Memory Efficient Acceleration Structures and Techniques for CPU-based Volume Raycasting of Large Data

Sören Grimm*
Vienna University of Technology
Austria

Stefan Bruckner†
Vienna University of Technology
Austria

Armin Kanitsar‡
Tiani Medgraph AG
Vienna, Austria

Eduard Gröller§
Vienna University of Technology
Austria

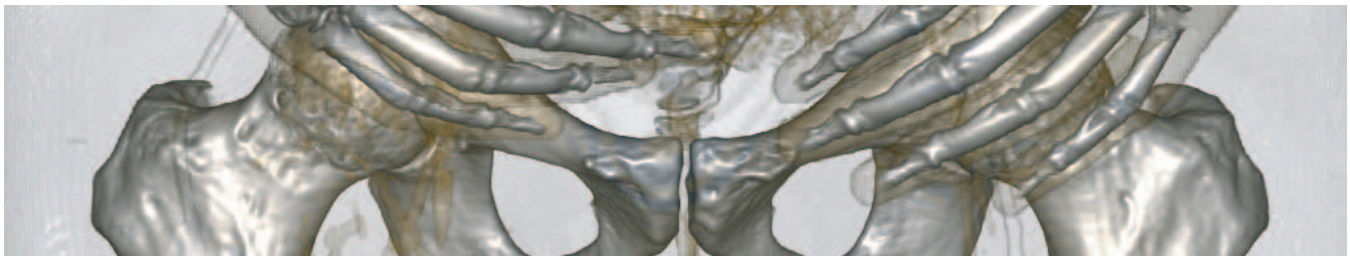


Figure 1: Close-up of the visible male.

ABSTRACT

Most CPU-based volume raycasting approaches achieve high performance by advanced memory layouts, space subdivision, and excessive pre-computing. Such approaches typically need an enormous amount of memory. They are limited to sizes which do not satisfy the medical data used in daily clinical routine. We present a new volume raycasting approach based on image-ordered raycasting with object-ordered processing, which is able to perform high-quality rendering of very large medical data in real-time on commodity computers. For large medical data such as computed tomographic (CT) angiography run-offs (512x512x1202) we achieve rendering times up to 2.5 fps on a commodity notebook. We achieve this by introducing a memory efficient acceleration technique for on-the-fly gradient estimation and a memory efficient hybrid removal and skipping technique of transparent regions. We employ quantized binary histograms, granular resolution octrees, and a cell invisibility cache. These acceleration structures require just a small extra storage of approximately 10%.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: volume raycasting, large data, acceleration techniques

1 INTRODUCTION

Direct Volume Rendering is known as a powerful technique to visualize complex structures within three-dimensional data. Its main advantage, compared to standard 3D surface rendering, is the ability to perform translucent rendering in order to provide more information about spatial relationships of different structures. In general 3D visualization helps to understand patient's pathological conditions, improves surgical planning, and is a big aid in medical education. A typical data size of today's clinical routine is up to 512x512x512. However, some examinations, such as peripheral CT angiography run-offs, require even larger scans. For example Rubin et al. [16] reported a mean of 908 transverse reconstructions. Furthermore due to improved capabilities of newer acquisition devices it is possible to scan with higher resolution. The higher resolution is often used for difficult cases which also results in larger data. This large data presents a challenge to current rendering architectures and techniques. The increasing demand of interactive 3D visualization is basically driven by the size itself. Conventional slicing methods already reach their limit of usability due to the enormous amount of slices. 3D visualization is more and more explored as an attractive alternative additional method for examinations of large medical data to support the obliged 2D examination. Figure 1 shows an example of a 3D visualization.

Within the research area of accelerating volume rendering, two main research streams can be distinguished. One stream is focused on exploiting special purpose hardware such as Volume Pro (Pfister et al. [15]), Vizard (Meissner et al. [8]) or graphic cards (GPU) (Cabral et al. [1], Westermann et al. [19], Guthe et al. [3] and many others). This approach usually provides high performance when data fits into internal memory. However, this issue becomes the most critical bottleneck once the data size exceeds the onboard internal memory capacity. Expensive main memory to internal memory transfers have to be performed, which lead to an enormous performance penalty. Furthermore, the accelerated pace of the GPU's development cycle produces heterogenous multi-user hardware environments. This makes the adoption of such special purpose hardware solutions even more difficult. The other research stream is based on CPU technologies. In general they provide better performance for large data due to the inherent larger memory capacity. Many proposed approaches for CPU based volume raycasting achieve high performance by utilizing super-computers or clusters; e.g. Parker et al. [14] presented a volume rendering approach on an SGI Reality Monster and were capable to render the Visible Woman (approx. 1 GB) with up to 20 fps utilizing 128 processors. However, it is a large scale solution which does not apply to the needs and capacities of an ordinary medical environment.

*e-mail: grimm@cg.tuwien.ac.at
†e-mail: bruckner@cg.tuwien.ac.at
‡e-mail: kanitsar@tiani.com
§e-mail: groeller@cg.tuwien.ac.at

IEEE Symposium on Volume Visualization and Graphics 2004
October 11-12, Austin, Texas, USA
0-7803-8781-3/04/$20.00 ©2004 IEEE

The purpose of this paper is to present a solution which resolves the issues presented before: an interactive real-time volume rendering approach for large medical data, capable of performing in a heterogeneous hardware environment, by using commodity computers such as notebooks, and providing high performance and high quality images. We achieve this by introducing an efficient method for on-the-fly gradient estimation and an efficient hybrid removal and skipping technique of transparent regions. The presentation of our new approaches is subdivided as follows: Section 2 surveys related work. Section 3 presents a brief overview of our raycasting processing work-flow. In Section 4 we introduce our refined caching scheme to accelerate on-the-fly gradient estimation. In Section 5 we focus on removing and skipping of transparent regions by employing quantized binary histograms, granular resolution octrees, and a cell invisibility cache. In Section 6 we present the results. In Section 7 we discuss and give a conclusion. Finally in Section 8 we give ideas for future work.

