Eurographics Symposium on Parallel Graphics and Visualization (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems

C. Müller, M. Strengert and T. Ertl

Institute for Visualization and Interactive Systems, University of Stuttgart

Abstract

In this paper, we present a sort-last parallel volume rendering system based on single-pass volume raycasting performed in the fragment shader unit. The architecture is aimed for displaying data sets that utilize the total distributed texture memory at interactive framerates. We use uniform texture bricks that are distributed by means of a kd-tree to employ object space partitioning. They are further used for implementing empty-space-skipping and a load balancing mechanism, which also makes use of the kd-tree, to increase the overall performance of the rendering system. Performance numbers are given for a mid-range GPU-cluster system consisting of eight render nodes with an Infiniband interconnection.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Distributed/network graphics I.3.3 [Computer Graphics]: Viewing algorithms C.2.4 [Distributed Systems]: Distributed applications

1. Introduction

The ongoing advances in volumetric data acquisition, e. g. medial imaging or oil and gas exploration, as well as in numerical simulation generate constantly larger data sets that pose a steady challenge to volume visualization. Cluster computers have become a common way to address this high demand for rendering power as they can be scaled according to the requirements. Likewise, graphics hardware makes more and more use of parallelism, particularly regarding the fragment shader units, and offers more and more flexibility. We try to combine both types of parallelism by distributing a fragment-shader-based volume raycaster on a GPU-cluster.

In detail, our contributions are: First, we extend fragment-program-based raycasting for seamless bricking without the need for additional render passes. This makes the overhead of bricking relatively small compared to other raycasting or slice-based approaches and therefore allows for further reducing the size of the bricks. Second, we employ empty-space-skipping on a per brick basis taking advantage of the pre-integration table used for rendering. Third, a load balancing approach built upon a kd-tree-based data structure in object space is applied to cope with imbalances caused by the characteristics of the rendered data set in connection with empty-space-skipping or caused by inhomogeneous cluster environments. This seems even more important to us, since the rendering performance of GPU-based raycasting is solely dependent on the graphics hardware fragment processing power, which is under rapid development, increasing the likelihood of inconsistent cluster setups. Finally, we discuss our system's performance on a mid-range GPU-based cluster system using homogeneous as well as inhomogeneous setups in order to evaluate the presented load balancing technique.

2. Related Work

The field of distributed volume rendering is an active field of research and a great variety of publications can be found in literature dealing with different aspects of this issue. Molnar et al. [MCEF94] provide a taxonomy that allows for dividing parallel architectures into the three categories *sort-first*, *sort-middle*, and *sort-last* depending on the location of distribution in the rendering pipeline.

Early techniques for parallel volume rendering were mostly built upon CPU-based raycasting. Ma et al. [MPHK93] presented a data distributed rendering system introducing the *Binary-Swap* compositing scheme. At the same time Neuman [Neu93] published a parallel volume

© The Eurographics Association 2006.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

raycasting architecture that used *Direct-Send* for image compositing. Based on off-the-shelf commodity PC-clusters equipped with graphics hardware Magallón et al. [MHE01] and Bajaj et al. [BPT02] both presented parallel volume rendering systems. These methods were later on extended by the use of wavelet compression [SMW*04] and load balancing based on a hierarchical space-filling curve [WGS04]. More recently, Allard and Raffin [AR05] presented the shader-based framework *FlowVR* for parallel rendering using GPU-Clusters. They present a sort-first distribution scheme for raycasting as application of the framework in order to drive a multi-tile projection wall.

In order to utilize the graphics hardware performance for volume raycasting, Krüger et al. [KW03] proposed a multi-pass rendering algorithm. In each pass, the integration along the rays is advanced by a single sampling interval and the actual position is stored for the following pass in textures in a ping-pong fashion. With the advent of dynamic flow control for fragment shaders it is possible to evaluate the complete integration along a ray in a single pass [Sch05, SSKE05]. Hadwiger et al. [HSS*05] extended these methods by using adaptive texture maps to include empty-space-skipping and save texture memory, but their technique is not easily applicable to bricks spread across various cluster nodes.

3. GPU-based Raycasting

In order to approximately evaluate the *volume rendering integral* on a GPU for each pixel, discrete positions along the ray get sampled and their associated color and opacity values derived from *classification* are accumulated. A commonly used technique is to render tri-linear interpolated textured slices to account for the data sampling at a given depth for all pixels in parallel and set the blending functionality to match the needed accumulation term [CCF94]. The integration along the rays is driven by generating new fragments for each sampling position. With the availability of dynamic flow control in the fragment processor of recent GPUs an alternative approach becomes possible that performs the integration along a ray completely inside the shader program [Sch05, SSKE05]. One obvious advantage of such an approach is that less fragments need to be generated, since one fragment per ray is sufficient compared to one fragment per sample point in slice-based approaches.

