Optimizing GPU Volume Rendering

Daniel Ruijters
Philips Medical Systems
Veenpluis 6
5680DA Best, the Netherlands
danny.ruijters@philips.com

Anna Vilanova
Technische Universiteit Eindhoven
Den Dolech 2
5600MB Eindhoven, the Netherlands
a.vilanova@tue.nl

ABSTRACT

Volume Rendering methods employing the GPU capabilities offer high performance on off-the-shelf hardware. In this article, we discuss the various bottlenecks found in the graphics hardware when performing GPU-based Volume Rendering. The specific properties of each bottleneck and the trade-offs between them are described. Further we present a novel strategy to balance the load on the identified bottlenecks, without compromising the image quality. Our strategy introduces a two-staged space-skipping, whereby the first stage applies bricking on a semi-regular grid, and the second stage uses octrees to reach a finer granularity. Additionally we apply early ray termination to the bricks. We demonstrate how the two stages address the individual bottlenecks, and how they can be tuned for a specific hardware pipeline. The described method takes into account that the rendered volume may exceed the available texture memory. Our approach further allows fast run-time changes of the transfer function.

Keywords

Volume Visualization, Direct Volume Rendering, Texture Slicing, Hierarchical Rendering, GPU.

1. INTRODUCTION

New developments in medical imaging modalities, numerical simulations, geological measurements, etc. lead to ever increasing sizes in volumetric data. The ability to visualize and manipulate the 3D data interactively is of great importance in the analysis and interpretation of the data. The interactive visualization of such data is a challenge, since the frame rate is heavily depending on the amount of data to be visualized. Inherently, the demand for faster visualization methods is always existing, in spite of hardware innovations.

An established method for interactive volume rendering on consumer hardware is GPU-based texture slicing [Ake93, CCF94, CN93, EE02, EKE01, MGS02, RGW+03, KW03]. Although this approach performs very well compared to CPU-based algorithms, since it benefits from the parallelism available in the GPU pipeline, it can be accelerated significantly by taking into account the various bottlenecks that are encountered in the graphics hardware. Every individual bottleneck has a different optimal data chunk size and data throughput. In this article, we present a novel approach to accelerate GPU-based volume rendering that allows to tailor and balance the load on the individual bottlenecks to reach an optimal exploitation of the graphics hardware power.

In section 2, we present an overview of related work. Section 3 discusses the main bottlenecks that come into play when performing GPU-based volume rendering. Then an outline of the proposed approach is drawn in section 4. Sections 5, 6 and 7 deal with the details of our approach. In section 8, the results are presented and discussed, and in section 9 we summarize our conclusions.

Journal of WSCG, ISSN 1213-6972, Vol.14, 2006
Plzen, Czech Republic.

2. RELATED WORK

The first rendering methods using the 3D texture capabilities of the graphics hardware were proposed by Cullip and Neumann [CN93], Akeley [Ake93] and Cabral et al. [CCF94]. Essentially these techniques consist of drawing polygons, which slice the volume in a back to front order. The data set is mapped as texture information on the polygons using tri-linear interpolation. The successive polygons are blended into the existing image.

Bricking is a technique to divide the volume data set into chunks, called bricks [Eck98, WWE04]. It can be employed to deal with data sets exceeding the available texture memory. The bricks have then a size that is equal to or smaller than the size of the texture memory, and are loaded sequentially from main memory into the texture memory while rendering. However, this leads to significantly lower frame rates, since the bus architecture, connecting the graphics hardware to the main memory and CPU, proves to be a major bottleneck. Tong et al. [TWTT99] propose a bricking technique that allows skipping empty regions. Their method, however, requires new textures to be generated for every change of the transfer function, which is time consuming for very large data sets.

Texture compression can help to fit the entire volume in the main memory, and to alleviate the bus bottleneck. However, all presently available compression methods supported by graphics hardware (S3TC, FXT1, DXT1, VTC, etc) are limited to lossy 8-bit RGB(α) compression, which make them unsuitable for the compression of the (often 12- or 16-bit) scalar values found in medical data, and therefore we do not use them. Further, Meissner et al. [MGS02] show that the lossy compression algorithms severely reduce the image quality. Wavelet compression, as proposed by Guthe et al. [GWGS02] is a promising technique, but there, not all parts of the volume are rendered at the highest resolution.

