Reducing Artifacts between Adjacent Bricks in Multi-resolution Volume Rendering

Rhadamés Carmona[1], Gabriel Rodríguez[1], and Bernd Fröhlich[2]

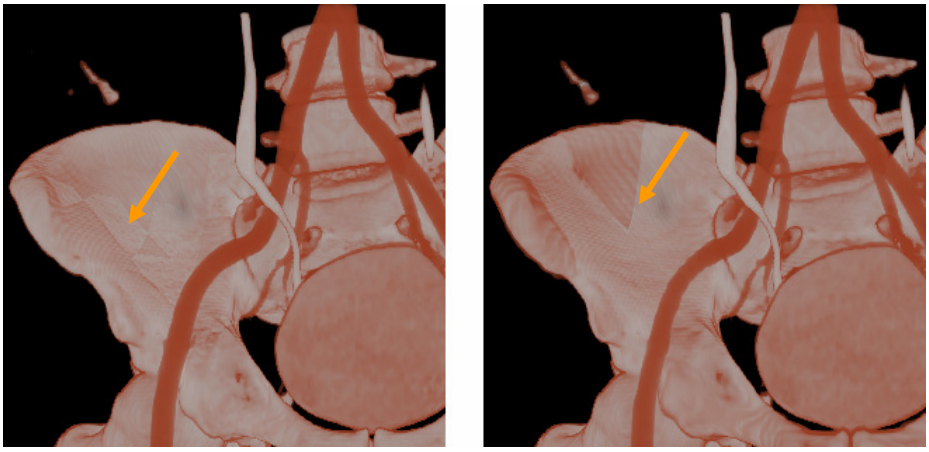
[1] Universidad Central de Venezuela, Centro de Computación Gráfica, 1041-A, Caracas- Venezuela
rhadames.carmona@ciens.ucv.ve, gaborodriguez@gmail.com
[2] Bauhaus-Universität Weimar, Fakultät Medien, 99423 Weimar, Germany
bernd.froehlich@uni-weimar.de

Abstract. Multi-resolution techniques are commonly used to render volumetric datasets exceeding the memory size of the graphics board, or even the main memory. For these techniques the appropriate level of detail for each volume area is chosen according to various criteria including the graphics memory size. While the multi-resolution scheme deals with the memory limitation, distracting rendering artifacts become noticeable between adjacent bricks of different levels of detail. A number of approaches have been presented to reduce these artifacts at brick boundaries, including replicating or interpolating data between adjacent bricks, and inter-block interpolation. However, a visible difference in rendering quality around the boundary remained, which draws the attention of the users to these regions. Our ray casting approach completely removes these artifacts by GPU-based blending of contiguous levels of detail, which considers all the neighbors of a brick and their level of detail.

1 Introduction

During the past years multi-resolution hardware-accelerated volume rendering has been an important research topic in the scientific visualization domain. Due to the progressive improvements of imaging devices such as tomographs and magnetic resonators, the size of volume datasets continuously increases. Such datasets often exceed the available memory of graphics processing units (GPU), and thus multi-resolution techniques need to be employed to guarantee interactive frame rates for GPU-based rendering approaches. Out-of-core techniques are required for even larger datasets exceeding the main memory capabilities of regular desktop computers, which are generated e.g. by scientific projects such as the visible human ® [1] and the time-dependent turbulence simulation of Richtmyer-Meshkov [2].

While multi-resolution approaches in combination with out-of-core techniques deal with the memory limitations, distracting rendering artifacts between adjacent blocks of different level of detail occur (see Fig. 1). The source of these visual artifacts is the interpolation process since coarser data generates different samples during interpolation, which may be mapped to different colors by classification and during integration along the ray. The presence of these kinds of artifacts in the resulting image subconsciously draws the attention of the user to these regions of the volume instead of allowing the user to focus on the actual data.



a. b.

Fig. 1. Angiography dataset with aneurism. Images have been generated by using (a) 30MB of texture memory and (b) 4 MB of texture memory. In both cases, disturbing artifacts are noticeable at levels of detail transitions.

We developed an approach for effectively removing the rendering artifacts related to the quality difference between adjacent bricks of different level of detail. The bricks of the current cut through the octree are interpolated with their representation at the next coarser LOD, such that the resolutions are identical on boundaries between adjacent bricks. This GPU-based interpolation results in an imperceptible transition between adjacent bricks. Our approach requires a restricted octree, where adjacent bricks differ only by one LOD. We consider the resolution of all the neighbors of a brick of the cut during the generation of a volume sample. The interpolation coefficients for a sampling point in the volume can be efficiently generated on the fly from a small pre-computed 3D texture.

