files need to be opened in order to display a 3D volume from cross-sections. Opening large numbers of files has proven to be prohibitively slow.
Therefore, an octree space subdivision algorithm is employed. Instead of using a database, the Unix file system and an efficient indexing scheme are used to access the right block of data instantly. The octree method is illustrated in Fig. 1. Three simple binary decisions (left or right, front or back, and top or bottom) are necessary at each level to determine the position of a voxel in an octree. Each decision corresponds to a two bit index, which can take on $2^3 = 8$ states (0-7),

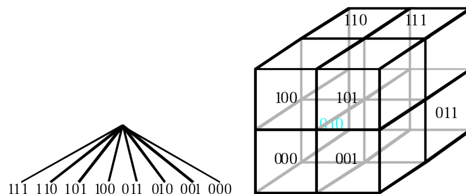


Fig. 1. Octant indexing scheme used to represent paths in the octree.

i.e., the eight child nodes of the octree. The number represents the file name, and several numbers in a file name indicate an octree traversal path. With this fast indexing scheme, it is possible to traverse the tree almost instantaneously in order to get to the leave of the tree which carries the data.

The main advantage of the octree data structure is the early termination of the traversal in branches that are empty. It works best for data sets where the object is located near the center of the data cube and has an empty or nearly empty background (e.g., for CT or MRI scans), but it also works for sparse data sets. The information is determined in a preprocessing step when creating the octree data structure. Empty areas are not stored, saving a significant amount of disk space and RAM.

Several space subdivision algorithms have been implemented in order to compare their performance and scalability [32] with the octree approach. Binary Space Partition (BSP) trees, for example, are based on a binary subdivision of three-dimensional space. A root node is recursively subdivided in alternating directions, in a cycle of several subdivisions (one for each dimension). Empty regions are marked, so that they can be skipped in order to speed up the rendering procedure.

A statistical analysis based on typical data sets (CT and MRI scans) has been conducted, and it was found that for the same data set size the octree technique consistently outperformed other techniques like the BSP tree both in terms of preprocessing time and, more importantly, traversal time during rendering [31]. At this point, depending on the depth of the octree, the data block to be read from the octree may still be too large to fit into the texture buffer of a rendering node's graphics card. For this reason, the leaves are stored in a multi-resolution wavelet format, which is described in the next section.

3.2 3D Non-standard Haar Wavelet Decomposition

We use a 3D non-standard Haar wavelet decomposition method to store the leaves. The benefit of using this method is the fact that low-resolution information can be stored at the beginning of the file, and if more detail is desired, more data in the form of detail coefficients can be loaded in order to refine the

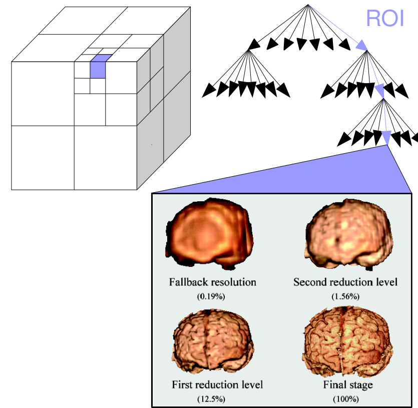


Fig. 2. Octree space subdivision uses multi-resolution wavelet representation for the leaves.

image. Depending on the number of wavelet levels, a tremendous reduction of data can be accomplished with acceptable visual artifacts in the rendered image. On the first level, only 1/8th of the data needs to be rendered, on the second level 1/64th (less than 2% of the total data set), and so forth. This exponential reduction leads to a significant acceleration and memory savings.
The volume pyramid can be compressed by removing small coefficients. A threshold determines whether a coefficient can be set to zero or not. Real numbers with fractions can be quantized, e.g., by removing the fraction part, and low-frequency coefficients could be stored at higher precision than high-frequency coefficients. However, one of the most important advantages of the wavelet approach is the fact that we can use the same amount of memory to store the hierarchy as to store the original volume. Typically, we would use float as the data type to store detail coefficients. We can replace this data type by integers with no loss of precision, and therefore we can make use of the fact that integer arithmetic is much faster than floating point arithmetic [7].
In our implementation, we use the following method: Volume data is represented as a three-dimensional array of scalar values. The resolution is 8 bits, which allows for 256 different gray levels. For the wavelet transformation, we use an unsigned integer (32 bits) to store coefficients, which is the same size as a float (4 bytes). Initially, our data is stored in bits 23...30. The leftmost bit 31 is usually unused. Only when the algorithm computes the sum of two data items in order to determine the reduced pixel, i.e., their representation in the lower resolution image, the intermediate result can become temporarily larger than 255, which causes an overflow into this unused bit. For the computation of reduced pixel values and detail coefficients, a division by two is involved, which corresponds to a bit shift to the right, i.e., an underflow. Since we have three dimensions per cycle, we can have a 3 bit underflow. We have 22 bits available. This means that we can apply the algorithm 7 times before we run out of bits. This is sufficient,

since the data set is then already reduced to $(1/8)^7 = 4.77 \times 10^{-7}$. This means, for instance, that a typical data set of $256^3 = 16,777,216$ voxels is then already reduced to 8 voxels.

4 System Details

The following section discusses hard- and software related aspects of the display cluster used for the visualizations.