Not rendering all parts of the volume in the highest resolution possible is a way to reach higher frame rates, as demonstrated by LaMar et al. [LHJ99], Weiler et al. [WWH+00], Boada et al. [BNS01] and Guthe et al. [GWGS02]. This is particularly suited to increase the render speed for perspective projections in a small view port, focusing on a detail of the volume. However, orthogonal projections of the entire volume in high resolution view ports, as is common in medical applications, can only profit from this technique at the cost of the image quality.

Space-skipping and space-leaping are techniques to accelerate volume rendering, that origin from ray-casting methods, see e.g. Levoy [Lev90], Zuiderveld et al. [ZKV92] and Yagel and Shi [YS93]. It is based on skipping empty parts of the volume. The idea of space-skipping can be applied to texture-mapping volume rendering as has been shown by Westermann and Sevenich [WS01].

Octree is an established multi-level data structure when dealing with voxel volumes, which has been used in numerous different applications. E.g. Srinivasan et al. [SFH97] apply an octree structure in volume rendering. Orchard and Möller [OM01] demonstrated the benefits of using adjacency information in splatting volume rendering.

Parker et al. have combined bricking and multi-level data structures to accelerate CPU-based iso-surface ray-tracing of volume data sets on multi-processor platforms and clusters [PSL+99, DPH+03]. Grimm et al have applied a two-staged space skipping, based on bricking and octrees, combined with gradient caching, to CPU-based ray-casting [GBKG04].

Roettger et al. [RGW+03] describe a GPU-based pre-integrated texture-slicing including advanced lighting. The authors also describe a GPU-based ray-tracing approach with early ray termination. Krüger and Westermann [KW03] propose a method to accelerate volume rendering based on early ray termination and space-skipping in a GPU-based ray-casting approach. The space-skipping addresses the rasterization bottleneck, using a single octree level only.

We have combined some of the techniques cited above, to accelerate GPU volume rendering on a single workstation, using off-the-shelf hardware. Often we found that acceleration of volume rendering has been treated as a singular problem to solve. We rather focus on the individual bottlenecks that are encountered while performing volume rendering, and tailor the different techniques to address specifically those bottlenecks.

3. BOTTLENECKS

Figure 1 illustrates the graphics pipeline, employed for GPU-based volume rendering [Zel02]. Here we discuss the most important points in the pipeline that result in a bottleneck.

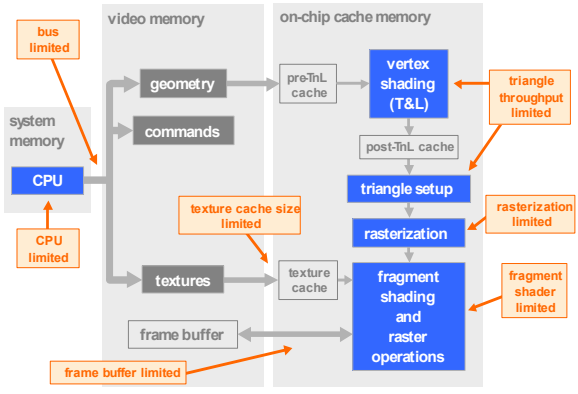
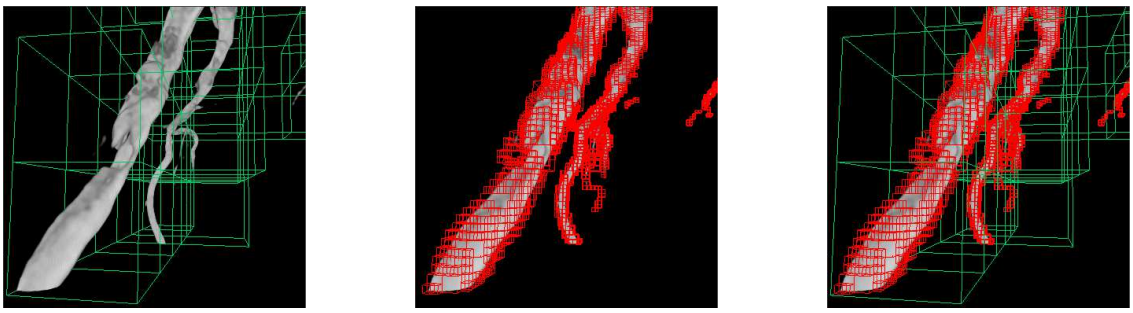


Figure 1: The graphics hardware pipeline and its bottlenecks [Zel02], light grey: memory units, dark grey: data structures, blue: processing units, red: bottlenecks.