Previous research on reducing these multi-resolution rendering artifacts [3], [10], [11] focused on replicating or interpolating boundary voxels between adjacent bricks. While these schemes produce a smooth transition only for the boundary voxels between adjacent bricks, the abrupt change in visual quality around the boundary is still quite noticeable similar to Fig. 1. Our work is inspired by LaMar et al. [12], who achieved imperceptible transitions between LODs for an oblique clipping plane through a multi-resolution volume.

The main contribution of this paper is an effective and efficient approach for removing multi-resolution volume rendering artifacts between adjacent bricks of different level of detail. We integrated our technique into a GPU-based volume ray casting system, which also supports pre-integrated rendering [24], [25]. Our experiments show that the typical memory overhead introduced by our approach is about 10 percent while the increase in computation time is about 20 percent. Our approach extends to CPU-based rendering as well, is easily integrated in any multi-resolution volume rendering system and has the potential to become a standard technique for multi-resolution volume rendering.

2 Related Work

Hardware-accelerated rendering is the current standard for real-time rendering of volume datasets. It was introduced by Akeley [13], who suggested considering the volume dataset as a hardware-supported 3D texture. Various implementations of hardware-accelerated volume rendering have been popular during the past decade, including view-port aligned polygons [14], spherical shells [3], and GPU-based ray casting [15]. These schemes have also been adapted to render large volume datasets [3], [4], [16]. The term "large" is used for volumes, which do not fit into texture memory, and can potentially also exceed the available main memory. To deal with these limitations, the volume can be divided into sub-volumes or bricks, generally of equal size, such that each brick can be rendered independently, and easily swapped with another one [17]. However, the limited bandwidth between the main memory and the GPU still represents a bottleneck for large datasets on desktop computers, limiting the interactivity. Distributed architectures using multiple GPUs have been recently evaluated to alleviate the texture memory and bandwidth limitations [27].

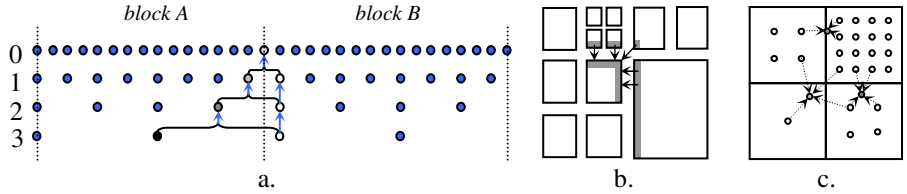


Fig. 2. Consistent interpolation between bricks or blocks of different LODs. (a) The voxels left and right of the boundary between the blocks *A* and *B* are used to replace voxels of that boundary for the next LOD. The right boundary voxel is copied into the right boundary voxel of the next coarser LOD, and the average voxel is copied into the left voxel of the next coarser LOD. Notice that when blocks *A* and *B* with different LODs are selected for rendering, a consistent interpolation between samples is obtained at boundaries. (b) Samples are replaced only in half of the boundary faces. The voxels of the right (*xmax*), up (*ymax*) and back (*zmax*) faces are replaced, taking the boundary data from adjacent bricks. (c) Interblock interpolation. In general, each sample to be reconstructed at brick boundaries is obtained by weighting up to 4 voxels (8 voxels for the 3D case).

Artifacts at brick boundaries have been reduced with various techniques. LaMar et al. [3] share the boundary voxels between adjacent bricks in each LOD. In this case, the artifacts are only removed between adjacent bricks of the same LOD. Weiler et al. [10] obtain a consistent interpolation between contiguous LODs by letting the boundary voxels between blocks interpolate the scalar field of the coarser LOD. Although this idea is initially outlined for transitions of contiguous LODs, it can be applied iteratively to achieve higher order transitions. Fig. 2a illustrates this process for one-dimensional textures. For the 3D case, this process has to consider the six faces of the blocks. Subsequently, Guthe et al. [7] use a similar concept for octrees, but they replicate voxels of only 3 faces of each brick (see Fig. 2b). Finally, Ljung et al. [11] partition the volume into blocks with a local multi-resolution hierarchy, but boundary voxels are not shared between adjacent blocks. During rendering, they perform the

interpolation between blocks of arbitrary resolutions in a direct way (see Fig. 2c), i.e. without replicating voxels or pre-calculating intermediate samples by interpolation.

