IEEE/ EG Symposium on Volume and Point-Based Graphics (2008)
H.- C. Hege, D. Laidlaw, R. Pajarola, O. Staadt (Editors)

Smooth Mixed-Resolution GPU Volume Rendering

Johanna Beyer[1] Markus Hadwiger[1] Torsten Möller[2] Laura Fritz[1]

[1] VRVis Research Center, Vienna, Austria
[2] School of Computing Science, Simon Fraser University, Canada

Abstract

We propose a mixed-resolution volume ray-casting approach that enables more flexibility in the choice of downsampling positions and filter kernels, allows freely mixing volume bricks of different resolutions during rendering, and does not require modifying the original sample values. A C^0-continuous function is obtained everywhere with hardware-native filtering at full speed by simply warping texture coordinates of samples in transition regions. Additionally, we propose a simple but powerful, flat texture packing scheme that supports mixing different resolution levels in a single 3D volume cache texture with a very simple and fast address translation scheme. Although this packing constrains full scalability, it enables mixing different resolution levels in GPU-based ray-casting with only a single rendering pass. We demonstrate our approach on large real-world data, obtaining a continuous scalar function and shading at brick boundaries, using single-pass ray-casting at real-time frame rates.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms I.3.6 [Computer Graphics]: Graphics data structures and data types

1. Introduction

The sizes of volumetric data sets, as for example generated in medical imaging, industrial scanners, or scientific simulations, are increasing at a very rapid rate. Although volume rendering approaches exploiting graphics processing units (GPUs) achieve real-time frame rates, GPU texture memory is still limited and increasing more slowly than data sizes. Therefore, many GPU volume rendering schemes incorporate multi-resolution volume bricking or data compression techniques to cope with large volumes. Multi-resolution schemes circumvent the memory constraints of GPUs by downsampling the volume or parts of it to a lower resolution. To reduce visual artifacts in the final image, level-of-detail (LOD) selection techniques can be employed to steer the selection of areas for downsampling.

Most multi-resolution schemes for GPU-based volume rendering restrict the sampling positions of the downsampled grid to a subset of the original sample positions. This, however, restricts the choice of downsampling filters and thus the attainable quality or smoothness of lower resolutions. A major reason for this restriction is that it simplifies the generation of a continuous function when different resolution levels are mixed. Most approaches use only nearest-neighbor downsampling, and moreover require higher resolution samples to be overwritten with values interpolated from lower resolution levels [WWH*00]. This implies that even in the highest resolution level many samples are not identical to the original volume and thus inaccurate.

In this paper we propose a GPU-based bricked mixed-resolution volume rendering scheme that is not restricted to downsampling at the original grid positions and does not require modifying the original sample values. Our approach can be used to render volumes larger than GPU memory with ray-casting in a single rendering pass, mixing different levels of resolution with continuous (C^0) transitions between resolution levels. It operates on a per-sample basis and solely modifies the coordinates of texture fetches in a thin transition region between bricks of different resolutions. Thus, it is fast and requires only minor modifications to existing ray-casters, and could also be applied to renderers using texture slicing. We place the sample positions of a downsampled level half-way between samples of the higher-resolution level above it. This is the natural choice for downsampling filters that weight an even number of samples in order to compute a lower-resolution sample. In this work, we employ a $2 \times 2 \times 2$ averaging filter, but our sampling scheme facilitates higher-order filters as well, e.g., cubic splines.

The major advantages of our approach are that: (1) bricks of different resolutions are stored in a single 3D cache texture and can be freely mixed during rendering; (2) address translation between volume space and the cache texture is extremely simple; (3) the transitions between bricks of different resolution levels are C^0-continuous, which is achieved by simply modifying the texture coordinates of chosen samples in the fragment shader; thus (4) actual interpolation can use the hardware-native tri-linear filtering at full speed; and (5) downsampling positions do not need to be aligned with the original sample positions, which allows for a wider range of filter kernels and thus more flexibility.

2. Related Work

Multi-resolution approaches for volume rendering try to circumvent memory restrictions of current hardware usually by breaking down a single large volume texture into several smaller ones (i.e, bricks). For low-resolution representations either the number of bricks are reduced (as in hierarchical bricking schemes) or the texture size of the bricks is reduced (as in flat bricking schemes). The breaking down of a larger texture into several smaller ones is called *bricking* and usually requires duplication of texels at brick boundaries.