For an application to object space partitioned parallel volume rendering and even more in connection with additional acceleration structures based on further bricking (see section 3.2) the most important feature of GPU-based raycasting is the very low overhead introduced for rendering the splitted data set. This overhead directly affects the scalability of the distributed architecture as well as the benefit of including brick-based empty-space-skipping. In a bricked data set the total number of vertices needed for rendering the proxy geometry of a slice-based approach highly increases on a per slice basis, while for raycasting the required amount of vertices solely depends on the number of bricks used, since only their front faces need to be rendered to set up the rays. This keeps the load on the vertex processor very low and prevents transforming vertices to become the limiting factor when using very small brick sizes. In addition, less CPU time is necessary, since the position of these vertices can be easily derived from a brick's location inside the volume. Compared to slice-based approaches, that need to calculate intersection between each slice and the brick bounding geometry, this can be implemented more efficiently and keeps the overhead for the CPU very low.

3.1. Basic Concept

Having the possibility of loops and conditionals inside a fragment shader allows for completely traversing a ray through a volume sampling the data set at equidistant intervals and evaluating the *volume rendering integral* in a single render pass.

As described in [SSKE05], the actual raycasting is performed in texture space in order to avoid costly transformations of texture coordinates during the ray traversal. Therefore, a mapping from object space to texture space is necessary for setting up the rays correctly. The geometric size e_c of a volume is determined by the number of slices S_c in each dimension $c \in \{x, y, z\}$ and the corresponding slice thickness D_c. Since in texture space the volume is addressed completely independently of the values S_c and D_c using a fixed interval of $[0, 1]$, a scaling is necessary to transform a geometric point inside a volume to its corresponding position in texture space. For non-uniform volumes it is significant to also map vectors from one space to the other to assure constant sampling distances independent of the direction of the ray. In order to allow for the same transformation to be applied to points as well as to vectors, we define the geometric position of the volume to lie in between $(0, 0, 0)^T$ and $(E_x, E_y, E_z)^T$ with

$$E_c = \frac{e_c}{\max\{e_x, e_y, e_z\}}$$

being the normalized volume size in each dimension. The mapping is then easily defined as multiplication with a factor of $F_c = 1/E_c$ for both positions and vectors.

For the ray setup only the front faces of the bounding geometry of a volume are rendered with the texture coordinates set to the corresponding position of the normalized volume as described above. The interpolated texture coordinates after transformation serve as first sample position on each ray and the direction is derived from additionally mapping the position of the camera to texture space. The fragment shader then traverses the ray as long as the sample position is still inside the volume, which is equivalent to check if the position in texture space is still located in the interval of $[0, 1]$. For each sample point the associated color and opacity val-

© The Eurographics Association 2006.

ues are obtained from *classification* and accumulated with front-to-back blending.

3.2. Bricking

The idea of subdividing a volume data set into a set of smaller data blocks is widely used for rendering very large data sets that cannot be processed as a whole by a single machine. The object space subdivision of the volume allows the distribution of the visualization task to multiple render nodes. Additionally, it permits the implementation of empty-space-skipping on each node, which can significantly increase the performance of the system (see Sec. 5.1).

In order to employ the concept of bricking, the volume data are split into uniform texture bricks and each of these bricks is processed like a separate volume following the basic concept, i. e. the front faces of the proxy geometry of each brick are drawn in depth sorted order. The final image is computed by alpha blending the resulting fragments.

As in prior slice-based distributed volume renderers this procedure causes problems at the boundaries of the bricks due to the tri-linear interpolation of the texels. A continuous interpolation of the volume data can be achieved by replicating the boundary voxels in adjacent bricks [SMW*04]. Additionally, a correct transition between the bricks requires the clamping of outbound rays. For the basic concept using only one brick the error introduced by ending the ray traversal at the last regular sampling point within the volume is neglectible and corresponds to the error also made by slice-based approaches. However, if the volume is split into multiple bricks, these borders are located somewhere in the middle of the data set and it will therefore cause significant artifacts like holes in isosurfaces, if the last sampling points do not lie on the back face of the proxy geometry. The computation of these last sampling points can be done

```
MOV scale, 0.0;

# Test for positions outside the extents
SUBC temp.x, volExtentMax, pos;
MOV scale.x (LT), temp.x;
SUBC temp.x, pos, volExtentMin;
MOV scale.x (LT), temp.x;

# Determine correction vector
DIV temp.x, scale.x, offset.x;
ABS temp.x, temp.x;

# Move back to max/min x-axis extent
MAD pos, -temp.x, offset, pos;
```