4.1 Cluster Display Wall

Our visualization cluster (see Fig. 3) consists of 25 PowerMac G5 computers, equipped with nVidia GeForce 6800 and nVidia Quadro FX4500 graphics cards. These 25 compute nodes drive fifty 30 inch Apple Cinema Displays that have been arranged in a 10×5 grid. Together they form a 200 megapixel tiled display wall which measures 23×9 ft. Each screen has a native resolution of 2560×1600 pixels. The wall can display scenes at a maximum resolution of $25,600 \times 8,000$ pixels. A designated front-end node processes user input and launches applications. The current operating system on the cluster is Mac OS X Tiger.

4.2 CGLX

As mentioned in section 2, there exists a variety of different distributed rendering libraries. Our implementation of the volume renderer is based on CGLX [11], a Cross-Platform Cluster Graphic Library. CGLX was developed at the California Institute for Telecommunications and Information Technology at UC San Diego and is a flexible and transparent OpenGL-based graphics framework for distributed visualization systems in a master-slave setup. Master-slave architectures run an instance of the application on each compute node of the cluster. Communication between nodes is realized through a light-weight, thread-based

Fig. 3. 200 megapixel tiled display cluster measuring 23×9 ft.

network communication protocol. Attempting to be as transparent as possible to the developer, CGLX offers a GLUT-like (OpenGL Utility Toolkit) interface, which allows the library to intercept and interpret OpenGL calls, providing a distributed large scale OpenGL context on tiled displays.
Theoretically, existing applications can be ported to CGLX by just changing the includes from `GL/glut.h` to `clgX.h` and calling `cglXOrtho`, CGLX's equivalent to `glOrtho`. However, in practice things like synchronized access to global variables or shared resources (e.g., files) require some more tweaking.

5 System Design

When doing volume rendering of large-scale data sets on desktop computers, the limited resources of the system pose restrictions to the visualizations. Above all, the available amount of video RAM determines the size of data sets that can be visualized on a given system. The same restrictions apply when designing cluster applications, where each node computes the entire scene. The waste of system resources is enormous, since nodes allocate memory for the entire scene, but only display a small part of it. Often however, this approach is favored by distributed rendering libraries. Since libraries try to be transparent, programmers write code almost as if they are developing applications for desktop PCs. The library lets each node compute the entire scene and sets viewport and viewing frustum according to the location of its screens within the grid. While this is a clear advantage for the developer, it leads to a waste of system resources, which we try to avoid with algorithm described in the following section.

5.1 Rendering Algorithm

Different methods have been developed to visualize volumetric data sets by approximating the volume rendering equation [29]. As opposed to isosurface algorithms [24], these techniques simulate the propagation of light in a transparent light emitting and absorbing medium.

Raycasting [21, 19, 42] is an image-order method that casts rays through each pixel of the viewing plane into the volume and samples along the ray. The method is able to yield high-quality results but is computationally expensive. In order to be interactive, it requires sophisticated adaptive sampling schemes, empty space skipping, and early ray termination. Their implementation is partly supported by today's modern programmable GPUs. However, even though raycasting is able to produce the best optical results, it generally means having to trade speed for image quality. We decided that raycasting is not going to pay off when the goal is to achieve interactive framerates on a fifty tile display wall.

Splatting [50, 49, 43] is a rendering approach in which voxels of the volume are projected in back-to-front order onto the 2D viewing plane. Splat kernels, e.g., Gaussian kernels, are used to blend splats and generate smooth

images. Splatting in general is faster than raycasting, but the implementation is tricky and results depend on the choice of the splat kernel. Therefore, splatting was not our first choice rendering algorithm.

Texture mapping [51, 6, 41] treats volumetric data as 3D textures that are mapped to sets of proxy geometry. The general approach is to map data sets to a stack of view-aligned planes via trilinear interpolation. Afterwards, blending of adjacent planes generates a continuous three-dimensional visualization of the volume. Texture mapping is very efficient on modern GPUs, since trilinear interpolation is hard-wired on today's graphics cards. Furthermore, the technique is easy to implement and produces good-quality results. Therefore, we decided to implement a rendering algorithm that is based on this approach.

It is worth pointing out that our prototype could theoretically be adapted to use any of the other rendering approaches. Most changes would concentrate on the main render loop and not on the data structure employed in our algorithm. We want to make full use of the resources of each rendering node and determine which parts of a scene its monitors are going to display. The rendering node allocates resources only for a portion of the data set and uses the available video memory to display its subvolume at the highest possible quality. For doing so, we make use of the data structure described in section 3. The volume subdivision of the octree is well suited for being used in a frustum clipping step (see section 6.1). It allows rendering nodes to disregard parts of a scene that do not have to be displayed. These parts do not need to reside in RAM and do not need to occupy precious texture memory. The wavelet representation of the octree data allows rendering nodes to determine a compression level for the subvolumes, so that they still fit in their graphics card's texture memory and at the same time provide the highest possible level of detail.
As a result we end up with a scene in which each LCD of the grid displays a portion of the volume at an individual but maximal quality. For large-scale data sets this quality can differ by a factor of $8 - 64$ (which corresponds to $1 - 2$ wavelet levels) from the results that can be achieved on a desktop PC.