LaMar et al. [LHJ99] were one of the first to introduce a hierarchical (octree) bricking scheme for hardware-assisted volume rendering. They propose a selection filter for downsampling bricks depending on their distance from the viewpoint and the view frustum, however they do not account for continuous transitions between different levels of detail during rendering. Weiler et al. [WWH*00] ensure continuous level transitions in their octree-based multi-resolution scheme by adapting the brick borders of the finer resolutions, throughout all hierarchy levels. They are, however, restricted to downsampling on the original grid points. Other hierarchical approaches including LOD selection algorithms were proposed by Boada et al. [BNS01] and Guthe et al. [GS04], who use a hierarchical wavelet representation and screen-space error estimation for LOD selection. Entezari et al. [EMBM06] use Cartesian, FCC and BCC lattices for downsampling the data using different sampling densities. Their approach, however, does not support mixing of different resolution levels. A method for iso-surface reconstruction of multi-resolution volume data was proposed by Westermann et al. [WKE99]. They create a hierarchical octree using averaging and focus on fixing cracks in the surface at transitions between different resolution levels.

A flat bricking scheme for multi-resolution data with continuous resolution transitions was proposed by Ljung et al. [LLY06]. Their approach does not need sample replication at brick boundaries, as they perform interbrick interpolation directly during rendering. This, however, requires complex fragment shaders to manually perform the correct interpolation. LOD selection is based on the transfer function [LLYM04].

Most multi-resolution approaches render each brick individually, storing them in different textures. Our work, however, is based on a scheme similar to adaptive texture maps introduced by Kraus et al. [KE02], where data bricks of different resolution are packed into a single texture. An additional index texture is used for address translation. In contrast to [KE02], however, we maintain our packed data and index texture dynamically.

Our volume visualization framework builds on previous research in the area of hardware assisted volume rendering using commodity GPUs. While first approaches were based on texture slicing [WE98, RSEB*00], GPU ray-casting is now a viable and very powerful alternative [KW03].

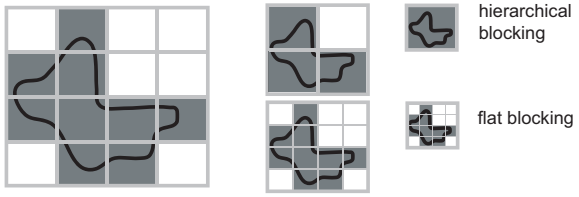

Figure 1: *Hierarchical bricking (top row) vs. flat bricking (bottom row). Culled bricks are marked in white.*

3. Mixed-Resolution Volume Rendering

Most multi-resolution volume rendering methods are based on hierarchical bricking schemes where the brick size in voxels is kept constant from level to level, and the spatial extent of bricks increases from high to low resolution until a single brick covers the entire volume (Figure 1, top row). Conversely, flat bricking schemes (Figure 1, bottom row) keep the spatial extent of bricks constant and successively decrease the brick size in voxels. A major advantage of flat bricking schemes is that the culling rate is much higher, illustrated by the number of white bricks in Figure 1, because the granularity of culling stays constant irrespective of actual brick resolutions. This not only reduces the required texture memory, as more bricks can be culled, but also allows for a much more fine-grained LOD or shader selection per brick [LKP06]. However, flat multi-resolution techniques have a bigger memory overhead when samples are replicated at brick boundaries, because for decreasing brick sizes the overhead of duplicated voxels increases. This overhead can be removed by avoiding sample duplication, but only at the cost of highly increased run-time filtering complexity and cost [LLY06]. We employ flat multi-resolution bricking with sample duplication, but reduce the run-time overhead significantly by using hardware filtering and only warping the texture coordinates of samples where necessary.

3.1. Volume Subdivision for Texture Packing

The original volume is subdivided into equally-sized bricks of size n^3 in a pre-process, where n is a power of two, e.g., $n = 32$. During this subdivision, the minimum and maximum value in each brick are stored for culling later at run time, and lower-resolution versions of each brick are constructed. For the latter we compute the value of the new sample at the center of eight surrounding higher-resolution samples as their average, but higher-order filters could also be used. We limit the number of resolution levels to minimize the overhead of duplicated boundary voxels, and also to allow tight packing of low-resolution bricks in the storage space reserved for high-resolution bricks (Section 3.2). By default we use only two resolution levels, e.g., 32^3 bricks with a downsampled resolution of 16^3. For fast texture filtering during rendering, voxels at brick boundaries are duplicated. In principle, duplication at one side suffices for this purpose [WWH*00], e.g., storing $(32+1)^3$ bricks. However,

in the high-resolution level we duplicate at both sides, because the space for a single $(32+2)^3$ brick provides storage for eight $(16+1)^3$ bricks. Coincidentally, this usually does not impose additional memory overhead. The brick cache texture (Section 3.2) always has power-of-two dimensions for performance reasons, and a cache of size 512^3, for example, can hold the same number of 34^3 and 33^3 bricks.