LaMar et al. [12] presented a multi-resolution technique for interactive texture-based rendering of arbitrarily oriented cutting planes. They achieved smooth transitions on a clipping plane by blending contiguous LODs. We extend this idea to 3D volume rendering considering the volumetric blending of bricks and show how it can be efficiently implemented. Their approach requires also that adjacent bricks along the clipping plane differ in at most one LOD. A similar constraint has been previously introduced for terrain visualization (restricted quad-trees) [18], to guarantee consistent triangulation and progressive meshing during roaming.

3 Multi-resolution Approach

We use a multi-resolution scheme to deal with large volume datasets. During pre-processing, the volume is downsampled into LODs, and partitioned into bricks, to build an octree-based multi-resolution hierarchy [3], [5], [6]. For each frame, a sub-tree of the whole octree is chosen according to a priority function $P(x)$, which may include data-based metrics [9] and image-based metrics [4]. The LOD of each volume area is chosen by a greedy-style subdivision process, considering the priority $P(x)$ [5], [8]. It starts by inserting the root node into a priority queue. Iteratively, the node with highest priority (head node) is removed from the queue, and its children are re-inserted into the queue. This process continues until the size limit is reached, or the head node represents a leaf in the octree hierarchy. We adapt this greedy-style selection algorithm such that the difference between adjacent bricks does not exceed one LOD. Also, the memory cost of the parent bricks has to be considered. Our rendering algorithm implements GPU-based ray casting, with pre-integrated classification.

The multi-resolution approach is presented in the following subsections. We first introduce the core of the approach, which is based on blending of contiguous LODs. Then, the selection algorithm is described, which considers the priority function and the constraint of the levels of detail between adjacent bricks. Finally, we include implementation details of the pre-integrated ray casting system with out-of-core support.

3.1 Blending

Fig. 3a shows a basic one-dimensional example of the blending approach. Consider brick B located at LOD i, and its adjacent brick CD located at the next coarser LOD (level $i-1$). Brick B is gradually blended with its representation at the next coarser LOD (brick AB) such that the coarser representation is reached at the boundary to brick CD. A weight t varying from 0 to 1 along B is required to perform the blending.

For the 3D case, any selected brick x located at the i-th LOD is adjacent to other bricks (26 bricks for the general case), which can be located at level $i-1$, level i or level $i+1$. Weights are assigned to each vertex of the brick; a weight of 0 is assigned if the vertex is only adjacent to bricks located at finer or the same LOD (level $i+1$ or level i), indicating that the resolution in this vertex corresponds to level i. Otherwise, its weight is 1 (the resolution of that vertex corresponds to level $i-1$). Fully transparent adjacent bricks are not considered during the weighting process, since they are

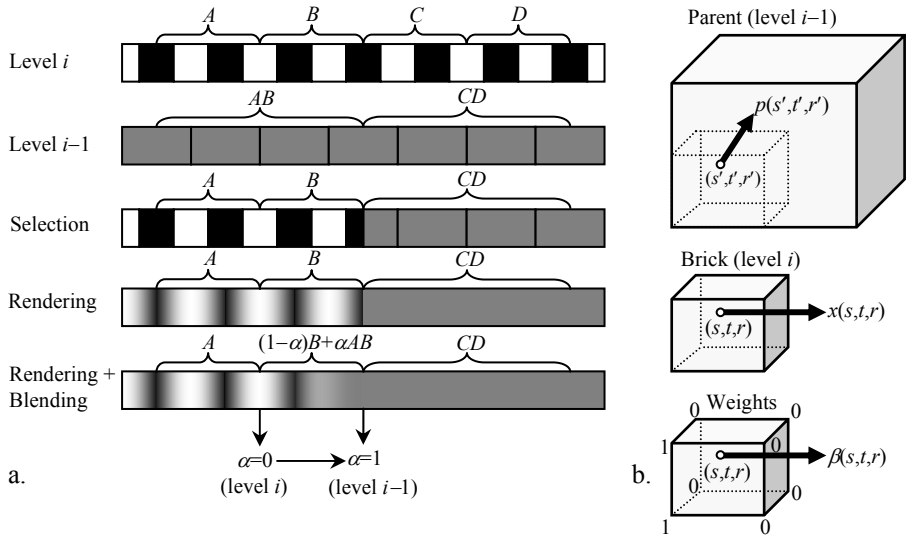


Fig. 3. Interpolation between LODs of adjacent bricks. (a) Levels i and i–1 are two subsequent LODs partitioned into bricks. Adjacent bricks of the same LOD share half a voxel at boundaries. The selection criterion selects bricks A, B and CD for rendering. During the rendering, the voxel data is interpolated to generate a volume sample. The difference between LODs of adjacent bricks generates visible artifacts. These artifacts are removed by means of gradual interpolation from the finer level to its next coarser representation, such that the respective LOD matches at the brick boundary. (b) Blending a brick located at level i with its parent at level i–1. The blending factor β of any position (s,t,r) inside the brick is computed by a tri-linear interpolation of the weights assigned to each brick vertex.