5.2 Progressive Refinement

When transforming the volume (by translation, rotation or scaling operations) frustum clipping has to be rerun because parts of the data set might fall into a different region of the display wall. Frustum clipping can slow the system down because it implicates time consuming file operations to load new data chunks from hard disk into main memory. Therefore, it is impossible to clip at interactive rates. Our solution to this bottleneck is a fallback texture. The fallback texture permanently resides in the graphics card's texture memory and contains the entire data set which is loaded only once at start-up time. Whenever the volume is transformed, i.e., when the user uses the mouse to change the current position or scaling factor of the volume, we switch from the individually

computed subvolumes of a node to the fallback texture. The size of the fallback texture is chosen in a way that each rendering node maintains a predefined minimum frame rate. That way we keep the system responsive and guarantee that the volume is displayed at any given time in its current position. Because the fallback texture contains the entire data set and has to be stored once on each node, its detail level is usually lower than the ones that result from the clipping process.
When the mouse is released, i.e., when user interaction is paused, the system gradually increments a counter, which determines the detail level. As a consequence, the image is gradually refined up to the maximum level of detail each graphics card can display.
Data sets can comprise several gigabytes and are stored in the octree-wavelet format. The size of the octree bricks is determined by the octree depth - a parameter that is specified by the user. The deeper an octree is, the more brick files there are and the smaller the brick files will be. However, loading the required information from these files in just one step may take too long. The data organization of the brick files allows a rendering node to load the content incrementally. The content of the files is organized in such a way that the data of the highest wavelet compression appears at the beginning of a file and is followed by the detail coefficients of lower wavelet levels (see section 3). Thus, it is possible to first load the lowest detail level and then gradually refine by sequentially loading detail coefficients for the reconstruction. In practice this means, that we start at a level i, display the data set at that resolution and then, triggered by a timer, switch to level $i-1$, $i-2$ and so forth. This way, after having interacted with the volume, the user is provided with instantaneous visual feedback.
Progressive refinement including fallback textures keeps the system responsive and enables users to explore data sets interactively. For small movements through the volume, where only parts of a scene change, the user experience could in the future be improved by loading additional high-resolution data around a hull of the frustum.
Synchronization between nodes is achieved through the render library's built-in synchronization mechanism. Before OpenGL render buffers are swapped, a barrier synchronization lets the system stall until all threads reach the barrier. While this means having to wait for the slowest node in the cluster, it gives us a way to switch synchronously between wavelet levels during progressive refinement and to reduce artifacts. Note, that within a tile the wavelet level is always constant, in contrast to the method that was presented by LaMar et al. [20]. Their algorithm generates regions of varying resolution within the final image.

6 Implementation

This section discusses implementation details of the frustum clipping and texture mapping stage.

6.1 Out-of-Core Frustum Clipping

During the clipping stage each rendering node determines for each of its monitors (i.e., two in our case) which parts of the data set they are going to display depending on the current view. This yields a subvolume for each screen for which the according files are loaded at the node. The subvolume is loaded at the maximum possible resolution, so that it still fits into a predefined portion of the texture memory. As a result, frustum clipping yields a partial copy of the entire data structure from hard disk in a node's main memory, ready to be visualized by 3D texture mapping.

To determine if an octree brick is going to be displayed by a monitor, we calculate where on the tiled display the eight corners of an octree brick are going to appear, i.e., we project them into 2D space and calculate their window coordinates. Determining the window coordinates (wx, wy) of a given point $v = (ox, oy, oz, 1.0)$ in object space under a modelview matrix M, a projection matrix P and viewport V can be accomplished via OpenGL's `gluProject` method. `gluProject` first computes $v' = P \cdot M \cdot v$. The actual window coordinates are obtained through

$$wx = V[0] + V[2] \cdot (v'[0] + 1)/2 \tag{1}$$

$$wy = V[1] + V[3] \cdot (v'[1] + 1)/2. \tag{2}$$

wx and wy specify the window coordinates of point v. The viewport V of course is defined by the size of the tiled display, i.e., $25,600 \times 8,000$ in our case. The projection matrix P is the matrix that orthogonally projects the viewing frustum to the viewport V. Since we transform the volume by manipulating OpenGL's texture matrix T, we use its inverse T^{-1} as an input to `gluProject` instead of the modelview matrix M (inverse, because T manipulates texture coordinates and not the texture image).

Once the window coordinates of the eight brick points are known, their bounding box is tested for intersection with the area of each LCD in the grid. The intersections with the screen of a node define a rectangular subvolume whose content we load from hard disk. The wavelet representation of the octree bricks allows us to load the subvolume at a detail level, that will not exceed a predefined number of voxels n. The wavelet level for a subvolume of dimensions dx, dy, dz is calculated as follows:

$wavelet_level = 0;$
$cMod = 1;$
while $(dx/cMod) \cdot (dy/cMod) \cdot (dz/cMod) > n$ **do**
  $wavelet_level{+}{+};$
  $cMod = 2 \cdot cMod;$
end while

We limit textures to $n = 16.7 \times 10^6$ voxels. A three-dimensional RGBA texture containing that many voxels has a size of 64MB. Notice, that we have to store three textures at each node: One high-resolution texture for each of the two monitors that are connected to a node and one common low-resolution fallback texture containing the entire data set.

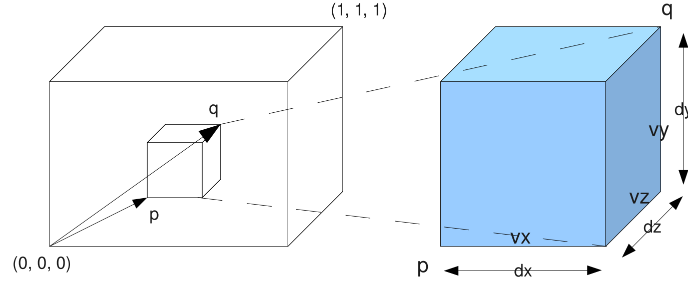


Fig. 4. Mapping information.