The bus - The volume data has to be transferred over the bus from the system memory into the



(a) (b) (c)

Figure 2: The same volume fragment, rendered with (a) bricking cubes visible, (b) octree cubes visible (note the various cube sizes) and (c) both bricking and octree cubes visible

graphics card memory. Since this is the slowest part of the entire pipeline, these transfers have to be as few as possible.

Triangle throughput - The triangle throughput is mainly limited by the vertex shading and triangle setup phase. A straight forward implementation of texture-mapping volume rendering would involve only few triangles, but techniques for space-skipping may increase the amount of triangles considerably. If the triangle count becomes too high, this will become a limiting factor for the frame rate.

Rasterization - When performing volume rendering based on texture slicing, the vast majority of the pixels on the screen are accessed multiple times. Space-skipping techniques may be used to reduce the amount of pixels to be accessed, but this also increases the triangle count.

Texture cache size - Texture lookup is one of the more time consuming operations performed during the rasterization step. When the texture fits in the cache, these lookup operations will be faster.

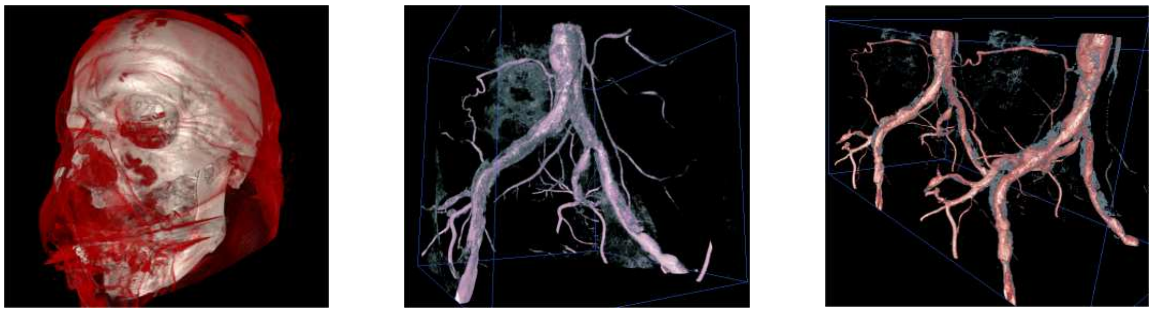
Fragment shader - Fragment shader programs impact the duration of the rasterization step. Simple fragment programs, such as applying a lookup table, generally do not limit the frame rate, however more complex operations, such as specular lighting [MGS02, RGW+03], multi-dimensional transfer functions [KKH01] or pre-integrated rendering [EE02, EKE01, RGW+03], can form a bottleneck. Especially fragment programs that perform multiple texture lookups (e.g. on-the-fly gradient calculation for specular lighting) are relatively slow.

4. OUR APPROACH

When performing volume rendering usually only a fraction of all voxels actually contribute to the final image, since a relatively small amount of voxels are of interest and another amount of them are occluded. In 3D medical data sets (obtained by e.g. ultrasound, CT, MR or rotational angiography [KodBA98, vdB03]) the anatomical structures of interest encapsulated in the data sets occupy only a part of the total data. Typically 5% to 40% of all voxels contain visible data, and even highly filled CT or MR data sets rarely exceed 55%. Especially vascular data sets can be marked as sparse data sets, since vessels, due to their tubular form, occupy only a small percentage of the volume (1% to 8%).

In this article, we seek to reach the maximum benefit in exploiting skipping void parts of the volume (space-skipping). The novelty we introduce lies in dividing the space-skipping in two stages; a course division using bricking (figure 2a) and a finer one using octrees (figure 2b). These steps are based on an analysis of the bottlenecks encountered in the graphics pipeline when performing texture-mapping volume rendering. The first stage, bricking, is chopping the volume in so called texture bricks. The bricks are loaded into the video memory, to serve as data for the volume rendering algorithm, which is executed by the GPU. The bricks address the bus- and texture cache size-bottleneck. To further alleviate the load on the fragment shaders, we additionally perform early ray termination to each brick. This benefits especially highly-filled data sets. The second stage is employing an octree within each brick. The octrees address the rasterization bottleneck. As we demonstrate, the two stages have to be balanced, because lifting one bottleneck may overload another bottleneck (e.g. rasterization bottleneck versus triangle throughput bottleneck).