Although this approach is not fully scalable, it is very simple and a good trade-off that is not as restrictive as it might seem. Because culling is very efficient in a flat scheme, fewer bricks need to be resident in GPU memory. Even without culling, if the size of the brick cache texture is 512x512x1024 (256 mega voxels), for example, and two resolution levels are used (brick storage size 34^3), 15x15x30 bricks fit into the cache. This yields a possible data set size of about 1.7 giga voxels, e.g., 960x960x1920, if all bricks actually need to fit into the cache. Due to culling, the real data set size can typically be much larger. Additionally, for very large data three levels could be used. For example, increasing the allocated space for each brick from $(32+2)^3$ to $(32+4)^3$, both 16^3 and 8^3 bricks can be packed tightly, including boundary duplication for filtering. Using three levels with storage for $(32+4)^3$ bricks, 14x14x28 bricks would fit into the cache, yielding a data set size of 10.7 giga voxels, e.g., 1792x1792x3584, and more when bricks are culled.

3.2. Mixed-Resolution Texture Packing

For rendering, a list of active bricks is determined via culling, using, e.g., the transfer function or iso value, and clipping plane positions to determine non-transparent bricks that need to be resident in GPU memory. The goal is to pack all active bricks into a single 3D brick cache texture (Figure 2, right). In the beginning, all cache space is allocated for high-resolution bricks. If the number of active bricks after culling exceeds the allocated number, individual bricks are chosen to be represented at lower resolution. In this case, the effective number of bricks in the cache is increased by successively mapping high-resolution bricks in the cache to eight low-resolution bricks each, until the required overall number of bricks is available. This is possible because the storage allocation for bricks has been chosen in such a way that exactly eight low-resolution bricks fit into the storage space of a single high-resolution brick, including duplication of boundary voxels, as described in the previous section.

After the list of active bricks along with the corresponding resolutions has been computed, the layout of the cache texture and mapping of brick storage space in the cache to actual volume bricks can be updated accordingly, which results in an essentially arbitrary mixture of resolution levels in the cache. The actual brick data are then downloaded into their corresponding locations using, e.g., `glTexSubImage3D()`. During rendering, a small 3D layout texture is used for address translation between "virtual" volume space and "physical" cache texture coordinates (Figure 2, top left), which is described in the next section.

3.3. Address Translation

A major advantage of our texture packing scheme is that address translation can be done in an identical manner irrespective of whether different resolution levels are mixed. Each brick in virtual volume space always has constant spatial extent and maps to exactly one brick in physical cache space. "Virtual" addresses in volume space, in $[0,1]$, corresponding to the volume's bounding box, are translated to "physical" texture coordinates in the brick cache texture, also in $[0,1]$, corresponding to the full cache texture size, via a lookup in a small 3D layout texture with one texel per brick in the volume. This layout texture encodes (x,y,z) address translation information in the RGB color channels, and a multi-resolution scale value in the A channel, respectively. A volume space coordinate $\mathbf{x}_{x,y,z} \in [0,1]^3$ is translated to cache texture coordinates $\mathbf{x}'_{x,y,z} \in [0,1]^3$ in the fragment shader as:

$$\mathbf{x}'_{x,y,z} = \mathbf{x}_{x,y,z} \cdot \mathbf{bscale}_{x,y,z} \cdot \mathbf{t}_w + \mathbf{t}_{x,y,z}, \tag{1}$$

where $\mathbf{t}_{x,y,z,w}$ is the RGBA-tuple from the layout texture corresponding to volume coordinate $\mathbf{x}_{x,y,z}$, and **bscale** is a constant fragment shader parameter containing a global scale factor for matching the different coordinate spaces of the volume and the cache. When filling the layout texture, the former is computed as:

$$\mathbf{t}_{x,y,z} = \left(\mathbf{b}'_{x,y,z} \cdot \mathbf{bres}'_{x,y,z} - \mathbf{o}_{x,y,z}\right) / \mathbf{csize}_{x,y,z} \tag{2}$$

$$\mathbf{t}_w = 1.0, \tag{3}$$

for a high-resolution brick, where $\mathbf{b}'$ is the position of the brick in the cache, $\mathbf{bres}'$ is the storage resolution of the brick, e.g., 34^3, and **csize** is the cache texture size in texels to produce texture coordinates in the $[0,1]$ range. For a low-resolution brick, this is computed with $\mathbf{t}_w = 0.5$. The offset $\mathbf{o}_{x,y,z}$ is computed as:

$$\mathbf{o}_{x,y,z} = \mathbf{b}_{x,y,z} \cdot \mathbf{bres}_{x,y,z} \cdot \mathbf{t}_w - \mathbf{t}_w, \tag{4}$$

where $\mathbf{b}$ is the position of the brick in the volume, and **bres** is the brick resolution in the volume, e.g., 32^3. The global scale factor **bscale** is computed as:

$$\mathbf{bscale}_{x,y,z} = \mathbf{vsize}_{x,y,z} / \mathbf{csize}_{x,y,z}, \tag{5}$$

where **vsize** is the size of the volume in voxels.