excluded from rendering. If the weight of every vertex is 0, then its parent brick is not required for blending. In any other case, the blending is performed for each volume sample with texture coordinates (s,t,r) in brick x. Thus, the volume sample $x(s,t,r)$ and the corresponding volume sample $p(s',t',r')$ of its parent are linearly interpolated using equation (1):

$$blend(\beta) = (1-\beta)\cdot x + \beta \cdot p\ , \tag{1}$$

where β is computed by tri-linear interpolation of vertex weights (see Fig. 3b). Notice that the blended volume sample $blend(\beta)$ is obtained by quad-linear interpolation since x and p are reconstructed via tri-linear interpolation. Particularly, $blend(\beta)=x$ (level i) if the interpolated weight is β=0, and $blend(\beta)=p$ (level i–1) if β=1. For any other value of β in (0,1), the level of detail of the resulting volume sample is i–β, i.e. an intermediate level of detail between the levels i–1 and i. This also shows that the rendering achieves voxel-based LODs, guaranteeing a smooth transition between adjacent bricks.

For fast tri-linear interpolation of the vertex weights, a single 3D texture of 2x2x2 voxels (corresponding to the weights of brick vertices) can be used for each brick. However, downloading one extra 3D texture into texture memory for each brick to

blend is not appropriate. Similarly to the marching cubes approach [26], this table could be reduced to 15 cases by transforming the texture coordinates (*s*,*t*,*r*) appropriately. However, this would increase the overhead for the fragment program and we opted for using a larger 3D texture instead. This 3D texture of 2x2x512 weights is downloaded only once into texture memory (see Fig. 4). Therefore, a simple index indicating which combination corresponds to the brick being rendered is required for accessing the correct weights for each volume sample.

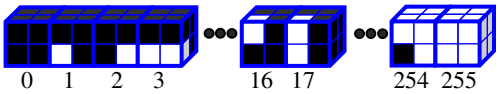


Fig. 4. Look-up table with all possible combinations of vertex weights

3.2 Priority Function

In our implementation, the priority of brick x, $P(x)$, combines two metrics similar to [20]: the distortion $D(x)$ in the transfer function domain, and the importance level $I(x)$, based on the distance from x to the viewpoint and region of interest. We present a short summary of the computation of the priority function. A more detailed explanation of the priority function and the real-time updating is found in [20].

$D(x)$ is the distortion of approximating the source voxels (s_i) by the voxels x_i of x. It can be written as (2):

$$D(x) = \sum_{i=1}^{n_s} D(s_i, x_i), \tag{2}$$

where $D(s_i,x_i)$ is computed in CIELUV color space after applying the transfer function to the voxels s_i and x_i, and n_s is the number of voxels of the source data approximated by x [4]. The importance level $I(x)$ of the brick x is computed considering the distance from brick x to the region of interest (ROI) and to the viewer:

$$I(x) = (1-t)\frac{length(x)}{length(x)+d(x,ROI)} + t\frac{length(x)}{length(x)+d(x,eye)}, \tag{3}$$

where $diag(x)$ is the diagonal length of the brick x in object space, $d(x,eye)$ is the minimum Euclidean distance from the brick x to the eye, and $d(x,ROI)$ is the average between the minimum distance between x and the *ROI* and the minimum distance between x and the *ROI* center. The value of t is used to weigh the distance to the ROI versus the distance to the eye. In our tests, we set t=0.25, giving more priority to the distance to the *ROI*. The distortion level and the importance level are multiplied to define the priority function $P(x)$ as (4):

$$P(x) = D(x) \cdot I(x). \tag{4}$$

3.3 Selecting the Bricks

Before rendering, the set of bricks for representing the volume under the texture memory constraint have to be selected. We use a greedy-style algorithm, which

selects the nodes with highest priority $P(x)$ for splitting. It uses a priority queue PQ to perform this process. Starting by inserting the root node into the queue, the selection process consists of splitting the node of PQ with highest priority [4], [5], [7], i.e. removing the head of PQ and re-inserting its non-fully transparent children into PQ. We use a min-max octree [22] to discard fully transparent bricks. During the refinement process, the following constraints are considered in this approach:

- Splitting a node must not violate the adjacency constraint. This constraint indicates that the difference with respect to the LOD between adjacent bricks selected for rendering must not exceed one level. It suggests building a restricted octree, which can be constructed using an adaptation of the restricted quad-tree algorithm for terrain rendering [18].
- Every node $x \in PQ$ requires blending with its own parent node during rendering, if at least one of its adjacent bricks is coarser than x.
- The number of bricks used for rendering (including parent nodes) is limited by a hardware constraint or user-defined texture size (N bricks).