The role of the transfer function in volume rendering is to map the scalar voxel information to optical properties (e.g. color and opacity) [KKH01]. The above described approach is implemented such that the flexibility to change the transfer function at run-time is preserved. This offers the possibility to focus on different scalar ranges in the volume, without lengthy calculations. To accomplish this, the unmodified scalar voxel values are stored in the brick textures, and a fragment shader program is used, to lookup the RGBα values post-interpolatively.



(a) (b) (c)

Figure 3: Test volumes: (a) 512^3 volume, used for testing early ray termination, (b) vascular 512^3 volume, (c) gigabyte volume of $642 \cdot 642 \cdot 1284$ voxels, generated by duplicating a large 3D-RA volume.

5. BRICKING

As mentioned in section 2, the voxel volume can be divided into chunks, called bricks, in order to cope with voxel data sets sizes exceeding the size of the texture memory of the graphics hardware. Note that our bricks contain the original scalar values of the voxel volume, thus the values *before* applying the transfer function. This enables us to change the transfer function on the fly, since a transfer function change does not require creating new textures.

To obtain a correct interpolation at the bricks' boundaries it is necessary that the data held by adjacent bricks overlap. The overlap depends on the convolution kernel used for interpolation [ML94], and should correspond to (*kernelsize* - 1). For nearest neighbor interpolation that means that no overlap is needed, since the width of the kernel is one. For tri-linear interpolation the overlap should be one voxel in every direction (for other kernels the overlap may even be larger). Pre-integrated rendering [EE02, EKE01, RGW+03] or the on-the-fly calculation of gradients require the overlap to be increased by another voxel in every direction. For bricks of b^3 voxels and an overlap of n voxels, the memory overhead is approximately $(3n/b)\cdot 100\%$.

The bricks are loaded into the video memory as 3D textures. Many graphics cards require 3D texture sizes to be a power of 2 in every direction. If the volume dimensions do not divide evenly into brick dimensions, either an additional layer of partially empty bricks should be added in each direction, or smaller rest-bricks should be used.

When the amount of data in the textures exceeds the available texture memory, textures are swapped between the main memory and the texture memory. If a requested brick is not resident in the texture memory, it is loaded from the main memory, replacing resident textures [SWND03]. In most OpenGL implementations resident textures are swapped out on a Least Recently Used (LRU) base.

Traditionally bricking in texture based rendering is used to be able to render data sets which exceed the size of the texture memory of the graphics hardware. The bricks are then chosen to be as large as possible, and they are sequentially loaded from the main memory into the texture memory. Which implies that for each frame the entire volume data is transferred over the bus.

In our approach, however, we choose brick sizes which are considerably smaller. The smaller the brick size is, the bigger is the chance of bricks being completely void *after* applying the transfer function, and void bricks do not need to be drawn. Therefore, once they are swapped out of the texture memory, they are never reloaded into the texture memory, and thus the bus bottleneck is alleviated.

We even apply bricking to volumes which completely fit into the texture memory to improve data locality, which will result in less cache trashing on the graphics card [HG97, CBS98, IEP98]. On the other hand smaller bricks could introduce a larger overhead due to the overlap needed for interpolation. Thus the optimal brick size needs to be defined depending on the available texture memory, optimal texture size (see section 3), nature of the data set, overhead due to overlap, and the constraints posed by the graphics hardware.

6. EARLY RAY TERMINATION

To be able to perform early ray termination at all, the volume has to be traversed in a front-to-back order. This can be done by evaluating the volume rendering integral in discrete steps, using the under operator:

$$C_{i+1} = (1 - A_i) \cdot \alpha_i \cdot c_i + C_i$$

$$A_{i+1} = (1 - A_i) \cdot \alpha_i + A_i$$

Whereby C, A denote the color, respectively the opacity value of the current ray, c, α the color and opacity value given by applying the transfer function to the current sample in the volume, and i denotes the

sample index. A ray is then saturated when A_i approximates 1.