2 Related Work

The most popular CPU-based direct volume rendering algorithms are shear-warp, splatting, and raycasting. Shear-warp is considered to be the fastest software algorithm (Lacroute et al. [6]), however the inherent bi-linear interpolation provides quality which is in general insufficient for medical purposes. Splatting was first proposed by Westhover et al. [20]. Later it was improved in terms of quality and speed by Mueller et al. [11, 12], and Hung et al. [4]. This technique provides high quality images. However it still lacks the speed provided by the general volume raycasting technique.

Volume raycasting is still widely used if high quality rendering of large data is desired. Several acceleration techniques for volume raycasting have been proposed over the last decade. Knittel et al. [5] and Mora et al. [10] proposed volume raycasting approaches for commodity computers. They achieve impressive frame-rates by using a spread memory layout and pre-computed gradients; however their method requires a huge amount of additional memory. The spread memory layout itself increases the memory usage by a factor of four. This becomes a rather limitation factor if large data needs to be handled, or if the the rendering system is part of a larger visualization systems and memory resources need to be shared.

In contrast to that, our approach does not rely on extensive precomputing or a spread memory layout; it is based on a bricked volume layout. To achieve high performance advanced acceleration structures and techniques are necessary. In the following Sections we present several memory efficient acceleration approaches.

3 Volume Raycasting Work-flow

The following paragraph presents a brief overview of the work-flow of our volume raycasting approach. Bricking of volume data is a well known method to exploit cache coherence [2, 3, 7, 14]. We decompose the volume data in bricks and perform processing brick-wise. The volume raycasting process is subdivided into pre-processing, pre-rendering, rendering, and post-rendering. The pre-processing step is done only once during start-up and the remaining steps are performed every time the image needs to be re-rendered. At first we give a brief overview of the four rendering steps.

Pre-Processing: During loading, the data is decomposed into small bricks of size 32^3. The data within the bricks and the bricks themselves are stored in common xyz-order. For each brick information about the contained density values is stored, e.g. min-max values, quantized binary histograms, etc.

Pre-rendering: In this phase transparent regions are removed and the rays-volume intersections are computed. There are eight different brick lists which are defined by the eight possible viewing-octants in 3D. Depending on the viewing direction the appropriate list is selected to process the volume brick-wise and in correct visibility order.

Rendering: According to the brick list, all rays traverse the bricks in visibility order, until all bricks are processed or all rays are terminated due to complete opacity accumulation. During traversing regular re-sampling, gradient computation, classification, shading and composition are performed.

Post-rendering: At this point the final image is displayed, written to a file, or sent over the network to a client.

A more detailed description of the used acceleration techniques and structures for the pre-rendering and rendering step is given in the following Sections. There are two major strategies to accelerate volume raycasting. The first one is to reduce the computational costs at one re-sampling location. We achieve this by using an acceleration technique for gradient estimation (Section 4). The second strategy is to efficiently remove and skip transparent regions, which we achieve by using quantized binary histograms, granular resolution octrees, and a cell invisibility cache (Section 5).

4 Efficient Gradient Caching

The most common method to accelerate gradient estimation is to read pre-computed gradients from memory. However, this acceleration technique has several drawbacks. In order to gain high performance the gradients must be stored in memory, resulting in an inefficient usage of resources. Furthermore such a solution is limited by memory bandwidth instead of preferably CPU throughput. The evolution of computer systems has shown that CPU performance increases faster than memory bandwidth. Going one step further if the data exceeds the main memory capacity, out-of-core rendering has to be performed and the gap between CPU throughput and memory bandwidth becomes even larger. Experience has shown, that not every gradient estimation scheme performs equally well on all kinds of data. Therefore the ability to switch between different gradient estimation schemes is an important feature and basically not efficiently given if pre-computing is used. Additionally, often only gradient direction is stored and the gradient magnitude is omitted, otherwise the storage requirements can become considerably high. Finally, pre-computing the gradients is quite time consuming. Considering a now-a-days medical visualization system, the doctor's main interest is to carry out the examination as fast as possible. The total time from scanning the patient to the actual examination is a highly critical factor.

To avoid these issues, our approach performs on-the-fly gradient estimation. In order to obtain highly accurate images, a dense object and image sample distance is inevitable, which implies high computational costs. A typical re-sampling resolution illustrated in 2D is shown in Figure 2a. In this case there are eight re-sample locations within a cell. Each gradient at the corners of one cell has to be computed eight times. Furthermore, each corner is shared between four cells in 2D. The total amount of redundant gradient computations at one corner is eight re-sampling positions multiplied by four cells which gives a total of 32 computations. In 3D the computational costs are even considerably higher. These very costly redundant gradient computations can be avoided by refined caching. However, not every gradient estimation scheme is suitable for caching. There are several studies on gradient filters for volume rendering with focus on accuracy, importance in terms of image quality and efficiency. Especially, Moeller et al. [9] give a thorough comparison of commonly used normal estimation schemes. They differentiate between four types of gradient estimation schemes:

1. *Continuous Derivative* uses a derivative filter which is pre-convolved with the interpolation filter. The gradient at a re-sample location is then computed by convolving the volume with this combined filter.

2. *Analytic Derivative* uses a special gradient filter derived from the interpolation filter for gradient estimation.

3. *Interpolation First* computes the derivative at the re-sample position by re-sampling the data on a new grid, such that the used derivative operator can be applied directly. This is very beneficial if orthographic rendering is performed.

4. *Derivative First* computes the gradients at the grid-points and then interpolates these at the desired re-sample position.