Figure 1: *Fragment program code for clamping the last ray sample position to the volume or brick extents. Only code for the x-axis is shown.*

in a way that is similar to the Sutherland-Hodgman polygon clipping: Each ray is clipped against the boundaries of the volume. Fig. 1 illustrates how the clipping against the x-axis is done in the fragment program. The resulting position of this operation must then be clipped against the y- and z-axis in the same way, which finally yields the intersection of the ray and the back face. An alternative way of acquiring the intersection between the rays and the back faces is an additional rendering pass: Krüger et al. [KW03] render the back faces to create a texture holding the exit points of the rays. Such a texture could be used to lookup the intersection point for each ray. However, we did not choose this way as additional render passes introduce higher overhead for small brick sizes.

Unfortunately, having one voxel overlap between the bricks and using the correct last sampling point on the back face is not sufficient for creating an artifact-free image when using pre-integration, since different sampling distances occur in this case. We use the technique for incremental preintregation tables presented by Weiler et al. [WKME03] to overcome this problem.

4. Parallel System Architecture

4.1. Viewer

Our system is a typical remote rendering system split in render nodes, that are responsible for generating the images, and a viewer application. The viewer shows the final image and handles user input events. Based on the user input, it creates render requests, which are sent to the cluster in order to refresh the image. Messages between the viewer and the cluster are sent over a TCP/IP network at the moment.

The current implementation of the viewer uses the GLUT and runs either on Windows or Linux PCs. However, the only graphics capability required for the viewer is drawing pixels into the framebuffer. Hence, the viewer could quite easily be ported to devices with limited graphics hardware like PDAs or systems without hardware accelerated graphics using only GDI or X functionality.

4.2. Render Nodes

The render nodes implement the GPU raycaster. Each node is responsible for raycasting a disjunct part of the data set consisting of one or more bricks that form a continuous and convex subvolume. The number of slices $S_c(\vec{b})$ that brick $\vec{b} = (b_x, b_y, b_z)^T$ consists of is computed from the total number of slices S_c and the user-defined number of bricks B_c in each dimension c:

$$S_c(\vec{b}) = \begin{cases} \left\lceil \frac{S_c}{B_c} \right\rceil & \text{if } b_c < B_c - 1 \\ S_c - (B_c - 1) \cdot \left\lceil \frac{S_c}{B_c} \right\rceil & \text{otherwise} \end{cases}$$

This creates mostly uniform bricks, but if the number of slices cannot be divided by the number of bricks, the bricks

© The Eurographics Association 2006.

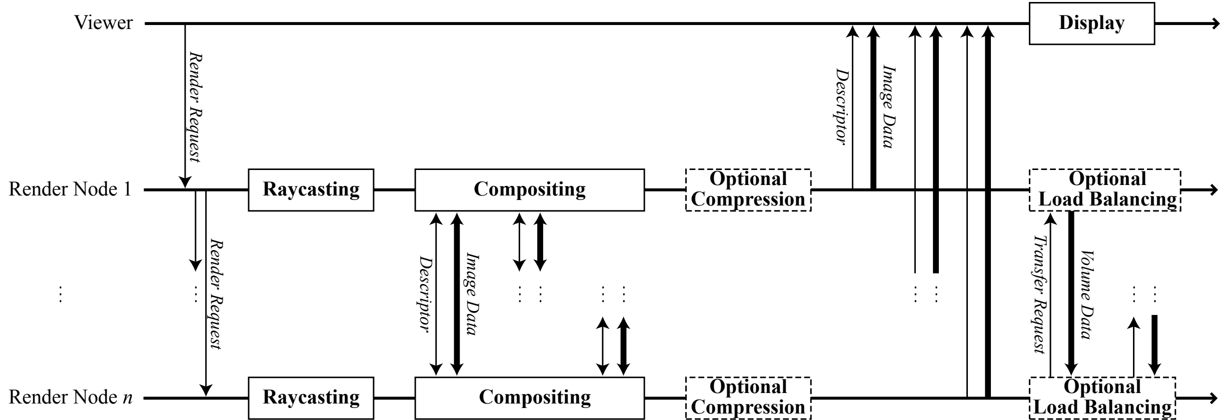

Figure 2: *Schematic view of the generation of a single frame.*

at the front, right and top border of the volume data set are smaller than the rest.