6.2 Assembling the Volume

Now that we have buffers containing different regions of the data set at different levels of resolution, the next step is to render them using the texture mapping approach discussed in section 5.1. Therefore, data chunks have to be assembled to form the entire volume. Since buffers represent 3D textures, one can think of two approaches to assemble them: The first approach maps each subvolume to its own stack of view-aligned proxy geometry, whose size and position within object space depends on the particular subvolume. The second approach maps each subvolume to the same (full screen) stack of view-aligned proxy geometry but scales and translates each texture so that it appears at the right position when being mapped. We chose the second approach, because it can easily be realized with the meta information that comes with each buffer and is provided by the data structure that implements the wavelet-octree (see Fig. 4).
The following facts are known about each subvolume: vx, vy and vz denote the buffer's axis dimensions. As explained above, $vx \cdot vy \cdot vz \leq 16.7 \times 10^6$ holds. Further, the rectangle's two diagonal points $p = (px, py, pz)$ and $q = (qx, qy, qz)$ are known. Their coordinates are normalized with respect to the full size of the data set, so $(0, 0, 0)$ and $(1, 1, 1)$ span the entire volume. p and q allow us to compute the actual portion of the data set, that is represented by all $vx \cdot vy \cdot vz$ voxels through $dx = qx - px$, $dy = qy - py$ and $dz = qz - pz$. Furthermore, p provides us the normalized offset within the entire volume with respect to the origin $(0, 0, 0)$.
Now, the calculation of appropriate scaling factors and translation vectors to place the texture at the right location in the volume is straight forward.

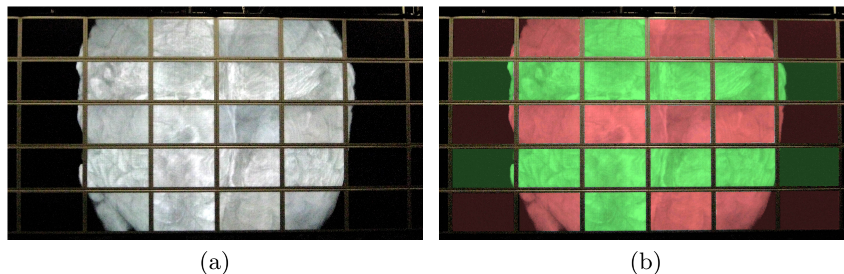
7 Results

Limiting the size of textures to 16.7×10^6 voxels, i.e., 64MB for an RGBA texture, does not seem logical considering the fact that each graphics cards is equipped with 256MB VRAM. However, each node maintains three textures, all with a maximum size of 64MB, resulting in a total of 192MB. Another 31.25MB are

used for the framebuffer of each display supporting a resolution of 2560×1600 pixels. Thus, only 32.75MB remain for additional operations. The calculation shows that a limit of 64MB per texture results in an optimal use of the available video resources. In fact, even the main memory workload of a node is kept low. Theoretically, the application only needs to allocate main memory for all three texture buffers. In practice about twice the amount of memory is needed due to the internal management of data. If more physical memory was added to the graphics cards, these observations would scale accordingly, and the visual quality of the displayed image would further improve. A variety of factors influence the quality of visualizations:

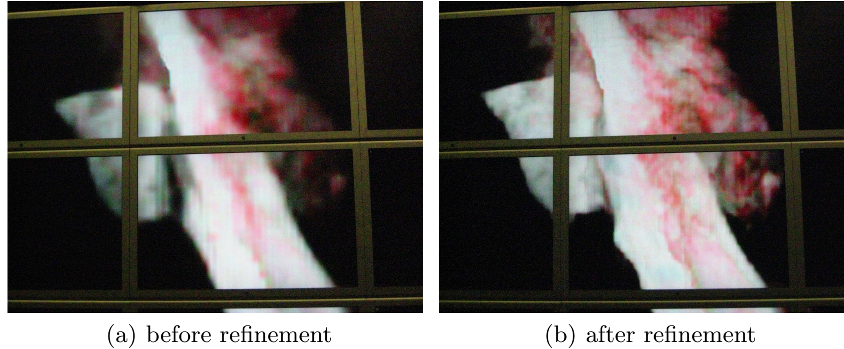
- Data sets themselves influence quality in that their dimensions partly determine the level-of-detail. Data sets consisting of many slices and therefore having a large extension in z-direction will not be displayed at the same detail level as data sets with fewer slices.
- The octree depth has a big influence on the final result. An octree of depth one partitions the entire volume into eight bricks, resulting in a rather coarse subdivision for the clipping process. An octree of depth two, however, produces 64 bricks and increases the quality since frustum clipping is able to extract smaller subvolumes that better represent the actual portion of data to be displayed.
- Last but not least, the current view has a strong effect on the detail level. Zooming-in maps fewer octree bricks to a tile and can therefore increase the level-of-detail. Rotations and translations have a similar influence.

Fig. 5(a) depicts a typical multi-resolution scene, where each monitor displays at its own tile-specific wavelet level. Note, that multi-resolution in this case refers to potentially different wavelet levels between tiles and that the level of detail within a tile is constant. The data that is visualized is a histology data set of a human cadaver brain. In order to illustrate the wavelet distribution within the wall, screens have been color coded in Fig. 5(b). Tiles marked red correspond to regions of wavelet level 2, meaning that these LCDs display the data set at



(a) (b)

Fig. 5. (a) Multi-resolution rendering of the Toga data set. (b) Wavelet level distribution. Red corresponds to tiles of level 2 ($1/64 = 1.5\%$ of original quality), green to level 1 ($1/8 = 12.5\%$ of original quality).