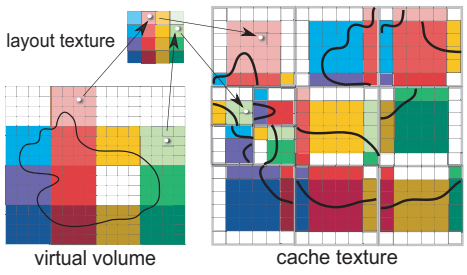

Figure 2: *Mixed-resolution texture packing and address translation from virtual volume space to physical cache texture space via the layout texture. Resolution levels are mixed by packing low-res bricks tightly into high-res bricks.*

4. Smooth Mixed-Resolution Interpolation

The most fundamental operation in direct volume rendering using ray-casting or texture slicing is taking individual samples along viewing rays into the volume. This (re-)sampling requires interpolating between the discrete samples (voxels) at the grid positions. In order to avoid discontinuities, this interpolation must reconstruct a continuous function. When only a single resolution level is used, piecewise tri-linear interpolation yields a continuous function, which is extremely fast on current GPUs, where it can be performed automatically when a 3D texture is sampled in the fragment shader.

However, when multiple resolution levels, i.e., bricks of different resolutions, are mixed, obtaining a continuous function usually requires matching sample positions and modification of original sample values [WWH*00]. Figure 3 shows the sample positions we are using in high-resolution (yellow) and low-resolution (blue) bricks, respectively. In order to allow a sample offset between resolution levels and avoid modifying original samples, we employ the following approach for (re-)sampling in the fragment shader:

- Sampling within bricks (at least 0.5 voxels away from the boundary) is performed as usual, with hardware-native tri-linear interpolation at the sample's texture coordinates.
- Sampling at the boundary between bricks of different resolution warps the texture coordinates of samples within 0.5 voxels from the brick's boundary in the brick of higher resolution. Coordinates are warped according to special interpolation primitives described in detail below.

Note that simply warping texture coordinates implies that the actual interpolation is still carried out by the hardware at full speed. Also, everything is performed on a per-sample basis, i.e., no explicit vertices, vertex attributes, or actual interpolation primitives are used. The primitives that ensure a continuous function in transition regions are only implicit, and solely determine how texture coordinates must be warped.

4.1. Smooth Transition Interpolation

Figure 3 illustrates the different cases of interpolation between bricks of different resolution in 2D. The extension to 3D is conceptually straight-forward and described after the 2D case below. The actual interpolation functions that have to be used in order to obtain a C^0-continuous scalar function depends on the configuration/adjacency of low-res and high-res bricks, all of which are depicted in Figure 3, apart from symmetry. Samples in low-res bricks are shown as black/colored crosses in blue bricks in Figure 3, and samples in high-res samples as white/colored dots in yellow bricks.

When two bricks of different resolution levels are adjacent to one another, the transition region that requires warping of texture coordinates is a band of 0.5 voxels inside the brick of higher resolution. A smooth function is guaranteed by modifying the texture coordinates in this band in such a way that the hardware-native tri-linear interpolation actually carries out a smooth warping between the two sample grids.

Consider the transition region shown in Figure 3 (2A). We have to obtain a smooth interpolation between the high-res samples inside the high-res brick above the brick boundary (colored dots above the colored crosses), and the low-res samples on the brick boundary (colored crosses). We apply two different, smooth interpolation functions: (1) trapezoids between two low-res and two high-res samples; and (2) triangles between one low-res sample and two high-res samples. Trapezoids can be interpolated using bi-linear interpolation, and triangles with linear (barycentric) interpolation. However, for efficiency both must be mapped to hardware bi-linear interpolation inside a square of four samples. Using bi-linear texture fetches in the transition region with the warping of texture coordinates described below, the result of bi-linear texture interpolation is the same as interpolating within these trapezoids and triangles.

Figure 4 (top) depicts this mapping for a single trapezoid, which can be used for any trapezoid in Figure 3 by using the appropriate coordinate offsets. In order to obtain the desired interpolation function with hardware-native bi-linear interpolation, coordinates within the trapezoid must be mapped to a square in such a way that the same interpolation function results. Inside a trapezoid with the coordinate extents

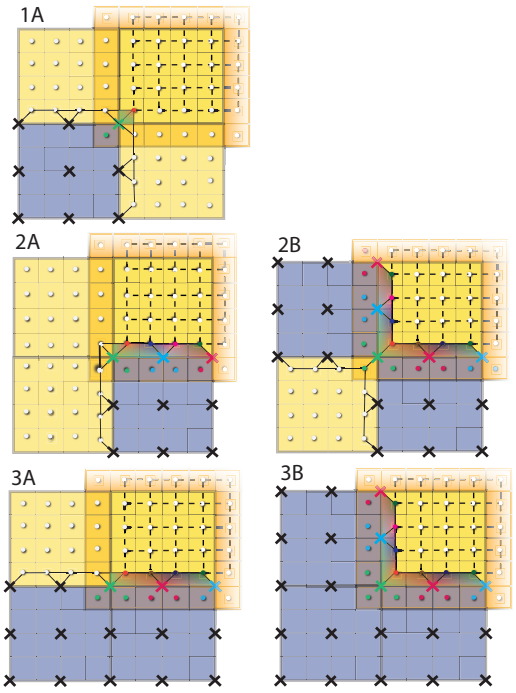

Figure 3: *Basic configurations of mixed-resolution interpolation in 2D transition regions. High-resolution bricks are shown in yellow with white dots at the sample positions, low-resolution bricks in blue with crosses at the sample positions. The figure focuses on the top-right high-resolution brick (shown with its border of duplicated voxels in orange), and the interpolation functions in its* 0.5 *texel border.*