The object space subdivision of the data set is constructed by assigning bricks to a render node using a kd-tree, which guarantees that the partition on every node is convex regardless of the number of cluster nodes used. The kd-tree has two other advantages: When a layer of bricks is moved from one render node to another along the subdivision planes of the tree, the new partitions will also be convex, and the kd-tree can be used to determine the correct order when blending the images of the render nodes to create the final result [MPHK93].

The render nodes basically implement a server loop that waits for render requests from the viewer and processes them (see Fig. 2). The viewer sends the render request containing the necessary information to set up the volume for the next frame to the first node, which acts as some kind of “master node”. The master node then broadcasts — we use MPI as communication middleware — the request message to all other nodes, which has the advantage that the viewer and the slow network connection between the viewer and the cluster are relieved. Each render node then raycasts its subvolume independently. The final image is constructed by compositing the images of all nodes and sent back to the viewer for display. Compositing is done in software using the *Direct-Send* communication scheme [Neu93]. To prevent blending empty pixels and to limit the bad performance impact of `glReadPixels`, we implement an *sl-sparse* system based on the projection of bounding boxes.

As we expect only a narrow-band connection for the transport of the final image from the cluster to the viewer, it is possible to compress the image using the LZO real time compression library before sending it over this “last mile”. The computation time for compressing the image is extremely low and the size of the compressed image is normally about 10 % of the original data.

Beside the possibility to compress the image, we have implemented two different ways of sending the image from the cluster to the viewer. The first is collecting the whole image on one of the cluster nodes and sending it from this node to the viewer. The advantage of this method is that the fast interconnect between the cluster nodes can be used to combine the results of the compositing on each node and the transfer of one large package over the possibly slower network should be more efficient than sending partial images from n nodes.

Sending the partial images from the node that did the compositing directly to the viewer is the second method. In that case, the viewer receives n smaller packages from n nodes one after the other. The advantage of this method is, that the cluster nodes are not synchronized after compositing and become therefore less idle.

4.2.1. Distributed Volume Raycasting

The distributed data set is basically rendered by processing each brick in a depth sorted order like a separate volume following the basic concept. However, some small changes in the framework and the fragment shaders have to be done.

First of all, all extents have to be computed on a per brick basis. With $e_c^b = \lceil S_c/B_c \rceil \cdot D_c$ being the extent of a full size brick and $e_c(\vec{b}) = S_c(\vec{b}) \cdot D_c$ the extent of a brick $\vec{b}$, the normalized volume size of $\vec{b}$ in each dimension c is

$$E_c(\vec{b}) = \frac{e_c(\vec{b})}{\max\{e_x^b, e_y^b, e_z^b\}}.$$

As stated above, a correct tri-linear interpolation at the border of the bricks can only be achieved by having overlapping voxels. We realize this overlap by replicating in each brick one slice of the brick that is right, before or above the current one and shift the whole texture by half the width of

© The Eurographics Association 2006.

a texel. A brick therefore consists of

$$\tilde{S}_c(\vec{b}) = \begin{cases} S_c(\vec{b})+1 & \text{if } b_c < B_c - 1 \\ S_c(\vec{b}) & \text{otherwise} \end{cases}$$

slices with valid data, but only $(\tilde{S}_c(\vec{b})-1)$ of these slices are visible. The fragment program must be aware of the replicated data, which can be achieved by changing the factor that does the transformation from geometric to texture space. If the volume texture of $\vec{b}$ has an actual dimension of $T_c(\vec{b})$, the new factor

$$F_c(\vec{b}) = \frac{\tilde{S}_c(\vec{b})-1}{T_c(\vec{b}) \cdot E_c(\vec{b})}$$

incorporates the transformation already described for the basic concept as well as the removal of the overlapping slice from the range of visible slices.

As the actual raycasting happens in texture space, i. e. within $[0,1]$ for each brick, the number of sampling points per brick is solely dependent on the user-defined sampling distance. Hence, the overall number of sampling points increases with the number of bricks used. To overcome this undesireable effect, the user-defined distance must be scaled depending on the number of bricks.

Texture coordinates and the geometric position of a vertex do not match in case of using multiple bricks. The texture coordinates must be computed by interpolation — especially, if a brick must be clipped against the near clipping plane. As the load on the vertex unit is not very high, we move this computation from the CPU to the vertex shader. We also move the computation of the ray direction from the fragment to the vertex program, which saves some instructions in the pixel shader. The result is passed as an additional set of texture coordinates and hence correctly interpolated between the vertices. The fragment program can directly access the direction vector and has no need for knowing the geometric position of the ray entry point any more.