For scheme one and two no caching mechanism is available. Only schemes three and four can be considered for efficient gradient caching. Due to their numerical equivalence only a comparison with respect to efficiency is necessary. Moeller et al. [9] proposed the *Interpolation First* method as the most efficient one. Considering volume rendering and no caching this is quite obvious. However, applying the *Interpolation First* scheme requires re-sampling of the original grid to a much larger grid if the object sample distance is significantly smaller than one. Already an object sample distance of 0.25 increases the grid size by a factor of four. This enormous amount of data makes caching inefficient and difficult. Especially if the object sample distance should be kept dynamical or if jittering techniques are applied to improve the image accuracy. Due to these reasons the *Derivative First* gradient estimation scheme is more efficient from a performance point of view, since it is more suitable for caching. In this case, the amount of data to cache is always determined. This makes interactive changes of the object sample distance possible.

4.1 Per Brick Gradient Caching

Our caching scheme requires two data structures: the cache itself and a second structure to store the corresponding valid bits. The used processing entity is not the whole volume; in fact the volume is decomposed in bricks and each brick defines a processing entity. The size of the cache matches the number of gradients needed for one brick. The most straightforward way to use this cache would be to pre-compute all gradients which correspond to the current brick and use those during brick processing. This would be very inefficient, since more gradients than necessary would be pre-computed if only parts of a brick are visible. In contrast to that we additionally use valid bits, which encode if a gradient is already computed and stored in the cache. During brick processing every time a gradient needs to be computed, it is checked if the gradient is already stored in the gradient cache. If not, the gradient is computed and stored in the cache and the corresponding valid bit is set to true. This mechanism ensures that gradients are computed only once at each sample position during brick processing. The cache remains only valid during the processing of one brick. Once the next brick is processed the cache is reset. This has the effect that the gradients which are also needed in adjacent bricks are processed more than once. The resulting performance penalty is low, since the number of those gradients is small compared to the number of all gradients.

5 Removing and Skipping of Transparent Regions

For medical imaging, interactive classification of data is mandatory. In general during examination it happens quite often, that large parts of the data are classified as transparent to allow a more precise view of the region of interest. For acceleration purposes it

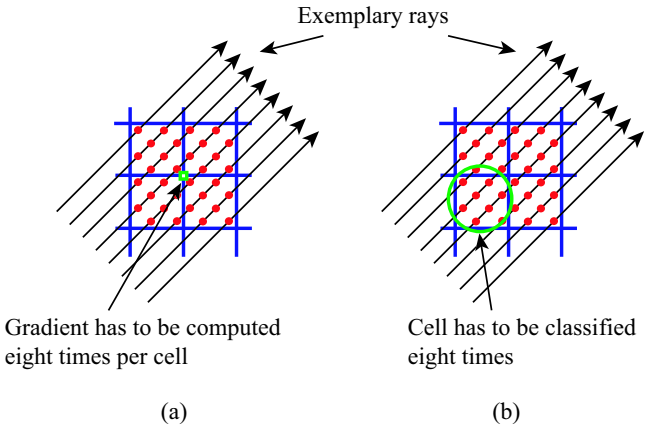


Figure 2: Typical re-sampling resolution of a cell in 2D. (a) In the shown case each gradient at the cell corners has to be computed 8 times while processing one cell. (b) In the shown case a cell has to be classified 8 times.

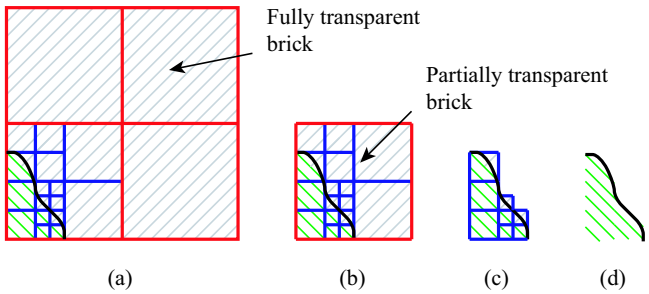


Figure 3: General work-flow of our hybrid transparent region removal and skipping technique: Red are brick boundaries, blue are octree boundaries, grey are transparent regions and green is visible volume data. (a) → (b): removal of transparent bricks, (b) → (c) removal based on octree projection and (c) → (d) removal using the cell invisibility cache.

is quite beneficial to exploit this transparency information and start the actual re-sampling of the data right where the visible data begins. The work-flow of our hybrid transparent region removal and skipping technique is shown in Figure 3. At first transparent regions are removed on a brick basis (Figure 3a → Figure 3b). Then to support even more refined removal of smaller transparent regions we perform octree projection (Figure 3b → Figure 3c). Due to efficiency reasons our octree subdivision does not fully go down to individual cells. The granular resolution of the octree leads to just approximate rays-volume intersections. To overcome the resulting performance penalty we introduce a Cell Invisibility Cache (CIC) to skip the remaining transparent cells (Figure 3c → Figure 3d). In the following we describe our hybrid transparent region removal and skipping technique in more detail.

5.1 Quantized Binary Histograms

At first we describe an efficient encoding for finding transparent bricks. The most common methods are minimum-maximum encodings and summed area tables. A summed area table encodes the

opacity integral by

$$S(0) = \alpha(0)$$
$$S(k) = S(k-1) + \alpha(k)$$

Hereby $k \in H = [0..4095]$, which is the possible range of Houndsfield units and α represents the opacity. i_{min} and i_{max} denote the minimum and maximum density value within a brick. The integral of the discrete function α over the interval $[i_{min}, i_{max}]$ can be approximated in constant time by performing two table lookups:

$$\sum_{k=i_{min}}^{i_{max}} \alpha(k) = S(i_{max}) - S(i_{min})$$

If $S(i_{max}) - S(i_{min}) = 0$ then the brick is transparent and can be skipped. At this point we differentiate between pre- and post-classification. For post-classification the min-max encoding is the most accurate, since due to interpolation of data all values between the minimum and the maximum may occur. However, if pre-classification is performed the min-max encoding may be too granular when applied on large regions. Figure 4 shows an example where the min-max encoding is too conservative. The min-max encoding would report both bricks as being visible. The main issue is, that the min-max encoding accuracy relies heavily on the underlying data. If the region is large it is quite likely that its values differ considerably. The min-max encoding becomes too granular to effectively encode the area. We use a more refined structure, i.e., a quantized binary histogram. In general a binary histogram is encoded as:

$$\sigma_x(B) = \begin{cases} 1, & x \in B \\ 0, & otherwise \end{cases}$$

Hereby B is the set of all density values a brick contains, with $B \subseteq H = [0..4095]$. $\sigma_x(B) = 1$ means that the density value x is given at least at one grid position in the underlying brick. This encoding is effective, however it is quite inefficient in terms of memory usage and efficient evaluation. Additionally to the binary codomain quantization we also quantize the domain itself. This *quantized binary histogram* is stored for each brick.

It is determined by

$$\sigma_i(B) = \begin{cases} 1, & \exists x : x \in B, x \in [128 \cdot i..128 \cdot (i+1)[\\ 0, & otherwise \end{cases}$$

Where $(0 \leq i \leq 31)$. Within quantized binary histograms the existence of data within a specific interval is encoded. The intervals are concatenated, disjunct, have same length, and cover the range of Houndsfield units. In the pre-processing phase every brick is parsed and encoded. The same encoding can be performed for the transfer-function with respect to opacity:

$$\lambda_i = \begin{cases} 1, & \exists x : opacity(x) \neq 0, x \in [128 \cdot i..128 \cdot (i+1)[\\ 0, & otherwise \end{cases}$$

Hereby $x \in H$ and $i \in [0,31]$. Every time the transfer-function changes, the transfer-function is re-encoded in this way.

With this information one can quickly determine the transparent bricks. A brick is transparent if

$$\forall i \in [0..31] : \lambda_i \wedge \sigma_i = 0$$

This conjunction test can be done very efficiently on an x86 based CPU. Note, that this is a conservative estimate of a brick's visibility. It is possible that due to the chosen encoding we consider a brick as visible although all contained values are classified as transparent. However, if we look at Figure 4, we can see that the quantized binary histogram would report the bricks correctly if pre-classification is performed. This is due to the fact, that the quantized binary histogram is more sensitive for largely varying data values. This property can be efficiently exploited if the binary histogram encodes a segmentation information volume. In such a volume, segmented objects are encoded by labels. These labels can differ largely and interpolation is not applicable. Only pre-classification can be performed in this case

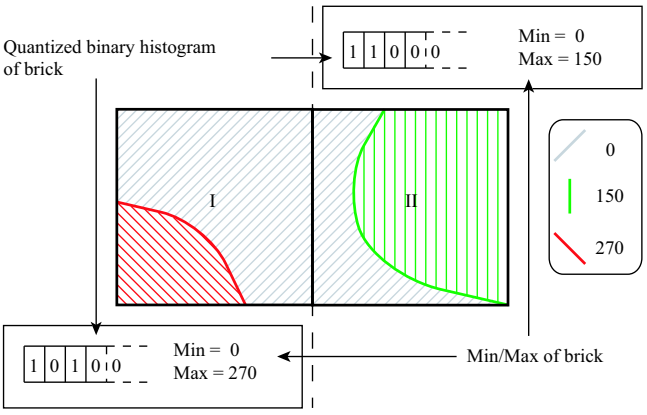


Figure 4: Min-Max encoding granularity issue if pre-classification is performed: If the range of non-transparent values is set from 130 to 150, the min-max encoding would report both bricks as being opaque. The quantized binary histograms would report brick I being transparent and brick II being opaque.

5.2 Granular Resolution Octrees

With the method described in Section 5.1 entirely transparent bricks are determined and therefore unnecessary processing is avoided. We also want to avoid processing of transparent regions within a brick. Therefore, each brick contains a granular resolution octree to enable the determination of transparent regions within a single brick. A min-max octree is one of the best known space subdivision structures to support refined skipping of small transparent areas (Lacroute [6], Wilhelms et al. [21], and Mora et al.[10]). Each brick (32x32x32) contains a 3-level min-max octree, shown in Figure 5a. For each octree level we store the minimum and maximum value as a pair of numbers. For level 0 we have 8 pairs, level 1 needs 8x8 = 64 pairs, and level 2 needs 8x8x8 = 512 pairs. When classification changes the octree is recursively evaluated by a summed area table for all bricks. We store the classification information efficiently by hierarchical compression [5]. Nodes of level 2 are either opaque or transparent. All other nodes have an additional inhomogeneous state. The information whether a node of level 2 is transparent or opaque is stored in one bit. The state of a level 1 node is determined by testing of one byte, which contains all the bits of its children. For level 0 such a hierarchical compression requires to test 8 bytes for a node and 64 bytes for the brick. Due to efficiency reasons we explicitly store the state information of level 0. We have three possible states, thus we need 2 bits for each level 0 node. Thus, we additionally require two byte per brick. Due to this encoding, the octree can be very efficiently traversed.