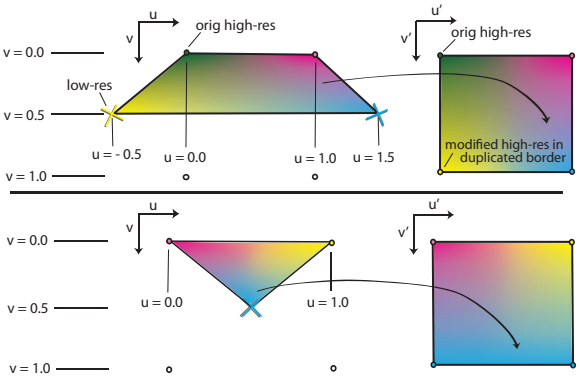

Figure 4: *Mapping texture coordinates within a trapezoid (top) and triangle (bottom), respectively, to a square, such that hardware bi-linear interpolation in the latter yields the same result as (bi-)linear interpolation in the former.*

given in Figure 4 (top left), the following mapping can be used:

$$u' = (u+v)/(1+2v), \tag{6}$$
$$v' = 2v. \tag{7}$$

The two top samples of the square in Figure 4 (top right) are original, unmodified high-res samples. In contrast, the two bottom samples of that square must be modified. However, these samples are duplicated voxels outside the high-res brick, i.e., their modification does not change original sample values. This is illustrated by the colored dots within the orange duplication border in Figure 3. These samples must be set to the values of the low-res grid at the bottom of the trapezoid in Figure 4 (top left). This modification only has to take place whenever the brick cache changes, i.e., is performed entirely independent from the volume rendering fragment shader. The details are explained in Section 4.3.

Figure 4 (bottom) depicts the analogous mapping for a triangle, which can be used for any triangle in Figure 3 by using the appropriate coordinate offsets. Inside a triangle with the coordinate extents given in Figure 4 (bottom left), the following mapping can be used to map a triangle of height 0.5 to a triangle of height 1.0 embedded in a square such that bi-linear interpolation within that square again yields the desired interpolation function:

$$u' = \begin{cases} (u-v)/(1-2v) & v \neq 0.5 \\ u & v = 0.5 \end{cases} \tag{8}$$
$$v' = 2v. \tag{9}$$

Again, the two top samples of the square in Figure 4 (bottom right) are original, unmodified high-res samples, and the two bottom samples of that square result from modification of the voxels in the duplicated border. They are both set to the value at the apex of the triangle in Figure 4 (bottom left).

In order to apply the two mappings above to any trapezoid or triangle in Figure 3, we start with texture coordinates $(\bar{u},\bar{v}) \in [0,1]$, where $[0,1]$ maps to an entire brick. We then shift the coordinates toward the center of the voxels in the high-resolution brick and map $[0,1]$ to the size of a single voxel, instead of the entire brick, to obtain the local coordinates (u,v):

$$u = fract(\bar{u} \cdot b_w - 0.5), \tag{10}$$
$$v = fract(\bar{v} \cdot b_h - 0.5), \tag{11}$$

where b_w and b_h are the width and height of the brick in voxels, respectively.

Extension to 3D

Fortunately, the 3D case is almost a direct extension of the 2D case. In 3D, three different primitives must be used for interpolation, which are shown on the left-hand side of Figure 5 (as top, front and side view). They fit together as illustrated on the right-hand side of Figure 5 (as 3D and top-down view). These primitives and the interpolation within them can be computed from projections into 2D: The first primitive (Figure 5a) is a truncated pyramid that can be constructed from two trapezoid projections, computing the interpolation using Equations 6 and 7. The second primitive (Figure 5b) is a pyramid that can be constructed from two triangle projections, computing the interpolation using Equations 8 and 9. The third primitive (Figure 5c) can be constructed from one trapezoidal projection and one triangular projection, computing the interpolation using Equations 6, 7, 8, and 9. Therefore, checking the current sample's position in the 2D projections (front and side views in Fig-