5. Optimization

5.1. Empty-Space-Skipping

The concept of empty-space-skipping is widely used to accelerate the raycasting process in CPU-based renderers [Lev90]. While employing this technique directly in the fragment programs of GPU-based systems is quite costly as it requires an additional hierarchy of textures, that allow looking up whether a certain part of the volume will make any contribution to the final image, discarding empty bricks comes nearly for free. If the framework knows the minimum and maximum scalar value of each brick, it can determine whether a brick will create visible fragments for isosurface and direct volume rendering shaders in advance and possibly skip the whole brick.

The implementation of the skipping mechanism is

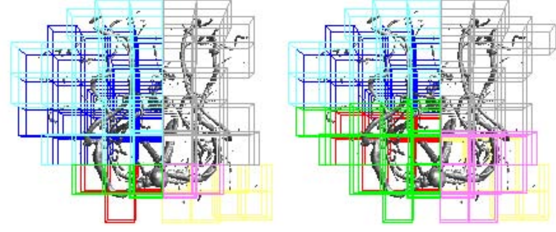

Figure 3: *The aneurism data set before (left) and after (right) load balancing. The color coding denotes the assignment to the different cluster computers. See also fig. III.*

straightforward for isosurface shaders: A brick can be discarded, if the current iso value is not between the minimum s_{min} and maximum s_{max} of this brick. For direct volume rendering shaders that allow interactively changing the transfer functions, a similar decision can be made using the pre-integration table. The pre-integration table returns for two scalar values s_f and s_b an approximation

$$\alpha_i \approx 1 - \exp\left(-\int_0^1 \tau((1-\omega)s_f + \omega s_b) d d\omega\right) \quad (1)$$

for the opacity of the i-th of uniform ray segments with a length of d [EKE01]. A brick can be skipped, if this opacity will surely be zero for all of its sampling points. That is the case, if α_i is zero for $s_f = s_{min}$ and $s_b = s_{max}$, because as negative opacities are not allowed, equation 1 can only yield zero, if all scalar values between s_f and s_b result in transparent fragments.

5.2. Load Balancing

For a maximum overall system performance, the work load should be well-balanced between the render nodes. However, asymmetrical data sets can cause a load imbalance — especially in conjunction with empty-space-skipping, which can lead in the worst case to nodes that need to raycast no brick at all, while other nodes have all the relevant parts of the data set. An inhomogenous cluster environment can also be the reason for a poorly balanced system, if some computers just do not have the capability to finish their task in the same time as the fastest can do. Finally, our kd-tree-based construction of the partitions has the inherent problem of assigning very different numbers of bricks to the nodes, if the number of cluster computers is not a power of two.

The system therefore tries to rebalance the work load between the nodes dynamically. Based on the assumption that the render time will not change very quickly, the render time of the current frame is used as a reference and bricks are moved from nodes with heavy load to nodes that finished the frame earlier. However, the possibilites for moving bricks from one node to another are limited by several con-

© The Eurographics Association 2006.

straints. First of all, the load balancing must ensure that the partitions on all nodes always form a continuous and convex subvolume, which can be guaranteed by moving a whole slice of bricks only along the subdivision planes in the kd-tree. Secondly, the load balancing must be fast as it must be done in the same thread as the raycasting due to the lack of thread-aware MPI implementations for most interconnections. Finally, the system should only move bricks over the network, if the exchange will surely improve the situation, as the transfer of bricks and the upload into the graphics memory of the target machine takes a significant amount of time. Most importantly, the system must avoid moving bricks from one side to the other in one frame and the other way around in the next.

The base for all load balancing operations is a local copy of the kd-tree that defines the volume partitions. The leave nodes of this tree represent a cluster computer and the bricks that are resident on this computer. After the times for rendering the current frame have been communicated between the nodes, the difference between the render times of the two halfspaces defined by any inner node of the kd-tree can be computed independently on each node.

When determining the need for load balancing, it must be taken into account that a node that is near the root of the kd-tree represents much more bricks than nodes deeper in the tree. Let s be the number of bricks that are direct neighbors of a subdivision plane on one side, i. e. the number of bricks that must be moved when bricks should be exchanged along this plane. Additionally, let t_l be the total render time for the left subtree and b_l the number of bricks in this subtree, t_r and b_r accordingly for the right subtree. The average render time for a brick in the tree represented by the current subdivision tree node then is $\bar{t} = (t_l + t_r)/(b_l + b_r)$. Assuming $t_l > t_r$, moving bricks along this subdivision plane makes sense, if