Before a brick is rendered, early ray termination is applied to its destination pixels. This is tested by executing a fragment shader program, while drawing a solid bounding box around the brick with back face culling switched on. The fragment shader program writes the maximal value in the depth buffer for saturated rays [KW03, RGW+03]. When slicing the brick texture the early z-test will prevent any fragment operations to be executed for those rays, reducing the load on the rasterization and fragment shader bottlenecks. Early ray termination is only performed once per brick, and not more often (e.g. for every octree node or every sample) because the overhead involved (changing fragment shaders, performing the test) would otherwise annihilate the benefits.

7. OCTREE

By not rendering the void bricks, the load on the rasterization bottleneck is already reduced. We seek to reduce it further by applying octrees. Every brick possesses its own octree. Every octree node corresponds to a cuboid part of the voxel volume, which can be divided into eight parts, corresponding to the child nodes (see figure 4). Our octree is kept in main memory. It only describes the geometry of the visible data. The actual voxel data is to be found in the brick textures.

For tri-linear interpolation, let a cell be defined as a cube, whose eight corners adjacent voxel values are assigned. For every position within the cell an intensity value is defined as the tri-linear interpolation of the corner values. Therefore a cell can only be completely void if its eight corner values are completely transparent ($\alpha = 0$) *after* applying the transfer function. This definition can easily be extended to any given interpolation kernel, by setting the size of a cell to $(kernelsize - 1)^3$.

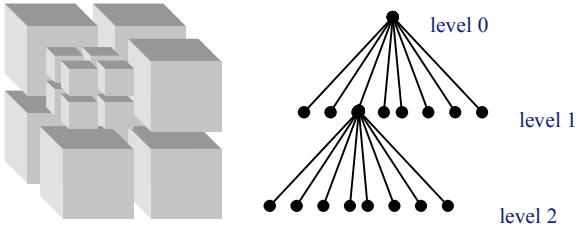


Figure 4: An octree division and its tree.

Every octree node carries a variable describing the ratio r of visible data to total data within its cube. At the final level of the octree, every node represents uniquely one cell, and is considered either completely filled ($r = 1$) or void ($r = 0$). Every higher octree level nodes ratio can be calculated by averaging the ratios of its children. This calculation only needs to be performed when the transfer function has changed.

Rendering an image means that the bricks have to be processed in a front to back order. For each brick the respective octree is traversed, starting with its parent node. Depending on its ratio r there are three ways to process a node:

$r = 0$: The node is completely void. It is not drawn at all, and is not traversed any further.

$0 < r <$ threshold: The nodes children will be traversed, and to each child node this strategy will be applied recursively.

$r \geq$ threshold: The node is drawn completely. It is not traversed any further.

If the *threshold* is set to 1, exactly all filled cells will be drawn, and no void cells. However, that would lead to a lot of tiny cubes at the boundaries of the visible data structures, and thus the load on the triangle throughput bottleneck becomes too high. Therefore the *threshold* should be chosen in such a way that some degree of void data is allowed to be drawn. A further strategy we use to prevent too much overhead is setting an octree level at which nodes, lower in the hierarchy, are not traversed any further. At this level, any node that is not void, will be drawn completely.

When traversing a node, its children have to be sorted in a front to back order. Since there are eight children, it would seem that there are 8! = 40320 ways to arrange the children. But since the arrangement along the three perpendicular axes is the same for all children, there remain $2^3 = 8$ possible orders. When a node is to be drawn, the cuboid box corresponding to this node is sliced, and the slices are rasterized and blended into the previously drawn slices. The slices can be axis-aligned or viewport-aligned. For the most straight-forward form of volume rendering, the brick texture is interpolated on every slice, taking its position in the brick into account, and after interpolation the transfer function is applied. However, it is also possible to perform more sophisticated forms of volume rendering on the slices, like pre-integrated volume rendering or include specular lighting [MGS02, RGW+03].

The octree is generated and traversed on the CPU. Its purpose is to lower the workload on the graphics pipeline, and thus the GPU. The octree reduces the time that the GPU spends on processing data which never contribute to the final image. The actual volume rendering algorithm, as well as interpolation, the post-interpolative transfer function, and optionally, specular lighting, is being performed by the GPU.