5.3 Removing of Transparent Regions

We have two structures, a quantized binary histogram and a granular octree, to find the rays-volume intersections up to the resolution of the granular octree (Figure 3c). The bricked geometry of the volume and the octrees within the bricks are converted to a polygonal structure and rendered into a z-buffer [17]. Basically we traverse

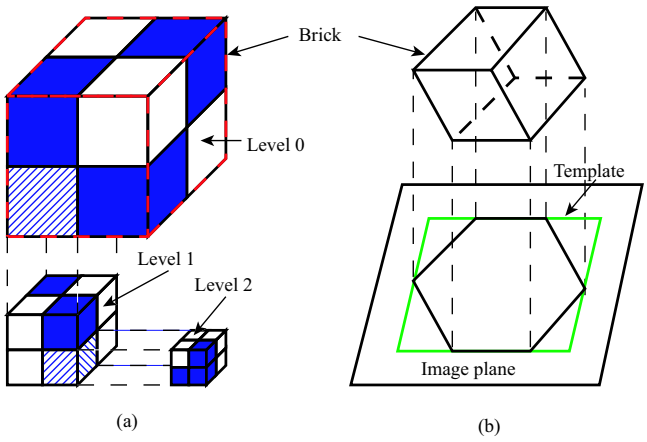


Figure 5: (a) Octree classification scheme for an individual brick. Transparent bricks are white, opaque bricks are blue and inhomogeneous bricks, partially transparent and partially opaque, are blue striped. , (b) Brick projection template.

through the brick list and determine which brick has visible data and needs to be evaluated for rays-volume intersection [3]. The evaluation is performed by quantized binary histograms in case of pre-classification or min-max encodings in case of post-classification. The octree of those bricks is evaluated and the sub-bricks which contain visible data are rendered. This rays-volume intersection computation by rendering requires the *granular* resolution of the octree. With more than three octree levels the number of polygons would exceed the rendering performance of commodity graphics hardware.

Utilizing OpenGL for rendering provides high performance and high accuracy; however, if the approach is used as an integrated module it requires off-screen rendering. This is available in OpenGL by PBuffers. Unfortunately, this feature is not available on every graphics card. Furthermore the rendering requires a huge amount of graphic cards memory. Considering a 1024x1024 image, the needed buffer is already 8 MB. Most of the more advanced medical visualization systems support high-resolution dual-displays. This feature normally utilizes all the available graphics card memory. There is no space left for graphics hardware accelerated off-screen rendering. Due to this reason, we also developed the rendering in software. This can be done very efficiently, if the simple polygonal structure of the bricked octree layout is exploited. Since every brick is of the same structure, one can use template based projection of the octree. Similar work has be done by Srinivasan et al. [18]. The main idea is to project just one brick per viewing direction for each octree level as shown in Figure 5b. This projection is used as a template for all other bricks of the same level. Any other brick of the same level has the same projection footprint and is obtained by translation. The projected footprint actually consists of z-values, since we are interested in the z-buffer footprint of the octree. All possible entry bricks are rendered in a front-to-back order by using the projected z-value template. The resulting z-buffer footprint of the octree is then used to determine the rays-volume intersections. This is as fast as the OpenGL implementation, since the costly projection itself has to be done only for one brick per viewing direction. Furthermore no costly OpenGL glReadPixels() instruction is involved and the resulting z-buffer directly contains the z-components of the ray starting-positions.

5.4 Cell Invisibility Cache: Skipping of Transparent Regions

As the granular resolution octree does not go down to cell level, a cell invisibility cache is used to skip the remaining transparent cells ((Figure 3c → Figure 3d)). The volume-rays intersections estimation by template-based projection of the octree subbricks brings us as close as 4x4x4 samples to the visible data. This is inefficient from a performance point of view. Especially if first-hit-raycasting is performed every non skipped sample has a large impact on the resulting frame-rate. A resolution of 4x4x4 results in a large number of non skipped samples. This is depicted by the red samples shown in Figure 7. All these samples have to be classified in order to determine which cell can be skipped. Depending on the object-sample distance and the zoom factor these cells have to be classified several times. This is shown for a typical re-sampling resolution in Figure 2b. In this case each cell has to be classified eight times. Considering the same example in 3D, the number of redundant cell classifications would be considerably larger. Due to this reason we introduce a refined cell invisibility caching. We extend the volume raycasting pipeline in such way that classification of these invisible cells has to be done only once. The extended pipeline is shown in Figure 6. A Cell Invisibility Cache (CIC) is attached at the beginning of the traditional volume raycasting pipeline. This CIC is initialized in such a way that it reports every cell as visible. In other words every cell has to be classified. Now, if a ray is send down the pipeline, every time a cell is classified as invisible (all its samples have zero opacity contribution) this information is cached in the CIC. A cell can either be invisible or visible, this information can be encoded in just one bit. Once a cell is classified as invisible, the costly classification of a whole cell is exchanged by a binary test. This leads to an enormous performance increase. On the one hand, this is due to the reduced memory access and on the other hand due to the inherent classification and conjunction information of 8 samples. The information stored in the CIC remains valid as long no transfer-function change is performed. The CIC is stored per brick and therefore allows interactive changes of the transfer function. If the transfer function changes only the CICs of the bricks which are affected need to be reset. During the examination of the data, e.g. by changing the viewing direction, the CIC fills up and the performance increases progressively. The same mechanism is also very beneficial for general empty space skipping within the data.

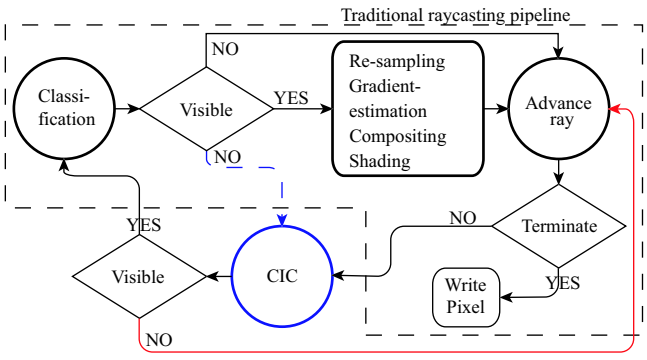


Figure 6: Cell Invisibility Cache (CIC) - Acceleration by caching invisibility information of cells. The acceleration path is emphasized in red.