$$|(t_l - s\bar{t}) - (t_r + s\bar{t})| < t_l - t_r. \tag{2}$$

When rewriting inequation 2 as $|t_l - t_r - 2s\bar{t}| < t_l - t_r$ and using the prior assumption that $t_l - t_r \geq 0$ and the fact that the render time $s\bar{t}$ must also be positive, the lower limit for $2s\bar{t}$ is 0 and the upper limit $2(t_l - t_r)$. Therefore, moving bricks along a subdivision plane improves the balance, if

$$s\bar{t} < |t_l - t_r| \tag{3}$$

holds true. However, this condition does not take into account that moving bricks from one node to the other is very time consuming, which results in a performance decrease as bricks are moved too often. Let therefore c be the average time it takes to move a brick from one node to another. The system can measure c while it is running. Applying c directly on the left side of inequation 3 is not very useful as c is probably a quite high value and will consequently prevent most load balancing operations. Therefore, c is weighted with an additional factor f. For computing this weighting factor as

$$f = \begin{cases} f_l & \text{if } t_l > t_r \\ f_r & \text{otherwise} \end{cases}$$

each node in the kd-tree stores two values f_l and f_r, which model the observation that it is an undesired behavior if bricks are moved back from the right to the left subtree, if they have been moved from the left to the right in the last frame. If the system is e. g. moving bricks from left to right, the factor f_r for moving bricks into the opposite direction is increased by a user-defined amount f_o and the factor f_l for moving bricks into the same direction is decreased by f_s. By modifying these two values the user can control how aggressively the load balancing is moving bricks. Including the average cost c for moving a brick and the factor f, the condition for moving a slice of bricks finally is

$$s\bar{t} + sfc < |t_l - t_r|\,. \tag{4}$$

In order to address the problem of the very limited computation time that can be used for load balancing, the system does not try to balance the whole tree, but incrementally processes only one node a frame. For that, inequation 4 is evaluated for the root node of the kd-tree. If it holds true, i. e. the imbalance between the two subtrees is so large that moving a slice of bricks is very likely to improve the situation, one slice of bricks is transferred along the subdivision plane represented by the root node. The load balancing is then finished for the current frame and starts again at the root node after the next image has been rendered. If the imbalance is not that large, the other nodes of the kd-tree are tested in a preorder traversal, and after the first slice of bricks has been moved, the load balancing is suspended until the next frame. Fig. 3 illustrates the rearrangement of bricks made by the load balancing mechanism.

Despite all our efforts to prevent the system from alternately moving bricks along one subdivision plane, it is not possible to completely stop this behavior, mainly for two reasons: First, choosing f_o and f_s and an appropriate initial value for f_l and f_r is quite difficult as it depends on the data set itself, the number of bricks that are created and the hardware that is used. Second, if the values are too high, the system might not reach the best possible state of balancing because the transfer costs have increased so much that the last iterations of the load balancing seem to be useless. The system might then even get stuck in a configuration that is worse than the initial partitioning. We have therefore added a brick caching mechanism that makes it faster to move bricks back to the original node. If a node is moving a brick to a neighboring node, it keeps a copy of this brick in its main memory. We currently use a simple least recently used approach as replacement strategy. If the system then decides to move a cached brick back to the initial node, it is not transferred over the network but the local copy is reactivated. The caching mechanism requires additional messages to be sent, because the two nodes must negotiate which bricks must actually be sent and which are still cached. However, such a message is much smaller than a whole brick. Hence, the system performs better with caching and it is possible allow

© The Eurographics Association 2006.

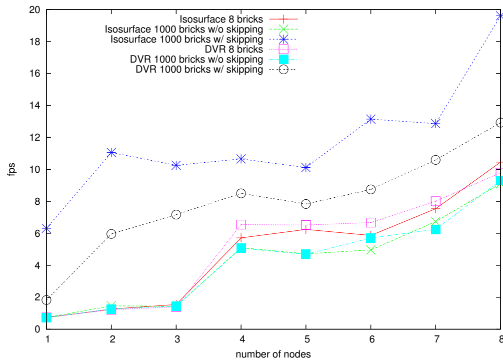

Figure 4: *Framerates for the aneurism data set.*

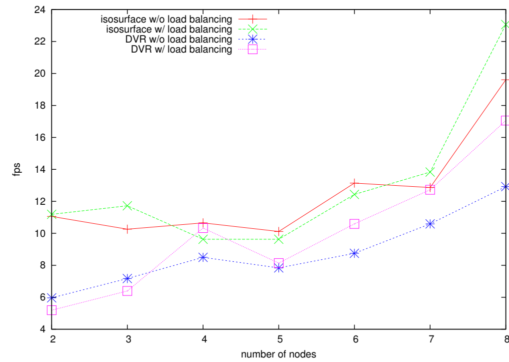

Figure 5: *Load balancing result with the aneurism data set.*

a more aggressive load balancing that can finally achieve a better balance between the nodes.