Graphics card	(a) Optimized	(b) Non-optimized	(a) / (b)
nVidia QuadroFX 3400	73.5 fps	9.6 fps	7.66
ATi FireGL X1, xy aligned	83.3 fps	0.23 fps	362
ATi FireGL X1, non xy aligned	27.4 fps	0.23 fps	119
3Dlabs Wildcat 7110	21.3 fps	0.38 fps	56.1

Table 1: Average frame rates reached when using (a) best combination of bricking and octrees, (b) GPU rendering without bricking or octrees.

8. RESULTS

The described approaches have been tested with several different graphics cards: the nVidia QuadroFX 3400 (256MB on board memory), the ATi FireGL X1 (128MB), and the 3DLabsWildcat 7110 (256MB). With each card the volume in figure 3b has been rendered, using the same lookup table settings. The volume data concerned the iliac vein, acquired through 3D rotational angiography. Since contrast media was injected into the vein, the vein could easily be classified using the transfer function. Only 3% of the voxels in this volume contain visible data. All results have been obtained, using a view port of 800^2 pixels and the sample rate for the volume rendering equation was set to 1.5 samples per voxel.

Since the optimal brick size is mainly determined by the properties of the texture memory (see section 5) and the optimal octree limit is primarily used to balance the rasterization load and the triangle throughput (see sections 3 and 7) they can be considered to be fairly orthogonal variables. Therefore their optimum can be found by varying one variable, while keeping the other one constant.

On each graphics card the test volume was rendered with different brick sizes, see figure 5, while the octree limit was set to 8^3 voxels.

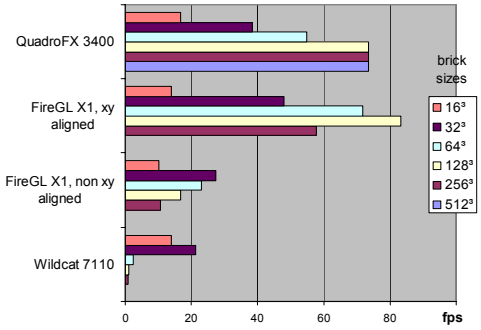


Figure 5: Performance using different brick sizes.

The ATi FireGL X1 and the 3DLabs Wildcat 7110 clearly show that their optimal brick size is considerably smaller than their largest possible brick size. The nVidia QuadroFX 3400 does not benefit from the bricking for the 256MB test volume. However, also this card clearly profits from the bricking for the sparse 1GB volume in figure 3c: the optimal brick size is then 64^3 voxels, with an average frame rate of 37 fps, while for 256^3 bricks only a mere 3.1 fps is reached.

The performance of the ATi FireGL X1 depends heavily on the sampling direction of the bricks, because the ATi card treats the 3D textures as a stack of 2D slices. When the bricks are traversed in the x or y direction, the slices are accessed rather linear, and the performance is much better than when they are traversed in the z direction. It is inevitable to traverse in the z direction, when the viewing direction and the z-axis of the textures differ more than 45°. This effect can be reduced by alternating the orientation of the textures for each consecutive brick [WWE04]. Especially striking is the fact that the optimal brick size and octree limit is different for each sampling direction. When sampled in the xy-plane direction larger bricks benefit from linear traversal, while in other directions smaller bricks benefit from less cache trashing. In figures 5 and 6 this fact is illustrated by the performance measurement when sampling aligned to the xy-plane, and when not.

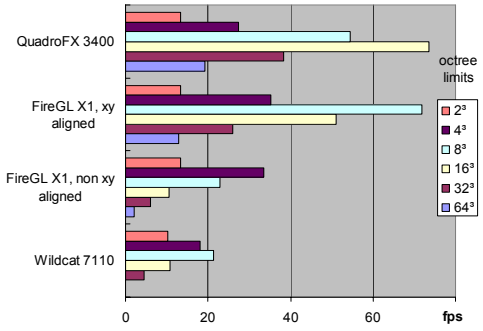


Figure 6: Performance using different octree limits.

Further the volume was rendered with a fixed brick size of 64^3 voxels and variable octree limits (the octree limit is the smallest octree cube allowed). Not every octree branch reaches this limit, see section 7. Figure 6 unsurprisingly shows that there is an optimum octree size for every graphics card. Smaller octree limits lead to too much CPU overhead and triangle count, and larger octrees to too much rasterization overhead. The 64^3 octree level

corresponds to not using any octrees at all, only bricking.