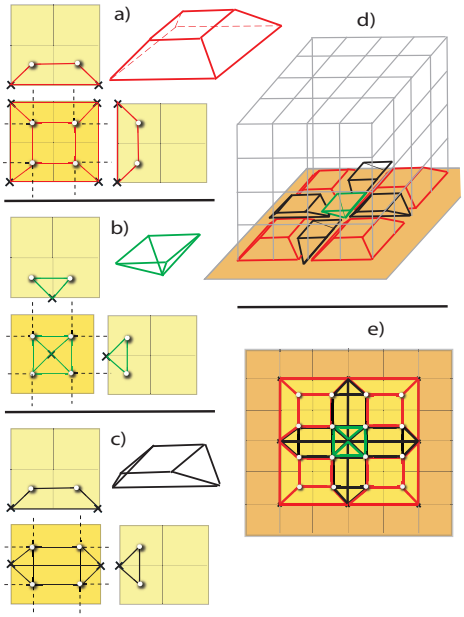

Figure 5: *Smooth mixed-resolution interpolation in 3D. In contrast to the 2D case, in 3D three different primitives have to be used (a, b, c). However, all fragment shader computations can be performed in the 2D front and side projections shown. In 3D, the primitives fit together as shown in d, e.*

ure 5a,b,c), allows to unambiguously identify the primitive the current sample point belongs to.

Primitive type precedence

As can be seen in Figure 3 for the 2D case, special care has to be taken at the corners of bricks (i.e., corners and edges in 3D). Depending on the resolution level of the surrounding bricks different primitives have to be used for correct interpolation. Case 2A in Figure 3, for example, uses a triangle at the leftmost part of the lower border, because the adjacent brick on the left is in high resolution. In case 2B, however, the adjacent brick on the left is in low resolution, which results in using a trapezoid at the leftmost part of the lower border of the high-res brick. In 3D, there are even more configurations how primitives can be combined in the corner of a brick, as shown in Figure 6a,b,c. In this figure, we focus on the lower left corner in the back of the brick, and assume that there is a low-resolution brick directly below the displayed brick. The figure shows the possible configuration of the different primitives to correctly interpolate samples at the lower left corner, in the back. Figure 6a depicts the case where all three surrounding faces of the corner are adjacent to low-res bricks, in Figure 6b two faces are adjacent to low-res bricks, and in Figure 6c only the bottom face is adjacent to a low-res brick. If a high-res brick is only adjacent to a low-res brick at an edge, there would be no face primitive (truncated pyramid) needed, but an additional edge primitive instead. If the low-res brick is only adjacent at the high-res brick's corner, additional corner primitives would be necessary. Consequently, if present, a face primitive always overrides an edge primitive, which in turn overrides a corner primitive.

4.2. Volume Rendering Fragment Shader Modification

As described above, for smooth interpolation sampling in the fragment shader must be modified in order to warp the texture coordinates of samples in transition regions between bricks of differing resolution. This is done as follows:

1. Determine whether the sample position is (1) in a high-res brick; and (2) in from one to three of its six 0.5 texel borders of faces adjacent to a low-res brick.

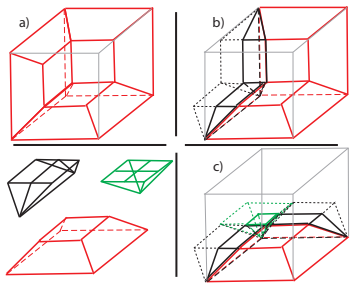

Figure 6: *Different configurations of face, edge and corner primitives, depending on the adjacency configuration of high-res and low-res bricks. Accordingly, the lower left back corner is composed of different primitive types (a, b, c).*

2. If it is, determine the primitive type the sample position projects to via the two 2D projections (Figure 5) orthogonal to each relevant border. For example, for a face orthogonal to the z axis, the (x,z) and the (y,z) projections.
3. If the sample is contained in more than one border, e.g., at an edge, determine the primitive type that must be used according to the primitive precedence described above.
4. Warp texture coordinates in the two 2D projections according to Equations 6–9, and composite the results for the final 3D coordinate, e.g., (x,z) and (y,z) to (x,y,z).

In order to distinguish all possible configurations of adjacency of high-res and low-res bricks efficiently (see Figure 3 for the 2D cases), we create a bit state of the 26-neighborhood of each high-res brick whenever the cache layout changes. A bit is set when a low-res brick is adjacent to a face, an edge, or a corner, respectively. For simplicity, this bit state is supplied to the fragment shader as an additional small 3D lookup texture with one texel per brick, similar to the 3D layout texture. This texture contains a 32-bit integer value for each brick, encoding the 26 neighborhood bits.

4.3. Brick Cache Fixup

In addition to adapting texture coordinates for sampling in the fragment shader, selected sample values must be modified in the brick cache in order to compute the smooth interpolation functions described in Section 4.1 between the correct source sample values. That is, whenever high-resolution bricks are adjacent to low-resolution bricks in volume space, the voxels in the duplicated border of high-resolution bricks might need to be modified in order to perform the correct interpolation. However, this modification solely depends on the layout and resolutions of bricks in the cache, and thus only needs be performed whenever the cache changes, e.g., due to a transfer function change. It is also completely independent from the volume rendering fragment shader and therefore does not influence rendering performance.