6. Results and Discussion

We conducted performance tests on our cluster of eight dual 2.1 GHz Opteron PCs with 4 GB RAM. The nodes have a GeForce 6800 Ultra with 256 MB of graphics memory and are connected with an Infiniband network. If not stated differently, the tests have been performed using a 512×512 viewport and a sampling distance assuring that each voxel is sampled at least once. We used a 512^3 aneurism and the $512^2 \times 999$ Christmas tree data set for testing.

The framerates were measured by the viewer application while the volume was rotated around the y-axis. Unless denoted differently, each render node sent its own part of the image directly to the viewer after compression. Sending compressed strips directly proved to be fastest in most cases, so all further tests use this transfer method.

Fig. 4 shows the results of the aneurism data set for an isosurface and a DVR shader. Without empty-space-skipping — no bricks can be skipped with only 2^3 bricks —, the data set does not fit into the graphics memory when using less than four nodes. It also becomes obvious, that node numbers that are powers of two generally perform better, because the partitioning made using the kd-tree is much more balanced. Isosurface shaders are normally faster, because they can stop sampling after the first isosurface was found, and more bricks can be discarded when using empty-space-skipping. E. g. 808 bricks can be discarded for the isosurface display, but only 596 for the DVR shader. The Christmas tree data set shows a very similar scaling behavior. Using four to seven nodes the framerate is around 7 fps, on eight nodes 11.6 fps are achieved.

We tested our load balancing mechanism using the aneurism data set with 10^3 bricks (see Fig. 5). Load balancing can, to a certain degree, resolve the problem of a bad initial partitioning caused by the kd-tree, provided that the transfer cost parameters are set appropriately. Finding these parameter values proved to be very difficult. In fact, we have several cases in which the performance was worse after load balancing than before, because the transfer costs grew too fast. With an increasing number of cluster nodes, the load balancing seems to work better, probably because the slices of bricks that have to be moved from one node to the other become smaller. To simulate an heterogenous cluster, we reduced the GPU clock speed on one of our nodes to 100 MHz. The framerate of the isosurface shader dropped to 6.2 fps, but increased again to 11.3 fps after load balancing. When using the DVR shader with the same load balancing parameters, the load balancing had nearly no effect.

In order to use the total texture memory of 2 GB, we used the $512 \times 512 \times 1877$ Visible Male data set, which requires including pre-computed gradients about 1,83 GB of texture memory. Using $6 \times 6 \times 12$ bricks, we reach 19.9 fps for the isosurface and 20.5 fps for the DVR shader. Empty-space-skipping works quite well with the isosurface shader: Only 40 % of the bricks must be rendered which results in a framerate of 48.5 fps. With $15 \times 15 \times 50$ bricks, only 16 % must be rendered, but the remaining 1837 cause a lot of overhead, so that the framerate only increases from 3.1 fps to 12.6 fps. A good compromise regarding the number of bricks therefore seems to be very important. Fig. I on the color plate shows an image, which has been rendered using a combined

	Empty-Space-Skipping	fps
1260^3	no	2.8
1260^3	yes	4.4
1260^3, half sampling rate	yes	8.1
1075^3	yes	4.9
1075^3, 1024×768 viewport	yes	3.2

Table 1: *Framerates of the large aneurism data sets.*

© The Eurographics Association 2006.

isosurface/DVR shader on a 768×1024 viewport. When sampling twice per voxel, 0.5 fps are reached.

As DVR shaders do not need gradients, we additionally tested two resampled versions of the aneurism data set with 1075^3 and 1260^3 slices. We used 10^3 bricks, which of 692 must be rendered when empty-space-skipping is enabled. Tab. 1 shows the results on eight cluster nodes.

7. Conclusion and Future Work

We presented a distributed GPU-based raycasting architecture for rendering uniformly bricked data sets. This brick structure is the basis for two optimization methods. First, an empty-space-skipping is applied to single-pass volume raycasting without the need for additional render passes and without introducing visible artifacts. Second, we addressed the problem of imbalanced work load across the cluster nodes caused by data set characteristics or inhomogenous cluster layouts with an object space balancing technique built upon a kd-tree. With the presented system it is possible to visualize data sets that make use of the complete distributed texture memory at 2.8 frames per second.

For future work we plan to further utilize the brick structure for adaptive sampling on the rays based on the standard deviation of the scalar values within a brick. To improve the load blancing we intend to extend the brick structure for non-uniform bricks. Using smaller brick sizes at the likely locations of data exchange during load balancing would decrease the transfer costs as well as allow for more accurate adaption of the work load.