The adjacency constraint is respected by evaluating the LOD of *adjacent* bricks before splitting a node $x \in PQ$. Each coarser node y adjacent to x needs to be split before splitting the selected node x. Notice that the adjacency constraint has to be evaluated again before splitting any adjacent node, and so on. This suggests using a recursive procedure or an auxiliary queue to perform this task. In addition, for each node x in PQ, we keep the list of its adjacent nodes $A(x)$. When an individual split is performed, the adjacency list of each child is created with the union of its brother nodes and a subset of $A(x)$; also, the adjacency list of each adjacent node is updated, replacing the entry x by the corresponding subset of $children(x)$.

The algorithm keeps track of the number of nodes selected for rendering, *including* the parent bricks. Each node x in PQ has a flag, indicating if such a node requires its parent for blending. One node requires its parent if any of its adjacent bricks is coarser than itself. If a node x is split, the flag of x and the flag of its adjacent bricks need to be re-evaluated.

The refinement process stops if a split operation exceeds the texture memory constraint, or no more refinement is possible.

3.4 Rendering

The selected bricks are rendered in front-to-back order, using GPU-based ray casting with pre-integrated classification. They are composited with the under operator [19]. The pre-integrated table is incrementally generated using the $O(n^2)$ incremental algorithm of Lum et al. [24], which requires about 0.06 seconds for n=256. Each brick is stored in an individual 3D texture object; thus, loading and rendering can be performed in an interleaved fashion, and potentially in parallel [21]. For each brick, the front faces of its bounding box are rasterized [15], interpolating the texture coordinates of the vertices, and the viewing vector. Therefore, each fragment obtained from rasterization contains the viewing ray and the entry point into the brick in texture space.

The ray is sampled with constant step length. For each pair of consecutive samples x_f and x_b, the 2D pre-integration table is fetched at $(s,t)=(x_f,x_b)$ to retrieve the corresponding color integrals $RGB(x_f,x_b)$ and the opacity $\alpha(x_f,x_b)$ in the interval $[x_f,x_b]$. If

the brick requires blending with its parent, the parent is sampled at constant steps as well. The front sample x_f of the brick x is blended with the corresponding front sample p_f of its parent (1), obtaining the blended front sample f. Also, the back samples are blended, obtaining the blended back sample b. Thus, the pre-integration table is only fetched at location (f,b), i.e. there is only one fetch of the pre-integration table per ray-segment. The last ray-segment inside the brick is clipped at the brick boundaries. In this case, the segment length is used to scale the color integrals, and also to correct the opacity [10].

3.5 Caching and Out-of-Core Support

A texture memory buffer containing 3D texture objects (bricks) is created to store the octree cut in texture memory. To exploit frame-to-frame coherence, we keep a list of used bricks in texture memory, and another list of unused bricks. Every time the list of bricks requested for rendering changes between frames, unused bricks are replaced by new bricks and moved to the used list, according to an LRU (replace the least recently used page) scheme [6]. Since the number of new bricks can be eventually large, it may influence the frame rate significantly. A better approach is suggested in [20], which incrementally updates the previous cut through the octree towards the new cut on a frame-by-frame basis, limiting the number of bricks that are exchanged between frames.

For large datasets fitting only partially into main memory, out-of-core techniques have to be considered. In our system, a simple paging on demand is implemented. A main memory cache is used to hold the bricks required for rendering, and also to keep some other bricks that can be used in future frames. The paging process is running in a separate thread to avoid stalling the rendering process [6], [23]. A simply LRU scheme is also used to replace unused bricks by new bricks [7], [23].

The multi-resolution dataset is stored in a single huge file, and the bricks are loaded in groups of at most 8 bricks (which share the same parent), since the disk bandwidth increases by loading contiguous data [20], [23]. In our tests, for brick sizes varying between 16^3 and 64^3 samples, loading blocks containing 8 bricks doubles the disk bandwidth, in comparison to loading independent bricks. While the requested data is not available in main memory, the rendering thread continues rendering the last available data, which keeps the system interactive.