Table 1 shows the acceleration achieved, using the volume in figure 3b, with an optimal combination of brick size and octree depth for each particular graphics card versus the same GPU volume rendering routines applied without any bricking or octrees at all. Since early ray termination does not provide any performance gain for sparse data sets, it was not used on this volume.

Early ray termination was tested on the QuadroFX 3400 using the volume in figure 3a. GPU volume rendering without optimizations yielded 2.2 fps, using 64^3 bricks and 8^3 octree limits 5.2 fps were reached, and with additionally early ray termination switched on, the average frame rate was 16.1 fps.

Since the rendering primarily depends on the graphics card, replacing e.g. a Xeon 3.0GHz by a Xeon 1.7GHz delivered approximately the same performance figures. The only part which is bounded by the CPU and main memory performance is building a new octree after the transfer function has been changed. For a volume consisting of 512^3 voxels (16 bit per voxel, 256MB for the entire volume), rendered with a brick size of 64^3 voxels and an octree limit of 8^3 voxels, building all new octrees for the entire 512^3 volume took 6.5 milliseconds on the Xeon 1.7GHz and 3.5 milliseconds on the Xeon 3.0GHz machine.

9. CONCLUSIONS

In this paper, we presented an approach to accelerate GPU-based volume rendering. The approach consisted of a two staged space-skipping and early ray termination, and was tailored to lift the various bottlenecks encountered in the graphics pipeline.

In the first stage, the entire volume is chopped into bricks, and from these bricks 3D textures are created. Empty bricks are never drawn, nor kept in the video memory, and therefore the bus bottleneck is relieved. The optimal brick size depends on the nature of the data (there should be a reasonable chance that there are bricks which are completely void), the available texture memory, the texture cache size and the overhead introduced by brick overlap. Since the brick textures' content does not depend on the transfer function, they need to be created only once for static data.

The octrees, which form the second stage, focus on skipping data that is not visible after applying the transfer function. In this way the rasterization bottleneck is addressed. To prevent too much overhead to be introduced, a certain amount of void data per octree box is allowed, and there is a limit to the granularity of the octree boxes. The optimal octree parameters are determined by the weight of the rasterization phase (i.e. are there complex fragment shader programs involved, etc.) and the trade-off between less rasterization operations and more triangles (triangle throughput bottleneck). Since the octree depends on the transfer function, it has to be recalculated when the transfer function changes.

In this article it has been shown how the individual bottlenecks have been addressed by a two-folded approach. First the bus bottleneck and texture cache size has been addressed by bricking, and consequently the rasterization bottleneck has been addressed by the octrees. The rasterization and fragment shader bottleneck were further lifted by employing early ray termination. The results show that the parameters can be optimized for different graphics cards. Since the transfer function only leads to recalculating the octrees, and not reloading the bricks, it can also be changed quickly and interactively.

The graphics industry are introducing more powerful hardware at an impressive pace. However developments in medical imaging modalities are equally impressive, resulting in larger volume data sets. Which means that in the foreseeable future the techniques that were presented here will preserve their benefits.

10. REFERENCES

[Ake93] K. Akeley. Reality Engine Graphics. In Proc. SIGGRAPH'93, volume 27, pp. 109-116, 1993.

[BNS01] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-based Octree. The Visual Computer, (17), pp. 185-197, 2001.

[CBS98] M. Cox, N. Bhandri, and M. Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. In ISCA '98, pp. 86-97, 1998.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. Proc. of the 1994 symposium on Volume visualization, pp. 91-98, 1994.

[CN93] T. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, 1993.

[DPH+03] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, C. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In Proc. 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, pp. 87-94, 2003.

[Eck98] G. Eckel. OpenGL Volumizer Programmer's Guide. Silicon Graphics, Inc, 1998.