(a) before refinement (b) after refinement

Fig. 6. Close-up before and after the refinement. The picture shown in the lower image reveals a lot more structural details after the refinement.

$1/64 = 1.5\%$ of the original resolution. Green tiles correspond to areas of wavelet level 1, i.e., they display $1/8 = 12.5\%$ of the original structure. The close-up in Fig. 6 illustrates the difference between these two levels for a human mandible data set. In the upper picture each LCD displays at a wavelet level of 2, which corresponds to the detail level of the fallback texture. After refinement, the upper two LCDs switch to wavelet level 1, thereby increasing the level of detail by a factor of 8 and revealing more anatomical structures.

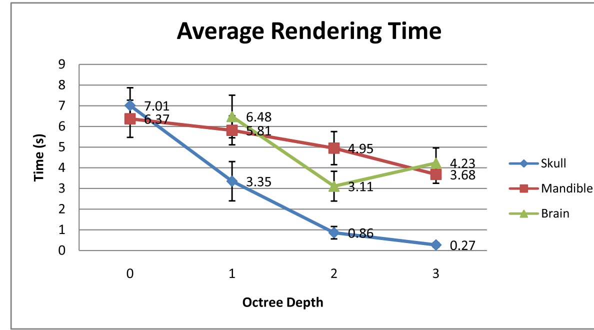
8 Analysis

The multi-resolution data structure is computed in a two-stage preprocessing step. In the first stage, a tool called *VolCon* computes the octree structure described in section 3.1, so that an input data set, consisting of a set of files storing slice information, is stored as a set of files storing octree bricks. In the second stage, another tool called *Compress* calculates the wavelet representation described in section 3.2 based on the results of stage one. Octree depth and maximum level of wavelet compression are passed as user-defined parameters. Generally, deeper octrees generate smaller brick files which lead to a crucially more accurate approximation of the subvolume of a tile. However, increased clipping accuracy comes cost of more file accesses. Table 1 illustrates the preprocessing time for various data sets with an octree depth of two and two levels of wavelet compression on a Core2Duo laptop machine with 2GB RAM.
We measured the time it takes each node to clip and render a scene (after interaction) using different octree depths for multiple data sets. Furthermore, we monitored the wavelet level at which each tile displays the scene. A fixed perspective was used during measurements, i.e., the orientation and zoom level remained unchanged. The results are depicted in Fig. 7. Fig. 7(a) shows that the average time it takes to clip and render a scene decreases with increasing octree depth and thereby increasing number of bricks, except one outlier for the brain data set. The overhead of loading unnecessary parts of the data set

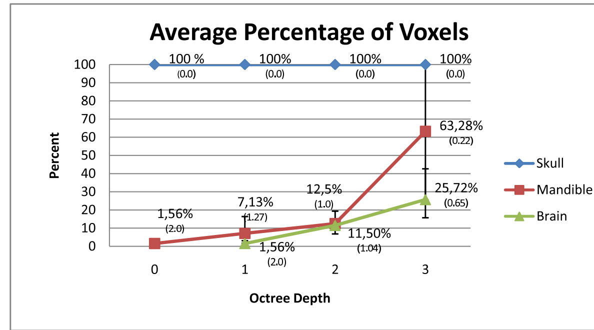
decreases because clipped subvolumes better match the data set portion, that a tile has to display. A possible explanation for the outlier is that there is a sweep point where the overhead caused by additional I/O operations overweighs the benefits of a better subvolume approximation. Fig. 7(b) shows that at the same time, with increasing octree depth, the average detail level of the display wall increases, i.e., the average wavelet level over all fifty tiles gets smaller, because the available texture memory is used more efficiently to store structural details of the volume. The detail level of the skull data set remains constant with increasing octree depth because the volume is small enough to fit entirely into the graphics card's texture memory. Note, that the real-valued wavelet levels in Fig. 7(b) result from averaging the discrete values for each tile. Though these compression levels cannot be achieved in practice, they give a good theoretical value describing the overall level of detail comprised in all fifty LCD panels. Further note, that the partly asymmetric variances depicted in Fig. 7(b) are due to the fact that the percentage of voxels scales non-linearly with the wavelet level, i.e., the level of detail comprised at level i is 1/8th of level $i-1$. In general, both the average rendering time and the level of detail are positively affected by a deeper octree. Our studies show that an octree depth of four in most cases is sufficient to guarantee almost interactive behavior when working with data sets.
In earlier experiments we investigated the data structure's capabilities when accessing volume data for a non-distributed application [35]. Fig. 8 compares a system using the octree-wavelet data structure to a system that uses neither spatial subdivision nor a multi-resolution representation. The unoptimized system loads the entire data by directly accessing a set of files storing slice information, i.e., slices are loaded sequentially into main memory. Fig. 8(a) compares the time it takes to load the entire mandible data set slice-wise into RAM to the time it takes to load at different wavelet levels at a fixed octree depth of three. Though the time difference observed for the system using the octree-wavelet structure is marginal between wavelet levels, loading is less than 7% of the average time measured to load the entire cross-sectional data.
Fig. 8(b) illustrates the influence of octree depth on the loading time for the skull data set. It can be observed that loading takes longer with increasing octree depth, which is due to the increased number of I/O operations that have to be performed and that are generally considered to be slow. This pattern also holds for various wavelet levels.
The results of Fig. 8(b) and Fig. 7(a) seem contradictory. Whereas a higher octree depth leads to a better approximation of subvolumes during frustum clipping

Data Set	**Dimensions** ($x \times y \times z$)	**Size**(MB)	**VolCon**	**Compress**
Skull	$256 \times 256 \times 113$	28.25	0m 2.5s	0m 2.7s
Mandible	$1024 \times 1024 \times 374$	1496	2m 17.2s	4m 2.5s
Toga Brain	$1024 \times 1024 \times 753$	3012	5m 9.9s	9m 45.8s

Table 1. Data set characteristics and preprocessing times for octree depth 2 (64 bricks) and two levels of wavelet compression.



