The Visual Computer manuscript No.
(will be inserted by the editor)

Enrico Gobbetti · Fabio Marton · José Antonio Iglesias Guitián

A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets

Abstract We present an adaptive out-of-core technique for rendering massive scalar volumes employing single pass GPU raycasting. The method is based on the decomposition of a volumetric dataset into small cubical bricks, which are then organized into an octree structure maintained out-of-core. The octree contains the original data at the leaves, and a filtered representation of children at inner nodes. At runtime an adaptive loader, executing on the CPU, updates a view- and transfer function-dependent working set of bricks maintained on GPU memory by asynchronously fetching data from the out-of-core octree representation. At each frame, a compact indexing structure, which spatially organizes the current working set into an octree hierarchy, is encoded in a small texture. This data structure is then exploited by an efficient stackless raycasting algorithm, which computes the volume rendering integral by visiting non-empty bricks in front-to-back order and adapting sampling density to brick resolution. Block visibility information is fed back to the loader to avoid refinement and data loading of occluded zones. The resulting method is able to interactively explore multi-giga-voxel datasets on a desktop PC.

1 Introduction

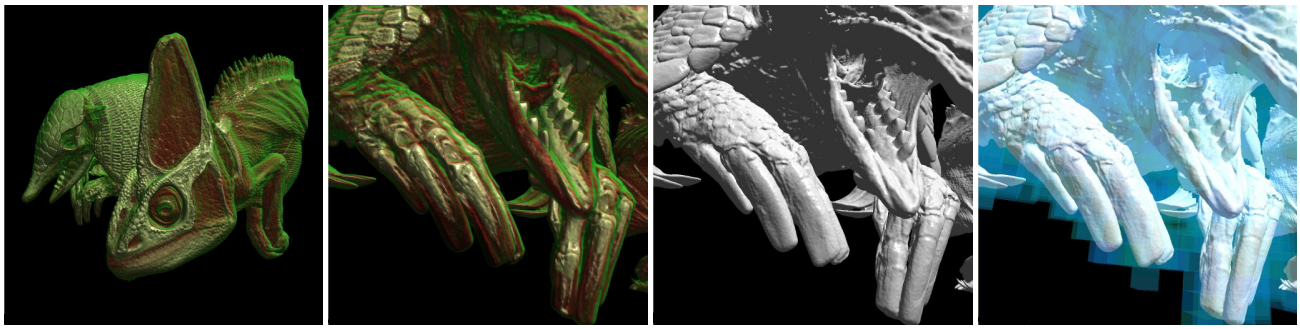
The ability to interactively render rectilinear scalar volumes containing billions of samples on desktop PCs is of primary importance for a number of applications, which include medical visualization and numerical simulation results analysis.

Many sophisticated techniques for real-time volume rendering have been proposed in the past, taking advantage of CPU acceleration techniques, GPU acceleration using texture mapping, or special purpose hardware. In the last few years, improvements in programmability and performance of GPUs have made GPU solutions the main option for real-time rendering on desktop platforms [4]. Current high quality solutions, based on ray-casters fully executed by GPU fragment programs, have demonstrated the ability to deliver real-time frame rates for moderate-size data, but they typically require the entire dataset to be contained in GPU memory. Rendering of large datasets can be achieved through compression, multiresolution schemes, and out-of-core techniques. Current solutions, however, are not fully adaptive and, with the exception of flat blocking schemes [15], are not typically implemented within a single-pass raycasting framework, with increased frame buffer bandwidth demands and/or decreased precision and flexibility in the computation of volume integrals (see section 2).

In order to remove such limitations, we present an adaptive out-of-core technique for rendering massive scalar datasets within a single-pass GPU raycasting framework. The method exploits an adaptive loader executing on the CPU for updating a working set of bricks maintained on GPU memory by asynchronously fetching data from an out-of-core volume octree representation. At each frame, a compact indexing structure, which spatially organizes the current working set into an octree hierarchy, is encoded in a small texture. This data structure is then exploited by an efficient raycasting algorithm, which computes the volume rendering integral by enumerating non-empty bricks in front-to-back order and adapting sampling density to brick resolution. The algorithm is a streamlined octree extension of an efficient stackless ray traversal method for kd-trees [7, 16], which reduces costly texture memory accesses by computing neighbor information on-the-fly. In order to further optimize memory and bandwidth efficiency, the method also exploits feedback from the renderer to avoid refinement and data loading of occluded zones.

Although not all the techniques presented here are novel in themselves, their elaboration and combination in a single system is not trivial and represents a substantial enhancement to the state of the art. The resulting method is extensible, fully adaptive, and able to interactively explore multi-giga-voxel datasets on a desktop PC (see figure 1).