4 Implementation and Results

The system prototype was developed and evaluated under 32-bit Windows XP using Visual C++ 2005 with OpenGL® support. The hardware platform used for the tests is a desktop PC with a 2.4GHz quad core Intel® processor, 2 GB of main memory, an NVidia® Geforce^{tm} 8800 graphics card with 640MB on-board memory, and a SATA II hard disk of 7200 rpm. Medical datasets (See Table 1, and Fig. 5) have been selected for testing: computer tomography of the visible female from the Visible Human project ® [1] (VFCT), angiography with aneurism (Angio), and grayscale-converted photos of the visible female from the Visible Human project ® [1] (VF8b).

Table 1. Test Datasets

Attribute	VFCT	VF8b	ANGIO	Measurement	VFCT	VF8b	ANGIO
Width (v=voxels)	512	2048	512	FPS naive approach	20.01	15.00	22.09
Height (v)	512	1216	512	FPS blended approach	16.7	11.72	17.80
Slices (v)	992	5186	1559	% Blending overhead	16.54%	21.87%	19.42%
Source Size (GB)	0.85	12.03	0.76	(a) Nr. Selected bricks	3865	2761	3255
Brick Size (v)	16^3	32^3	16^3	(b) Nr. Parents used	426	308	426
Bits per voxel	16	8	16	Total bricks = (a)+(b)	4291	3069	3681
Tex. Cache (MB)	35	100	30	% Parent bricks	9.93%	10.04%	11.57%
Ram Cache (MB)	140	400	120	% Blended bricks	65.74%	54.04%	70.26%
Block Size (KB)	64	256	64	(c) LOD ave. naive	5.39	5.67	5.48
Bandwidth (MB)	10	33	10	(d) LOD ave. blended	5.21	5.53	5.30
LODs	0..7	0..8	0..8	LOD difference (c) – (d)	0.17	0.14	0.18

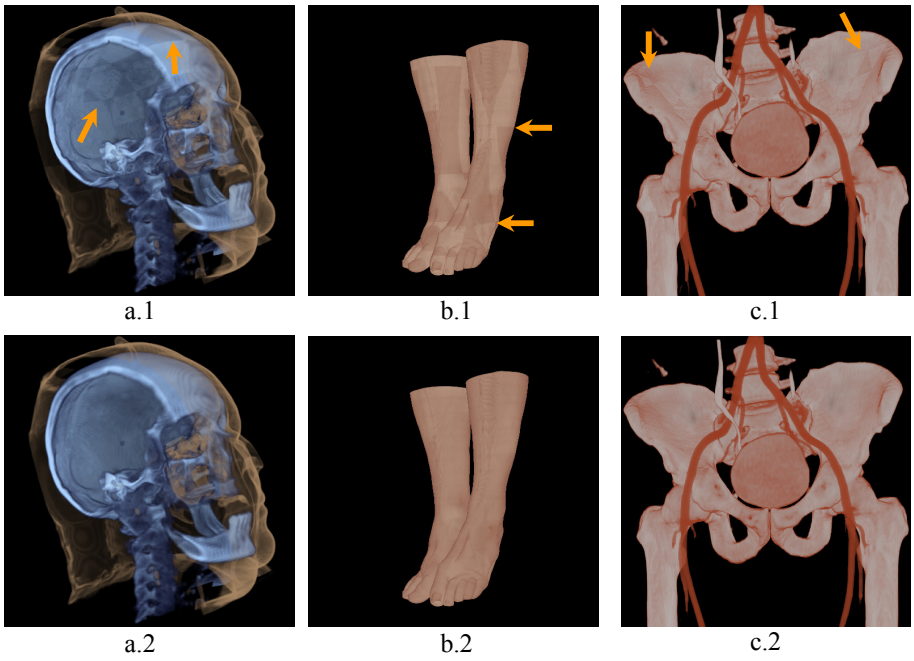


Fig. 5. Removing artifacts of our test datasets. (a) VFCT: Head of the visible female, obtained from CT. (b) VF8b: Feet of the visible female, from full color images converted to grayscale. (c) Angio: Angiography with aneurism. Upper images (*.1) are rendered using the naive approach; bottom images (*.2) are rendered with blending. Each brown arrow points to a visual artifact between adjacent bricks.