(a)



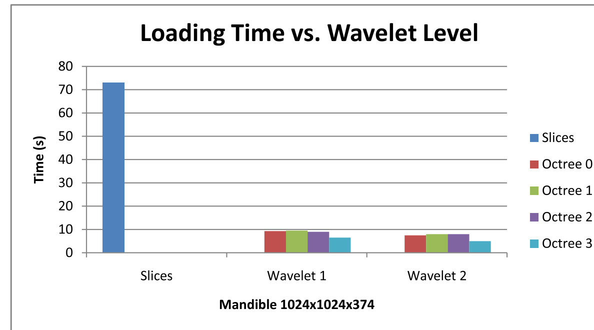
(b)

Fig. 7. (a) Average time it takes to perform frustum clipping and volume assembling. (b) Average percentage of original voxels being displayed (including average wavelet level).

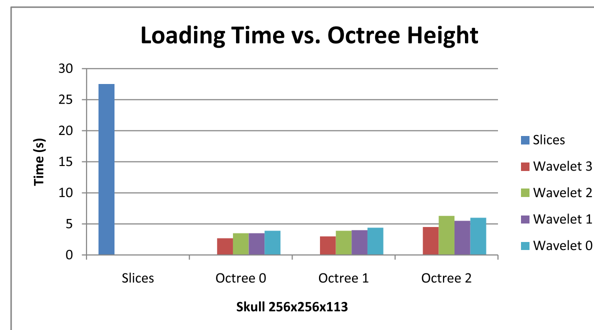
and reduces the amount of data that has to be loaded, the increased number of file operations works against this effect. However, the benefits of having a smaller clipping volume overweigh the negative effect of additional I/O operations.

9 Discussion

According to Molnar's [33] classification of distributed rendering algorithms the system presented in this paper implements a static sort-first algorithm. A general advantage of the sort-first strategy is the fact that communication overhead and network traffic are kept low. This is an important characteristic for the visualization of large-sized data sets because it prevents the network from being a performance bottleneck.
The data structure employed in this paper including the multi-resolution approach resembles a scenegraph-based system [1]. Scenegraphs also perform frustum culling in order to reduce scene complexity and they implement LOD techniques that often estimate an object's level of detail based on its distance to



(a)



(b)

Fig. 8. (a) Loading time vs. wavelet level. (b) Loading time vs. octree height.

the viewing plane. However, scenegraphs are generally oriented towards geometric data and not so much suited for managing volumetric data sets. To our knowledge, no scenegraph-based system running on a distributed display cluster exploits resources for direct volume rendering with the efficiency of the technique we presented in this paper.
Note, that 3D texture mapping results in uniform sampling along the viewing ray. If one wants to sample adaptively, for example based on the distance to the eye point, this can be achieved by raycasting the volume (see section 5.1).
The algorithm we presented scales with the processing power and size of the display system. The higher the number of tiles the more compute nodes are required. If the number of compute nodes increases the subvolumes get smaller, because the relative area covered by a monitor decreases. As a consequence, the quality of visualizations improves, since the same amount of resources is used to render smaller subvolumes. Likewise, if the amount of video memory increases, the buffer sizes can be adapted in order to store more detail information. Instead of allocating 64MB for texture objects, larger memory blocks would improve the average wavelet level of tiles and thus also increase the quality of renderings.

So far, when talking about "increased quality" we refer to a higher resolution. However, visual quality also encompasses contrast, color, and shading. Thus far, these topics have been mostly untouched in our work. In the future it might be interesting to investigate how to incorporate them into our system. For instance, the human visual system is strongest in the center field of view. Objects in the periphery are perceived with less detail. In order to account for this, one could apply a focus+context technique to create areas of high contrast with complex shading in the center of the tiled display and fuzzy regions in the periphery, similar to what Baudisch et al. [4] did with the resolution of the focus+context screen.
Our display wall is a monitor-based display cluster that combines the images of multiple LCD panels to form one large high-resolution display area. In contrast, projector-based tiled displays combine the images of a set of computer projectors [26, 22] and can easily exceed the dimensions of monitor-based systems. A major advantage of projector systems is their ability to form truly seamless displays [27, 28] while monitor-based systems inherently suffer from monitor bezels causing discontinuities in visualizations. At the moment our implementation does not cope with sharp transitions between adjacent tiles due to different resolution levels. We found that the benefit of perceiving finer details of the data set outweighs these slight optical confusions. A user study we conducted [12] showed that sharp transitions between different levels of resolution on a tiled wall do not cause as much confusion as mismatches in brightness and color between screens. The problem of discontinuities caused by monitor bezels requires deeper investigation in the context of human-computer interaction. Software [25] as well as hardware dependent solutions [12] have already been proposed.