8. Acknowledgements

The aneurism data set is provided by Michael Meißner, Viatronix Inc., the Christmas tree by the University of Vienna and Vienna University of Technology. The Visible Male data set is courtesy of the National Library of Medicine.

References

[AR05] ALLARD J., RAFFIN B.: A Shader-Based Parallel Rendering Framework. In *Proceedings of IEEE Visualization Conference '05* (2005), pp. 127–134. 2

[BPT02] BAJAJ C., PARK S., THANE A.: *Parallel Multi-PC Volume Rendering System*. CS & ICES Technical Report, University of Texas at Austin, 2002. 2

[CCF94] 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* (1994), pp. 91–98. 2

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01* (2001), pp. 9–16. 5

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. In *Proceedings of Eurographics '05* (2005), pp. 303–312. 2

[KW03] KRÜGER J., WESTERMANN R.: Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization '03* (2003), pp. 287–292. 2, 3

[Lev90] LEVOY M.: Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9*, 3 (1990), 245–261. 5

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32. 1

[MHE01] MAGALLÓN M., HOPF M., ERTL T.: Parallel volume rendering using PC graphics hardware. In *Pacific Conference on Computer Graphics and Applications* (2001), pp. 384–389. 2

[MPHK93] MA K. L., PAINTER J. S., HANSEN C. D., KROGH M. F.: A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering* (1993), pp. 15–22. 1, 4

[Neu93] NEUMANN U.: Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering* (1993), pp. 97–104. 1, 4

[Sch05] SCHARSACH H.: Advanced GPU Raycasting. In *Proceedings of CESCG '05* (2005), pp. 67–76. 2

[SMW*04] STRENGERT M., MAGALLÒN M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04)* (2004), pp. 41–48. 2, 3

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05* (2005), pp. 187–195. 2

[WGS04] WANG C., GAO J., SHEN H.-W.: Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04)* (2004), pp. 23–30. 2

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization Conference '03* (2003), pp. 333–340. 3

© The Eurographics Association 2006.

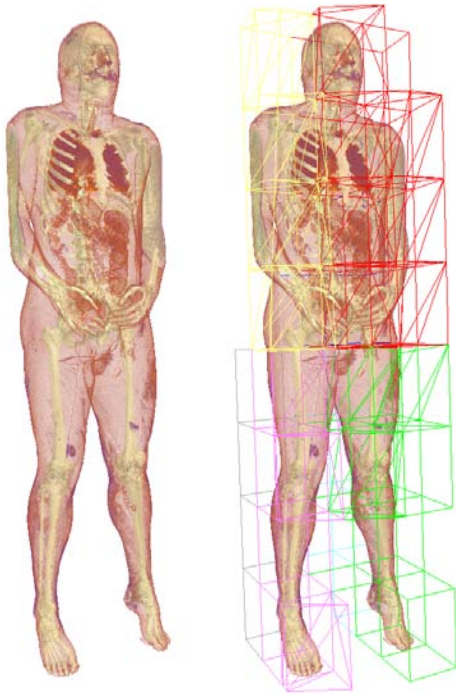

Figure I: *The Visible Male data set rendered using a combined isosurface/DVR shader.*

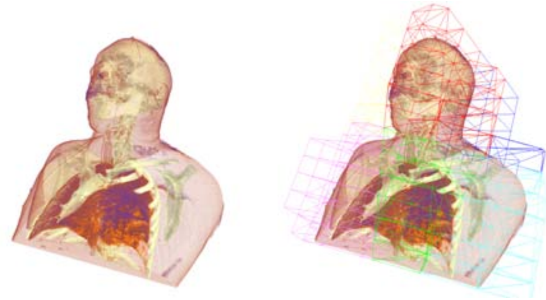

Figure II: *Combined isosurface/DVR image of the upper* 512^3 *part of the Visible Male.*

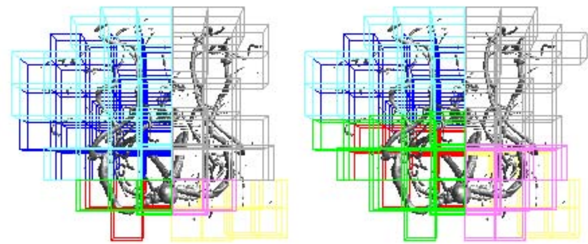

Figure III: *The aneurism data set before (left) and after (right) load balancing. The color coding denotes the assignment to the different cluster computers.*

© The Eurographics Association 2006.