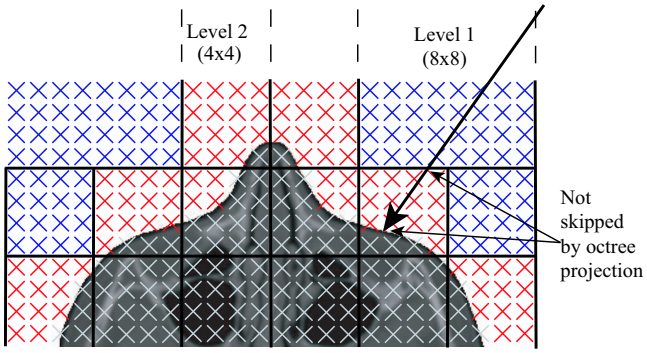


Figure 7: Zoomed in granular octree of one brick. Blue crosses: skipped samples, white crosses: opaque samples, and red crosses are samples that can not be skipped due to the granular resolution of the octree.

6 Performance Results and Memory Consumption

6.1 Gradient Cache

The memory consumption of the gradient cache is very low. The cache size is not related to the volume dimensions. It is related to the brick dimensions. The brick dimensions in our case are 32x32x32, the size of the gradient cache is (*dimension of brick*+1)^3 multiplied by *dimension of gradient* multiplied by *size of gradient component*, which is $(33)^3 \cdot 3 \cdot 4 \approx 421{,}14$ KByte. Additionally we store for each cache entry a valid bit, which adds up to 33^3 Bit $\approx$ 4.39 KByte. This is altogether less than 512 KB. For performance reasons the data shall remain in the level 2 cache. This is not an issue as current commodity CPUs have a level 2 cache size of 1 MB.

Figure 8 shows the effect of per brick gradient caching compared to per cell gradient caching and no gradient caching at all. Per cell gradient caching means that gradients are cached while a ray resamples a cell. For gradient estimation we used the gradient filter proposed by Neumann et al. [13]. This filter produces slightly better quality than the Sobel filter, supports inherent volume filtering and has approximately the same computational cost. Due to the on-the-fly computations, the filtering can be enabled and disabled interactively. The on-the-fly filtering has low computational cost and can be used to increase the quality, when a smaller number of rays are shot to increase the frame-rate during interaction.

For testing we chose an adequate opacity transfer function to enforce translucent rendering. The charts in Figure 8 show different timings for object sample distances from 1.0 to 0.125 for three different zooming factors 0.5, 1.0, and 2.0. In case of zooming factor 1.0 we have one ray per cell, already here per brick gradient caching performs better than per cell gradient caching. This is due to the shared gradients between cells. For a zooming out factor of 0.5 both gradient caching schemes perform equally well. The rays are so far apart that nearly no gradients can be shared. On the other hand for zooming in (2.0), per brick caching performs much better than per cell caching. This is due to the increased number of rays per cell. As more rays process the same cell, the more beneficial the per brick caching becomes. Per brick gradient caching compared to no caching shows already with a zoom factor of 2.0 and an object sample distance of 0.5 an impressive speedup of approximately 3.0. The speedup favorably scales with the zoom factor. Figure 9 shows an example rendering of the Visible Male with a high proportion of transparency. Our caching scheme compared to no caching shows a speedup factor of ≈ 2.2.

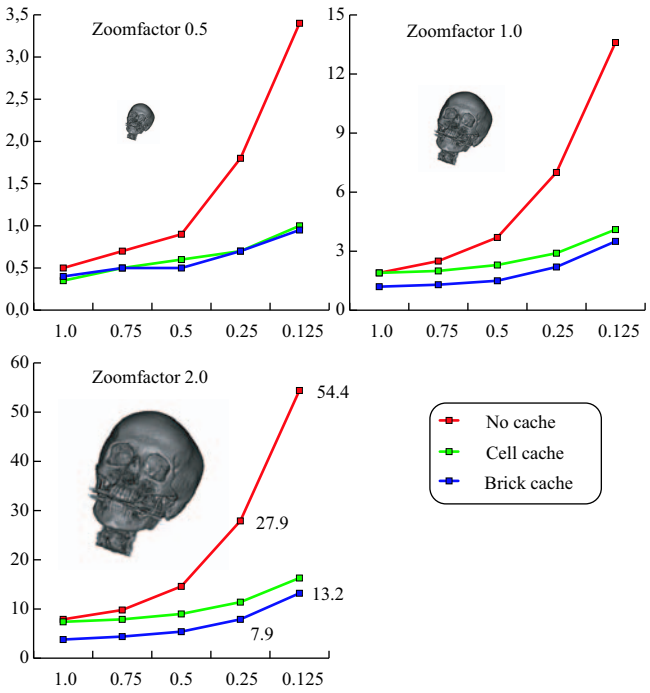


Figure 8: Comparison between no gradient caching, per cell gradient caching, and per brick gradient caching. Timings are given in seconds. The tested object sample distances are 0.125, 0.25, 0.5, 0.75, and 1.0. Data: UNC head, 256x256x224, 12 bit. Intensity range [0,1136] is mapped to 0.0 opacity and range [1136,4095] to a linear opacity ramp between 0.0 and 1.0. System specification: CPU - Intel®Pentium®M 1.6 GHz, Cache - 1 MB Level2, RAM - 1 GB, GPU - GForce4 4200 Go (32MB).

6.2 Acceleration Structures for Skipping Empty Regions

The additional memory usage of all three acceleration structures, i.e., quantized binary histogram, granular resolution octree, and the cell invisibility cache, is rather low. Considering the size of the volume as 100%, they increase the size by approximately 10%. We use bricks of size 32x32x32 storing 2 bytes for each sample, which is a total of 65536 bytes. Additionally for each brick we store: Quantized binary histogram $\rightarrow$ 4 byte, Min-max information $\rightarrow$ (512+64+8+1) * 4 = 2340 byte, Octree classification information $\rightarrow$ (64 + 2) = 66 byte, and Cell Invisibility Cache $\rightarrow$ 32^3 bit /8 = 4096 byte. In total the storage increase is $((4+2340+66+4096)/65536) \cdot 100 \approx 9.9\%$.