10 Conclusion

In this paper we presented a method to effectively visualize volumetric data sets on tiled displays. Our implementation is based on a spatial subdivision scheme using octrees and incorporates a multi-resolution wavelet representation of the data. We were able to show that our approach increases the quality of visualizations by individually exploiting the resources of each rendering node. The system is interactive and allows users to explore data sets nearly in a real-time fashion. Our implementation was tested on a fifty tile display that is driven by twenty-five rendering nodes. In the future, we plan to investigate modifications of our data structure and implement memory caching and prefetching strategies to minimize I/O latencies. Additionally, we want to exploit further features of current state-of-the-art GPUs.

11 Acknowledgements

This work was supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of the International Graduate School (International Research Training Group, IRTG 1131) in Kaiserslautern, Germany. The

authors would like to thank Stephen F. Jenks and Sung-Jin Kim (University of California, Irvine), Falko Kuester and Kai-Uwe Doerr (University of California, San Diego) and the National Science Foundation for their support.

References

1. OpenSG, http://www.opensg.org/
2. Amin, M.B., Grama, A., Singh, V.: Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In: PRS '95: Proceedings of the IEEE symposium on Parallel rendering. pp. 7–14. ACM, New York, NY, USA (1995)
3. Bajaj, C., Ihm, I., Park, S., Song, D.: Compression-Based Ray Casting of Very Large Volume Data in Distributed Environments. In: HPC '00: Proceedings of the The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 2. p. 720. IEEE Computer Society, Washington, DC, USA (2000)
4. Baudisch, P., Good, N., Stewart, P.: Focus plus context screens: combining display technology with visualization techniques. In: UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology. pp. 31–40 (2001)
5. Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., Cruz-Neira, C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. In: VR '01: Proceedings of the Virtual Reality 2001 Conference (VR'01). p. 89 (2001)
6. Cabral, B., Cam, N., Foran, J.: Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In: VVS '94: Proceedings of the 1994 Symposium on Volume Visualization. pp. 91–98. ACM, New York, NY, USA (1994)
7. Calderbank, A.R., Daubechies, I., Sweldens, W., Yeo, B.L.: Wavelet transforms that map integers to integers. Appl. Comput. Harmon. Anal. 5(3), 332–369 (1998)
8. CAVELib Application Programmer Interface (api), http://www.mechdyne.com/integratedSolutions/software/products/CAVELib/CAVELib.htm
9. Chen, H., Clark, D.W., Liu, Z., Wallace, G., Li, K., Chen, Y.: Software Environments for Cluster-Based Display Systems. In: CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid. p. 202. IEEE Computer Society, Washington, DC, USA (2001)
10. Deller, M., Thelen, S., Steffen, D., Olech, P., Ebert, A., J.Malburg, Meyer, J.: A Highly Scalable Rendering Framework for Arbitrary Display and Display-in-Display Configurations. In: Proceedings of the 2009 International Conference on Computer Graphics and Virtual Reality, CGVR'09 (2009)
11. Doerr, K.U., Kuester, F.: http://vis.ucsd.edu/~cglx/
12. Ebert, A., Thelen, S., Olech, P.S., Meyer, J., Hagen, H.: Tiled++: An Enhanced Tiled Hi-Res Display Wall. IEEE Transactions on Visualization and Computer Graphics 16(1), 120–132 (2010)
13. Elvins, T.T.: Volume Rendering on a Distributed Memory Parallel Computer. In: VIS '92: Proceedings of the 3rd Conference on Visualization '92. pp. 93–98. IEEE Computer Society Press, Los Alamitos, CA, USA (1992)
14. Garcia, A., Shen, H.W.: An Interleaved Parallel Volume Renderer with PC-clusters. In: EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization. pp. 51–59. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002)

15. The Geowall2., http://www.evl.uic.edu/cavern/optiputer/geowall2.html
16. Heirich, A., Moll, L.: Scalable Distributed Visualization using off-the-shelf Components. In: PVGS '99: Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics. pp. 55–59. IEEE Computer Society, Washington, DC, USA (1999)
17. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. ACM Trans. Graph. 21(3), 693–702 (2002)
18. KIST Imaging Media Research: IMRC Wiki: The NAVER Framework, http://www.imrc.kist.re.kr/wiki/NAVER_Framework
19. Kruger, J., Westermann, R.: Acceleration Techniques for GPU-based Volume Rendering. In: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03). p. 38. IEEE Computer Society, Washington, DC, USA (2003)
20. LaMar, E., Hamann, B., Joy, K.I.: Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In: VIS '99: Proceedings of the Conference on Visualization '99. pp. 355–361. IEEE Computer Society Press, Los Alamitos, CA, USA (1999)
21. Levoy, M.: Efficient Ray Tracing of Volume Data. ACM Trans. Graph. 9(3), 245–261 (1990)
22. Li, C., Lin, H., Shi, J.: A Survey of Multi-Projector Tiled Display Wall Construction. In: ICIG '04: Proceedings of the Third International Conference on Image and Graphics. pp. 452–455. IEEE Computer Society, Washington, DC, USA (2004)
23. Lombeyda, S., Moll, L., Shand, M., Breen, D., Heirich, A.: Scalable Interactive Volume Rendering using off-the-shelf Components. In: PVG '01: Proceedings of the IEEE 2001 Symposium on Parallel and large-data Visualization and Graphics. pp. 115–121. IEEE Press, Piscataway, NJ, USA (2001)
24. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH Comput. Graph. 21(4), 163–169 (1987)
25. Mackinlay, J.D., Heer, J.: Wideband displays: mitigating multiple monitor seams. In: CHI '04: CHI '04 extended abstracts on Human factors in computing systems. pp. 1521–1524. ACM, New York, NY, USA (2004)
26. Majumder, A., Brown, M.S.: Practical Multi-projector Display Design. A. K. Peters, Ltd., Natick, MA, USA (2007)
27. Majumder, A., He, Z., Towles, H., Welch, G.: Achieving color uniformity across multi-projector displays. In: VIS '00: Proceedings of the conference on Visualization '00. pp. 117–124. IEEE Computer Society Press, Los Alamitos, CA, USA (2000)
28. Majumder, A., Stevens, R.: Perceptual photometric seamlessness in projection-based tiled displays. ACM Trans. Graph. 24(1), 118–139 (2005)
29. Max, N.: Optical Models for Direct Volume Rendering. IEEE Transactions on Visualization and Computer Graphics 1(2), 99–108 (1995)
30. Meyer, J., Borg, R., Hamann, B., Joy, K., Olsen, A.: Network-Based Rendering Techniques for Large-Scale Volume Data Sets. In: Farin, G., Hamann, B. and Hagen, H., eds., Hierarchical and Geometrical Methods in Scientific Visualization. pp. 283–296. Springer-Verlag, Heidelberg, Germany (2002)
31. Meyer, J.: Interactive Visualization of Medical and Biological Data Sets. Ph.D. thesis, University of Kaiserslautern (1999)
32. Meyer, J., Gelder, S., Heiming, C., Hagen, H.: Interactive Rendering - A Time-Based Approach. In: SIAM Conference on Geometric Design (1997)
33. Molnar, S., Cox, M., Ellsworth, D., Fuchs, H.: A Sorting Classification of Parallel Rendering. Tech. rep., Chapel Hill, NC, USA (1994)