For each dataset, a texture memory size is set for caching. A further budget is reserved in main memory to page bricks from disk. We use four times the texture memory size for caching bricks in main memory. During pre-processing, datasets are split into bricks and downsampled to build the multi-resolution hierarchy. Each set of nodes sharing the same parent is grouped into a single block. All blocks are stored in

a single binary file. Datasets of 12 bits per sample are scaled to 16 bits to increase the interpolation accuracy [23].

Results are shown in Table 1 and Fig. 5. The transfer function used for VFCT and VF8b is illustrated in Fig. 6. The blended version reduces the frames rate (FPS) rate by about 20%, although more than 54% of the bricks require blending with their parent during rendering. Notice that only about 10% additional bricks (parent bricks) are required for our blending approach. In theory, up to $1/8^{th}$ of the bricks in the cut may be required for a full octree. However, not every selected brick requires its parent for rendering.

We estimated the average LOD for the generated images, both for the naive approach as well as for the blended approach. The average LOD for the naive approach is calculated as the sum of the LODs of each selected brick, weighted by the volume ratio represented by each brick. Let T be the list of nodes selected for rendering (without parent bricks). The average level of detail is defined by equation (5).

$$LOD(T) = \frac{\sum_{x \in T} Volume(x) \cdot LOD(x)}{\sum_{x \in T} Volume(x)}. \tag{5}$$

The average LOD of the blended approach is also calculated by equation (5). However, the level of detail of brick x ($LOD(x)$) is estimated by averaging the level of detail assigned to each brick vertex. Due to the blending of some bricks with the next coarser LOD, the average LOD of the blended approach is lower. However, it is only about 0.18 levels coarser than the naive approach for our examples.

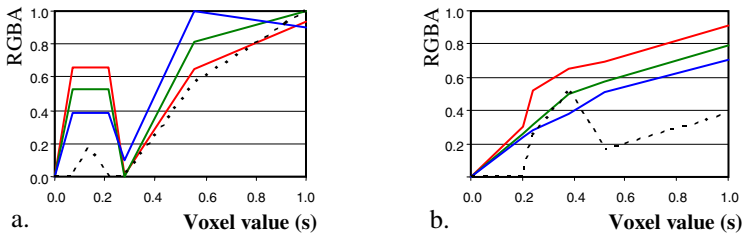


Fig. 6. Normalized transfer functions for the visible female. (a) VFCT, (b) VF8b. R, G and B channels are shown with the corresponding color. Absorption function is stored in the alpha channel A, denoted by a black dotted line.

5 Conclusions and Future Work

We introduced an efficient and effective technique to remove disturbing artifacts between adjacent bricks for direct multi-resolution volume rendering. Our approach blends a brick of the cut with its next coarser representation in such a way that the resolutions match at boundaries with its adjacent bricks. Thus, the LOD is gradually reduced inside a brick instead of simply displaying bricks of different level of detail next to each other or blending a few boundary pixels. Our results show the effectiveness of this technique,

which only increases the rendering time by about 20% while requiring only about 10% of texture memory overhead.

Our approach can be extended to work in a multi-resolution framework, which supports roaming through a volume [6], [20]. In this case the cut is often updated from frame to frame, which may incur popping artifacts if the LOD changes in a certain area of the volume. Our technique can be used to generate an animated transition between the previous cut and the current cut. However, the cuts should be nowhere more different than one LOD and the animated transition becomes a 5D interpolation, since each cut requires already a 4D interpolation to perform the blending. Fortunately, the 3D interpolation part is directly hardware-supported.