Figure 10 shows the effect of our hybrid removal and skipping technique of transparent regions and shows the corresponding rendering output. For benchmarking we used a commodity notebook equipped with an Intel®Pentium®M 1.6 GHz CPU, 1 MB Level2 cache, 1 GB RAM, and a GForce4 4200 Go (32MB). The graphics card capabilities are only used to display the final image. We tested different data sets. A rather small data set, the UNC head is used to be able to compare our speed to the approach of Mora et al. [10]. This approach is slightly faster than the UltraVis system [5]. They are both based on a spread memory layout and use pre-computed gradients. This leads to an inefficient memory usage and so they are restricted to rather small data. Mora's total render time is approximately a factor of two faster than our approach. However, Mora's approach uses pre-computed gradients, does pre-shading, and its template based interpolation scheme limits the zooming to a zooming-factor of four. In contrast to that we chose to sacrifice

some performance for increased flexibility, high quality, and a significantly lower memory usage. This enables us to render large data, used in clinical routine, on commodity hardware. We tested three different large typical medical data sets. The results show that our acceleration techniques typically achieve render-times of about 2 fps even for these large data sets. Figure 10, fourth column, shows the total render time achieved by brick based transparent region removal. In the fifth column we additionally apply the granular octree projection. And finally in the sixth column we enabled the Cell Invisibilty Cache to see the overall total render time achieved by the combined effect of all three acceleration structures.

7 Discussion and Conclusion

We presented a volume raycasting approach which provides high-quality images in real-time for large data on standard commodity computers without advanced graphics hardware. For large medical data such as computed tomographic (CT) angiography run-offs (512x512x1202) we achieve rendering times up to 2.5 fps on a commodity notebook. This shows that real-time rendering of such large data on commodity notebooks is within reach. Our method can be straightforwardly adapted to other modalities such as MR. Furthermore, our approach can utilize symmetric multiprocessing systems as processing is performed brick-wise. It scales well and achieves a speedup factor of approximately 2.0 on a dual CPU machine. This is very beneficial if a large amount of data has to be processed. Although we avoided costly pre-computing and compute each part of the volume raycasting pipeline on-the-fly, we achieved performance in the same range as approaches which heavily rely on the memory bandwidth such as Mora et al. [10]. Our refined caching scheme for gradient estimation in conjunction with hybrid skipping and removal of transparent regions enables us to achieve high quality while maintaining high performance. Due to the efficient memory consumption of our acceleration structures (quantized binary histogram + granular resolution octree + Cell Invisibility Cache) and the bricked volume layout we are able to handle very large data. All acceleration structures require only an extra storage of approximately 10%. Data sizes up to 2GB are possible, which is a limitation imposed by the virtual address space of current consumer operating systems.

8 Future Work

In the future we want to support out-of-core rendering to be prepared for the next generation of data sizes. First commodity prototype scanners already deliver data in the range of 1024x1024x2048, which is about 4GB of data and quite challenging to handle. Additionally we want to incorporate brick-based compression to achieve interactive rendering times for these large data. Furthermore we want to support perspective rendering. The changes, which have to be made, are basically to split the brick rendering lists such that the correct processing order is ensured. Additionally we have to render perspectively distorted bricks to be able to find the exact entry points of the rays. Other than that all the presented accelerating techniques should work as well as for orthogonal projection.

Additional material can be found at:
http://www.cg.tuwien.ac.at/research/vis/adapt/2004_meas

9 Acknowledgements

The work presented in this publication has been funded by the ADAPT project (FFF-804544). ADAPT is supported by *Tiani Medgraph*, Vienna (http://www.tiani.com), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See http://www.cg.tuwien.ac.at/research/vis/adapt for further information on this project. The used data sets are courtesy of Univ.-Klinik Innsbruck, AKH Vienna, Univ.-Klinikum Freiburg, and NLM.

References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the Symposium on Volume Visualization*, pages 91–98, 1994.

[2] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers and Graphics*, 28(5), 2004. To appear.

[3] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization*, pages 53–60, 2002.

[4] J. Huang, K. Müller, N. Shareef, and R. Crawfis. Fast splats: optimized splatting on rectilinear grids. In *Proceedings of IEEE Visualization*, pages 219–226, 2000.

[5] G. Knittel. The Ultravis system. In *Proceedings of the IEEE Symposium on Volume visualization*, pages 71–79, 2000.

[6] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University, Computer Systems Laboratory, 1995.

[7] A. Law and R. Yagel. Multi-frame thrashless ray casting with advancing ray-front. In *Proceedings of Graphics Interfaces*, pages 70–77, 1996.

[8] M. Meissner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Strasser, M. Doggett, P. Forthmann, and R. Proksa. Vizard II, a reconfigurable interactive volume rendering system. In *Proceedings of the Eurographics Workshop on Graphics Hardware*, pages 137–146, 2002.

[9] T. Möller, R. Machiraju, K. Müller, and R. Yagel. A comparision of normal estimation schemes. In *Proceedings of IEEE Visualization*, pages 19–26, 1997.

[10] B. Mora, J. Jessel, and R. Caubet. A new object order ray-casting algorithm. In *Proceedings of IEEE Visualization*, pages 107–113, 2002.

[11] K. Müller, T. Möller, and R. Crawfis. Splatting without the blur. In *Proceedings of IEEE Visualization*, pages 363–370, 1999.

[12] K. Müller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[13] L. Neumann, B. Csebfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4D linear regression. In *Proceedings of Eurographics*, pages 351–358, 2000.

[14] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization*, pages 233–238, 1998.

[15] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Proceedings of SIGGRAPH*, pages 251–260, 1999.

[16] G. D. Rubin, A. J. Schmidt, L. J. Logan, and M. C. Sofilos. Multidetector row ct angiography of lower extremity arterial inflow and runoff: Initial experience. *Radiology*, 221:146–158, 2001.