34. National Library of Medicine: The Visible Human Project, http://www.nlm.nih.gov/research/visible/visible_human.html
35. Nguyen, H.T.: Large-scale Volume Rendering using Multi-resolution Wavelets, Subdivision, and Multi-dimensional Transfer Functions. Ph.D. thesis, University of California, Irvine (2008)
36. Ni, T., Schmidt, G.S., Staadt, O.G., Ball, R., May, R.: A Survey of Large High-Resolution Display Technologies, Techniques, and Applications. In: VR '06: Proceedings of the IEEE Conference on Virtual Reality. p. 31. IEEE Computer Society, Washington, DC, USA (2006)
37. Palmer, M.E., Totty, B., Taylor, S.: Ray Casting on Shared-Memory Architectures: Memory-Hierarchy Considerations in Volume Rendering. IEEE Concurrency 6(1), 20–35 (1998)
38. Plate, J., Tirtasana, M., Carmona, R., Fröhlich, B.: Octreemizer: a hierarchical approach for interactive roaming through very large volumes. In: VISSYM '02: Proceedings of the Symposium on Data Visualisation 2002. pp. 53–ff. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002)
39. Raffin, B., Soares, L.: PC Clusters for Virtual Reality. In: VR '06: Proceedings of the IEEE Conference on Virtual Reality. pp. 215–222. IEEE Computer Society, Washington, DC, USA (2006)
40. Reiners, D.: OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications. Ph.D. thesis, Technische Universität Darmstadt (2002)
41. Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., Ertl, T.: Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In: HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. pp. 109–118. ACM, New York, NY, USA (2000)
42. Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., Strasser, W.: Smart hardware-accelerated Volume Rendering. In: VISSYM '03: Proceedings of the symposium on Data visualisation 2003. pp. 231–238. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2003)
43. Rusinkiewicz, S., Levoy, M.: Qsplat: a multiresolution point rendering system for large meshes. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques. pp. 343–352. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
44. Schaeffer, B., Goudeseune, C.: Syzygy: Native PC Cluster VR. In: VR '03:Proceedings of the IEEE Virtual Reality 2003. p. 15 (2003)
45. Schwarz, N.: Distributed Volume Rendering of Very Large Data on High-Resolution Scalable Displays. Master's thesis, University of Illinois at Chicago (2007)
46. Schwarz, N., Venkataraman, S., Renambot, L., Krishnaprasad, N., Vishwanath, V., Leigh, J., Johnson, A., Kent, G., Nayak, A.: Vol-a-Tile - A Tool for Interactive Exploration of Large Volumetric Data on Scalable Tiled Displays. In: VIS '04: Proceedings of the Conference on Visualization '04. p. 598.19. IEEE Computer Society, Washington, DC, USA (2004)
47. Soares, L.P., Zuffo, M.K.: JINX: an X3D browser for VR immersive simulation based on clusters of commodity computers. In: Web3D '04: Proceedings of the Ninth International Conference on 3D Web Technology. pp. 79–86 (2004)
48. Weiler, M., Westermann, R., Hansen, C., Zimmermann, K., Ertl, T.: Level-of-Detail Volume Rendering via 3D Textures. In: VVS '00: Proceedings of the 2000 IEEE Symposium on Volume Visualization. pp. 7–13. ACM, New York, NY, USA (2000)

49. Westover, L.: Interactive Volume Rendering. In: VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization. pp. 9–16. ACM, New York, NY, USA (1989)
50. Westover, L.: Footprint Evaluation for Volume Rendering. In: SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques. pp. 367–376. ACM, New York, NY, USA (1990)
51. Wilson, O., VanGelder, A., Wilhelms, J.: Direct Volume Rendering via 3D Textures. Tech. rep., Santa Cruz, CA, USA (1994)