References

1. Visible Human Project® (2009), http://www.nlm.nih.gov/research/visible/visible_human.html
2. Mirin, A., Cohen, R., Curtis, B., Dannevik, W., Dimits, A., Duchaineau, M., Eliason, D., Schikore, D., Anderson, S., Porter, D., Woodward, P., Shieh, L., White, S.: Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. In: Proc. ACM/IEEE Supercomputing Conference 1999, vol. 13(18), p. 70 (1999)
3. LaMar, E., Hamman, B., Joy, K.I.: Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In: Proc. IEEE Visualization 1999, pp. 355–362 (1999)
4. Wang, C., García, A., Shen, H.-W.: Interactive Level-of-Detail Selection Using Image-Based Quality Metric for Large Volume Visualization. IEEE Transactions on Visualization and Computer Graphics 13(1), 122–134 (2007)
5. Boada, I., Navazo, I., Scopigno, R.: Multiresolution Volume Visualization with Texture-based Octree. The Visual Computer 17, 185–197 (2001)
6. Plate, J., Tirtasana, M., Carmona, R., Froehlich, B.: Octreemizer: A Hierarchical Approach for interactive Roaming through Very Large Volumes. In: Proc. EUROGRAPHICS/IEEE TCVG Symposium on Visualization 2002, pp. 53–60 (2002)
7. Guthe, S., Wand, M., Gonser, J., Straßer, W.: Interactive Rendering of Large Volume Data Sets. In: Proc. IEEE Visualization 2002, pp. 53–60 (2002)
8. Ljung, P., Lundström, C., Ynnerman, A., Museth, K.: Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets. In: Proc. IEEE Symposium on Volume Visualization and Graphics 2004, pp. 25–32 (2004)
9. Guthe, S., Straßer, W.: Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. Computers & Graphics 28(1), 51–58 (2004)
10. Weiler, M., Westermann, R., Hansen, C., Zimmerman, K., Ertl, T.: Level-Of-Detail Volume Rendering via 3D Textures. In: Proc. IEEE Symposium on Volume Visualization and Graphics 2000, pp. 7–13 (2000)
11. Ljung, P., Lundström, C., Ynnerman, A.: Multiresolution Interblock Interpolation in Direct Volume Rendering. In: Proc. EUROGRAPHICS/ IEEE-VGTC Symposium on Visualization 2006, pp. 256–266 (2006)
12. LaMar, E., Duchaineau, M., Hamann, B., Joy, K.: Multiresolution Techniques for Interactive Texture-based Rendering of Arbitrarily Oriented Cutting Planes. In: Proc. EUROGRAPHICS/IEEE TVCG Symposium on Visualization 2000, pp. 105–114 (2000)
13. Akeley, K.: Reality Engine Graphics. In: Proc. annual conference on Computer graphics and interactive techniques SIGGRAPH 1993, pp. 109–116 (1993)

14. Dachille, F., Kreeger, K., Chen, B., Bitter, I., Kaufman, A.: High-Quality Volume Rendering Using Texture Mapping Hardware. In: Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 1998, pp. 69–76 (1998)
15. Krüger, J., Westermann, R.: Acceleration techniques for GPU-based Volume Rendering. In: Proc. IEEE Visualization 2003, pp. 287–292 (2003)
16. Ljung, P.: Adaptive Sampling in Single Pass, GPU-based Ray Casting of Multiresolution Volumes. In: Proc. EURO-GRAPHICS/IEEE International Workshop on Volume Graphics 2006, pp. 39–46 (2006)
17. Grzeszczuk, R., Henn, C., Yagel, R.: Advanced Geometric Techniques for Ray Casting Volumes. Course Notes No. 4. In: Annual Conference on Computer Graphics - SIGGRAPH 1998 (1998)
18. Pajarola, R., Zürich, E.: Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation. In: Proc. Visualization 1998, pp. 19–26 (1998)
19. Max, N.: Optical Models for Direct Volume Rendering. In: Visualization in Scientific Computing, pp. 35–40. Springer, Heidelberg (1995)
20. Carmona, R., Fröhlich, B.: A Split-and-Collapse Algorithm for Interactive Multi-Resolution Volume Rendering. Elsevier Computer & Graphics (Submitted for publication, 2009)
21. Rezk-Salama, C.: Volume Rendering Techniques for General Purpose Graphics Hardware. Thesis dissertation, Erlangen-Nürnberg University, Germany (2001)
22. Lacroute, P.G.: Fast Volume Rendering Using Shear-Warp Factorization of the Viewing Transformation. Technical Report CSL-TR-95-678, Stanford University (1995)
23. Ljung, P., Winskog, C., Persson, A., Lundström, K., Ynnerman, A.: Full Body Virtual Autopsies using a State-of-the-art Volume Rendering Pipeline. IEEE Transactions on Visualization and Computer Graphics 12(5), 869–876 (2006)
24. Lum, E., Wilson, B., Ma, K.L.: High-Quality Lighting and Efficient Pre-Integration for Volume Rendering. In: Proc. EUROGRAPHICS/IEEE TCVG Symposium on Visualization 2004, pp. 25–34 (2004)
25. Engel, K., Kraus, M., Ertl, T.: High Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In: Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware 2001, pp. 9–16 (2001)
26. Lorensen, W., Cline, H.: Marching Cubes: A high resolution 3D surface construction algorithm. Computer Graphics 21(4), 320–327 (1987)
27. Monoley, B., Weiskopf, D., Möller, T., Strengert, M.: Scalable Sort-First Parallel Direct Volume Rendering with Dynamic Load Balancing. In: Eurographics Symposium on Parallel Graphics and Visualization 2007, pp. 45–52 (2007)