Depending on the location of the low-resolution neighbor, we either have to adjust the face, edge, or corner of the adjacent high-resolution brick. Figure 7 shows the 2D case where the left border of the high-resolution brick needs to be modified because it is adjacent to a low-resolution brick in volume space. Therefore, the duplicated border voxels of the high-resolution brick have to be set to the same value as the nearest low-resolution sample of the adjacent brick (shown by the replicated crosses in the high-resolution brick).

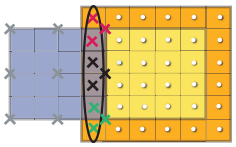

Figure 7: *Fixup of the high-resolution brick's duplicated border to the nearest sample of its low-resolution neighbor in volume space, for smooth hardware-native interpolation.*

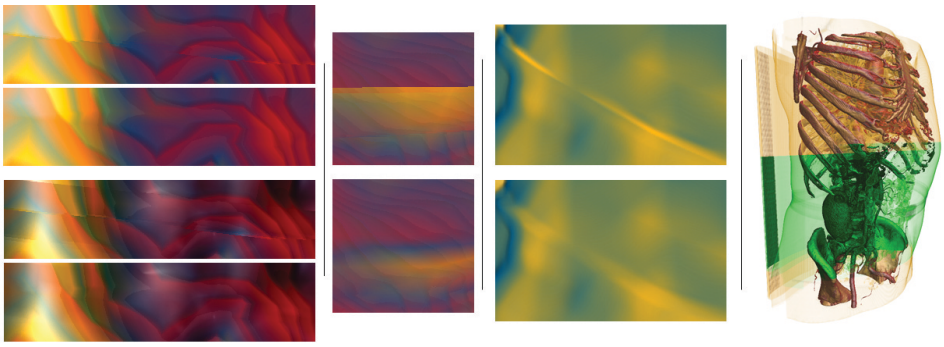

Figure 8: *Transitions between different resolution levels in an abdomen data set (512x512x1112). The upper images show the original discontinuous transitions, whereas the respective lower images show the smooth transitions obtained by our method. Far right: Overview of whole data set with high-resolution bricks colored in green.*

To efficiently implement this fixup step on the GPU, an additional reverse layout texture for address translation between the brick cache and virtual volume space is necessary. Using this reverse address translation, we can look up the volume coordinates for a given brick in the fragment shader that does the fixup. Note that this is not the rendering fragment shader, but an additional shader that is only invoked once whenever the cache changes.

Whenever the brick cache is updated, all high-res bricks that are adjacent to a low-res brick are marked. In order to modify sample values in the duplicated borders where necessary, we rasterize all marked bricks slice-by-slice and perform the fixup on each slice separately, either copying a 2D slice buffer back into the 3D cache texture or directly rendering into its slices. In the shader we check for each fragment if it is located on the rasterized brick's boundary. For all boundary voxels, a reverse lookup is performed to fetch the coordinates in virtual volume space. Translating those volume coordinates back to the brick cache automatically fetches the correct neighbor of the rasterized brick. Now we only have to check if this neighbor is a low-resolution brick. If this is the case, we overwrite the current sample's value with the sample from the low-resolution brick (using nearest neighbor interpolation for the texture fetch in the low-resolution brick). This simple scheme works well for fragments on the brick's border that are positioned on the brick's face (i.e., there is only one direct neighbor). For voxels on the edges and corners we have to check all adjacent bricks and perform the texture fetch on the first low-resolution brick that we encounter. To fetch all adjacent bricks, the current sample is first transformed to virtual volume coordinates and then translated by one voxel in the virtual volume space, depending on the adjacent brick we want to fetch. Performing a lookup from volume space back to the cache brick yields again the correct adjacent brick.

5. Results

We have tested our mixed-resolution volume rendering approach on large real world data. Figure 8 shows highly magnified views (unshaded and shaded DVR) of a medical abdomen data set of size 512x512x1112 (16-bit voxels). Artifacts at brick boundaries of different resolution are clearly visible using standard volume rendering (Fig. 8, upper images). Using our approach, however, a smooth and continuous function can be obtained (Fig. 8, lower images).

Figure 9 shows an iso-surface rendering (first-hit raycasting) of a high-resolution industrial CT scan of a metal ring (1518x1518x232, 16-bit voxels). The magnified views show the junction of two high-res and two low-res bricks. Again, disturbing artifacts at the brick boundaries are visible in the standard volume rendering (Fig. 9, left) whereas our method (Fig. 9, right) obtains a smooth function. Table 1 lists the framerates of both data sets, as tested on a Core 2 Duo with 3 GHz, 4 GB RAM and an NVidia Geforce GTX 280[†].

Calculating the brick cache fixup step does not impose an additional limitation of the frame rates, as this step is only performed whenever the cache itself changes, e.g., after a transfer function change. However, the brick cache can be updated several times a second, if necessary.