[17] L. Sobierajski and R. Avila. A hardware acceleration method for volumetric ray tracing. In *Proceedings of IEEE Visualization*, pages 27–34, 1995.

[18] R. Srinivasan, S. Fang, and S. Huang. Volume rendering by template-based octree projection. In *Proceedings of the Eurographics Workshop on Visualization in Scientific Computing*, pages 155–163, 1997.

[19] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH*, pages 169–177, 1998.

[20] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, 1990.

[21] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

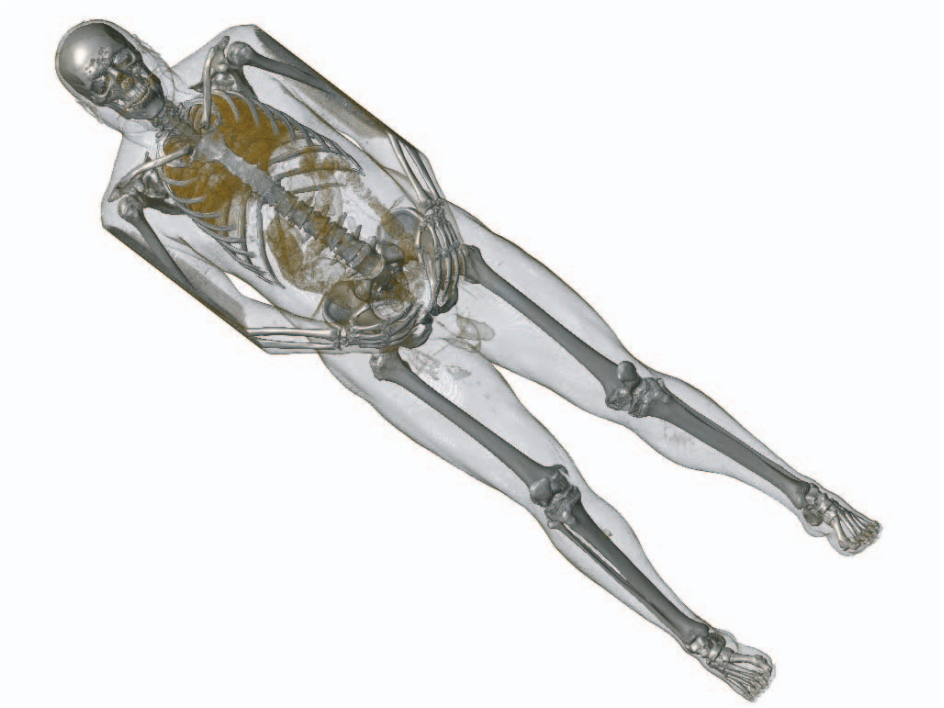
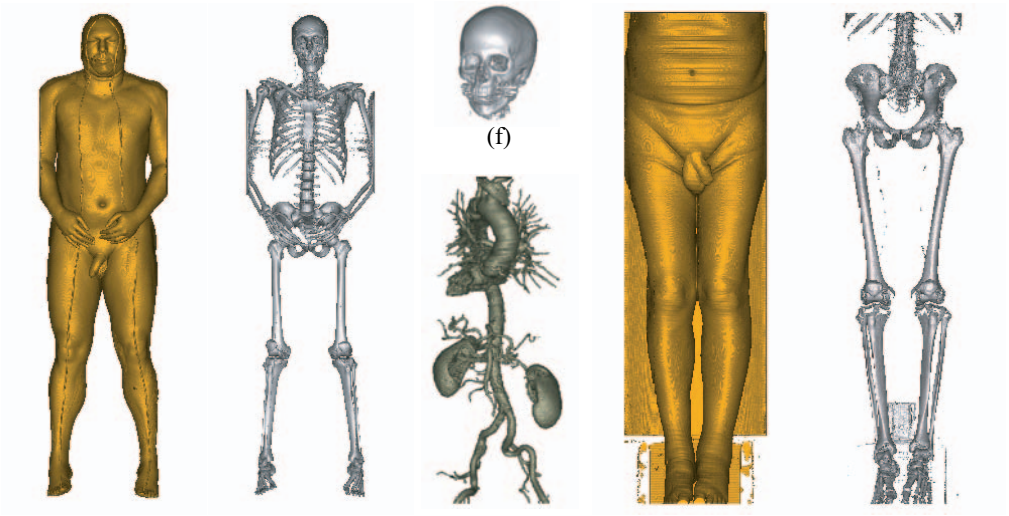


Figure 9: Data: Visible Male (587x341x1878). System: (Notebook) Intel®Pentium®M 1.6 GHz. Image size: 1024x768. Object sample distance: 0.25. Render timings: no gradient caching → 21.1 sec, with full gradient caching → 9.6 sec.



	Name	Dimensions	Size	TBR	TBR + OPR	TBR + OPR + CIC
(a)	Visible Male	587x341x1878	0.70 GB	0.61 sec	0.46 sec	0.40 sec
(b)	Visible Male	587x341x1878	0.70 GB	0.68 sec	0.53 sec	0.45 sec
(c)	Run-off	512x512x1112	0.54 GB	1.16 sec	0.93 sec	0.61 sec
(d)	CTA run-off	512x512x1202	0.59 GB	0.86 sec	0.70 sec	0.64 sec
(e)	CTA run-off	512x512x1202	0.59 GB	0.69 sec	0.46 sec	0.37 sec
(f)	UNC head	256x256x256	0.03 GB	0.71 sec	0.26 sec	0.18 sec

Figure 10: Performance results for different data sizes, which are used in daily clinical routine. Image size: 512x512, Sample rate: 0.5, and Hardware: CPU - Intel®Pentium®M 1.6 GHz, Cache - 1 MB Level2, RAM - 1 GB, GPU - GForce4 4200 Go (32MB). TBR: brick based transparent region removal. OPR: octree projection based transparent region removal. CIC: cell based transparent region skipping.