[EE02] K. Engel and T. Ertl. Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware. In Eurographics '02 - State of the Art Report, 2002.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-quality Pre-integrated Volume Rendering using Hardware-Accelerated Pixel Shading. Proc. of the 2001

Eurographics workshop on Graphics hardware, pp. 9-16, 2001.
[GBKG04] S. Grimm, S. Bruckner, A. Kanitsar and E. Gröller. Memory Efficient Acceleration Structures and Techniques for CPU-based Volume Raycasting of Large Data. IEEE Symposium on Volume Visualization and Graphics, pp. 1-8, 2004.
[GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive Rendering of Large Volume Data Sets. Proc. IEEE Visualization 2002, pp. 53-60, 2002.
[HG97] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In ISCA '97: Proc. of the 24th annual international symposium on Computer architecture, pp. 108-120, 1997.
[IEP98] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a Texture Cache Architecture. In Proc. of the 1998 Eurographics Workshop on Graphics Hardware, pp. 133-142, 1998.
[KKH01] J. Kniss, G. Kindlmann, and C. Hansen. Interactive Volume Rendering using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. Proc. IEEE Visualization 2001, pp. 255-262, 2001.
[KodBA98] R. Kemkers, J. op de Beek, and H. Aerts. 3D-Rotational Angiography: First Clinical Applications. Proc. in Computer Assisted Radiology and Surgery, pp. 182-187, 1998.
[KW03] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In Proc. IEEE Visualization 2003, pp. 287-292, 2003.
[Lev90] M. Levoy. Effcient Ray Tracing of Volume Data. ACM Transactions on Graphics 9(3), pp. 245-261, 1990.
[LHJ99] E. LaMar, B. Hamann, and K. I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In Proc. IEEE Visualization '99, pp. 355-361, 1999.
[MGS02] M. Meissner, S. Guthe, and W. Strasser. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In Graphics Interface 2002, pp. 209-218, 2002.
[ML94] S. R. Marschner and R. J. Lobb. An Evaluation of Reconstruction Filters for Volume Rendering. Proc. IEEE Visualization '94, pp. 100-107, 1994.
[OM01] J. Orchard and T. Möller. Accelerated Splatting using a 3D Adjacency Data Structure. In Graphics interface 2001, pp. 191-200, 2001.
[PSL+99] S. Parker, P. Shirley, Y. Livnat, C. Hansen, P.-P. Sloan, M. Parker. Interacting with Gigabyte Volume Datasets on the Origin 2000. The 41st Annual Cray User's Group Conference , 1999.
[RGW+03] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart Hardware-Accelerated Volume Rendering. In VisSym'03: Proc. of the symposium on Data Visualisation 2003, pp. 231-238, 2003.
[SFH97] R. Srinivasan, S. Fang, and S. Huang. Rendering by Template-based Octree Projection. Proc. of the 8th Eurographics Workshop on Visualization in Scientific Computing, pp. 155-163. Eurographics, 1997.
[SWND03] D. Shreiner, M. Woo, J. Neider, and T. Davis. OpenGL Programming Guide: The Offcial Guide to Learning OpenGL (red book). Addison-Wesley Pub Co, 4 edition, 2003.
[TWTT99] X. Tong, W. Wang, W. Tsang, and Z. Tang. Efficiently Rendering Large Volume Data Using Texture Mapping Hardware. In Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym), pp. 121-132, 1999.
[vdB03] J. C. van den Berg. Three-Dimensional Rotational Angiography. Endovascular Today, (March 2003), 2003.
[WS01] R. Westermann and B. Sevenich. Accelerated Volume Ray-Casting using Texture Mapping. Proc. IEEE Visualization 2001, pp. 271-278, 2001.
[WWH+00] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-Of-Detail Volume Rendering via 3D Textures. In Proc. Volume Visualization and Graphics Symposium 2000, pp. 7-13, 2000.
[WWE04] D. Weiskopf, M. Weiler, T. Ertl. Maintaining Constant Frame Rates in 3D Texture-Based Volume Rendering. Computer Graphics International 2004 (CGI'04), pp. 604-607, 2004.
[YS93] R. Yagel and Z. Shi. Accelerating Volume Animation by Space-Leaping. Proc. IEEE Visualization '93, pp. 62-69, 1993.
[Zel02] C. Zeller. Balancing the Graphics Pipeline for Optimal Performance, Graphics Developer Conference 2002, http://developer.nvidia.com/, 2002.
[ZKV92] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Accelaration of Ray-Casting Using 3D Distance Transform. Proc. of Visualization in Biomedical Computing II, pp. 324-335, 1992.