[†] The framerates have been updated after paper publication time due to shader optimizations.

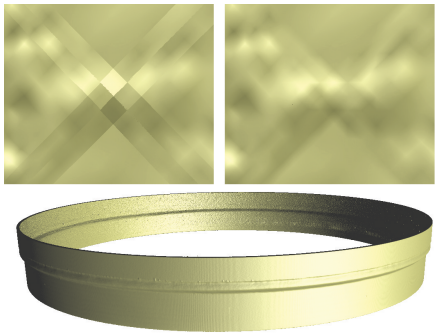

Figure 9: *High-resolution industrial CT scan of a metal ring (1518x1518x232). Zoom-in: Junction of two low-res and two high-res bricks. Discontinuous (left) vs. continuous (right).*

Data set	Fig.	Resolution	Cache Size	Transition	Discontinuous		Smooth	
				Bricks	unshaded	shaded	unshaded	shaded
Abdomen	8	$512 \times 512 \times 1112$	$512 \times 512 \times 512$	29%	60 fps	41 fps	33 fps	9 fps
Ring	9	$1518 \times 1518 \times 232$	$512 \times 512 \times 256$	22%	65 fps	45 fps	35 fps	6 fps

Table 1: *Frame rates of our mixed-resolution approach with and without smooth transitions between resolution levels obtained by warping texture coordinates in the fragment shader. Measured for 512x512 viewport on a Geforce GTX 280.*

6. Conclusion

In this paper we have introduced a mixed-resolution volume rendering approach for high-quality rendering of large data. We use a downsampling scheme where the samples are shifted by half a voxel in each dimension, permitting more flexibility in the choice of downsampling filter kernels. Our approach offers C^0-continuous transitions between different resolution levels by special handling of high-resolution brick boundaries which are adjacent to low-resolution bricks. We do this by warping the texture coordinates which are used for hardware-native tri-linear interpolation of the sample's value during ray-casting. Prior to rendering, the duplicated voxels in the border outside high-res bricks at each high-res/low-res boundary are adjusted in the corresponding high-resolution brick by a brick cache fixup step.

All the necessary steps to ensure continuous transitions between resolution levels are implemented on the GPU and offer interactive frame rates. Furthermore, we have described an efficient texture packing scheme that allows to dynamically store bricks of different resolution in the same large 3D cache texture.

In the future we want to integrate a LOD selection algorithm into our system to reduce the screen-space error that is introduced by rendering arbitrarily chosen low-resolution bricks. Additionally, we want to perform an evaluation and in-depth comparison of different multi-resolution schemes for volume rendering, with the emphasis on memory consumption, required overhead for sample replication at brick boundaries, and sample placement for downsampling.

7. Acknowledgments

We thank Agfa Vienna and the Austrian Foundry Research Institute for the data sets, and the Austrian funding agency FFG for funding.

References

[BNS01] BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer 17* (2001), 185–197.

[EMBM06] ENTEZARI A., MENG T., BERGNER S., MÖLLER T.: A Granular Three Dimensional Multiresolution Transform. In *Proc. of Eurographics/IEEE VGTC Symposium on Visualization* (2006), pp. 267–274.

[GS04] GUTHE S., STRASSER W.: Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics 28* (2004), 51–58.

[KE02] KRAUS M., ERTL T.: Adaptive Texture Maps. In *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2002), pp. 7–15.

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

[LHJ99] LAMAR E., HAMANN B., JOY K.: Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proc. of IEEE Visualization* (1999), pp. 355–362.

[LKP06] LINK F., KOENIG M., PEITGEN H.-O.: Multi-Resolution Volume Rendering with per Object Shading. In *Proc. of Vision, Modeling and Visualization* (2006), pp. 185–191.

[LLY06] LJUNG P., LUNDSTRÖM C., YNNERMAN A.: Multiresolution Interblock Interpolation in Direct Volume Rendering. In *Proc. of Eurographics/IEEE-VGTC Symposium on Visualization* (2006), pp. 259–266.

[LLYM04] LJUNG P., LUNDSTROM C., YNNERMAN A., MUSETH K.: Transfer Function based Adaptive Decompression for Volume Rendering of Large Medical Data Sets. In *Proc. of IEEE Symposium on Volume Visualization and Graphics* (2004), pp. 25–32.

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization. In *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2000), pp. 109–118.

[WE98] WESTERMANN R., ERTL T.: Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings of SIGGRAPH* (1998), pp. 169–178.

[WKE99] WESTERMANN R., KOBBELT L., ERTL T.: Real-time exploration of regular volume data by adaptive reconstruction of iso-surfaces. *The Visual Computer 15*, 2 (1999), 100–111.

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMAN K., ERTL T.: Level-Of-Detail Volume Rendering via 3D Textures. In *Proc. of IEEE Symposium on Volume Visualization* (2000), pp. 7–13.