CRS4 Visual Computing Group
Sardegna Ricerche Edificio 1, C.P. 25
09010 Pula, Italy
E-mail: gobbetti@crs4.it marton@crs4.it jalley@crs4.it
www: http://www.crs4.it/vic/



(a) Volume rendering overview (b) Volume rendering detail (c) Isosurface detail (d) Isosurface detail's octree structure

Fig. 1 Interactive exploration of multi-gigabyte CT datasets. This two-Gvoxels 16bit dataset is interactively explored on a desktop PC with a NVIDIA 8800 Ultra graphics board using a 1024×1024 window size. Transfer functions and isovalues can be interactively changed during navigation. The volume rendered images have a full Phong model with specular reflections and view-dependent transparency.

2 Related Work

Volume visualization is a well researched subject. In the context of this paper, we limit our discussion to the approaches most closely related to ours. We refer the reader to the recent book of Engel et al. [4] for a survey on GPU volume rendering.

The out-of-core organization of massive volumetric data into a volume octree is a classic one. Lamar et al.[12] proposed a multiresolution sampling of octree tile blocks according to view-dependent criteria. Boada et al. [2] proposed a coarse octree built upon uniform sub-blocks of the volume, and used, instead, data dependent measures to select block resolution. In such systems, as in most previous GPU accelerated multiresolution schemes, rendering of multiresolution volumes on graphics hardware is accomplished by separate rendering of blocks and frame buffer composition. For instance, Guthe et al. [6] exploits a decomposition into wavelet compressed blocks, uses block resolution to determine inter-slice distance, and introduces methods for empty space skipping and early ray termination. Li et al.[14] propose to accelerate slice-based volume rendering by skipping empty blocks and exploiting an opacity map for occlusion culling. Slice-based implementations are, however, rasterization limited and hard to optimize from an algorithmic point of view. Furthermore, when applying a perspective projection the integration step size will vary along viewing rays when using planar proxy geometries, leading to visible artifacts. In order to solve some of these problems, other authors [8,9] separately render blocks using volumetric raycasting on the GPU and devise propagation methods to sort cells into layers for front-to-back rendering, therefore reducing framebuffer demands. The separate rendering of blocks, however, is prone to rendering artifacts at block boundaries, and does not easily allow an implementation of optical models with viewing rays changing direction, as it does occur in refracting volumes, or with non-local effects, as it does occur in global illumination. Our method, instead, is based on a full-volume GPU ray-casting approach [11,17], with a fragment shader that performs the entire volume traversal in a single pass [19]. Such an approach, made possible by modern programmable GPUs, is more general, but, until very recently, has been limited to moderate size volumes that fit entirely into texture memory. In this context, the issue of large volumes has been typically addressed by compressing data using adaptive texturing schemes to fit entire datasets into GPU memory in compressed form [20], or by using flat multiresolution blocking methods [15]. In the first approach, data is stored at various resolution levels using adaptive texture maps [10] to reduce storage needs, but sampling density is not adapted as the ray passes through different blocks of data. The flat multiresolution blocking technique, instead, represents a volume as a fixed grid of blocks and varies the resolution of each block to achieve adaptivity. The disadvantage of this fine-grained approach in comparison with our hierarchical approach is that the number of blocks is constant and the method remains performing only if individual blocks are within a small range of sizes. Our method relies instead on the ability to rapidly traverse a octree structure and is based on the stackless ray traversal method for kd-trees [7] recently extended to GPUs for surface rendering [16]. Our method exploits the regular structure of octrees to reduce costly texture memory accesses by computing bounding boxes on-the-fly. In addition, our algorithm takes advantage of occlusion queries to avoid loading occluded data. Other authors have proposed using depth information to optimize full-volume GPU raycasters, but, in general, the focus is on implementing early-ray termination in multi-pass methods by exploiting early z-tests features [11, 17]. Our scheme exploits spatial and temporal coherence to schedule queries in an order that strives to reduce end-to-end latency, similarly to what is done for recent surface renderers [5,1]. The central idea of these methods is to issue multiple queries for independent scene parts and to avoid repeated visibility tests of interior nodes by exploiting the coherence of visibility classification. In our case, the partitioning occurs in image space, rather than in object space.

3 Method overview

Since massive volumetric datasets cannot be interactively rendered by brute force methods, applications must ideally employ adaptive rendering techniques whose runtime and memory footprint is, as much as possible, proportional to the number of image pixels, not to the total model complexity.

For efficiently updating the rendering working-set, these methods require the integration of level-of-detail and visibility culling techniques. Out-of-core data management is used for filtering out as efficiently as possible the data that is not contributing to a particular image.

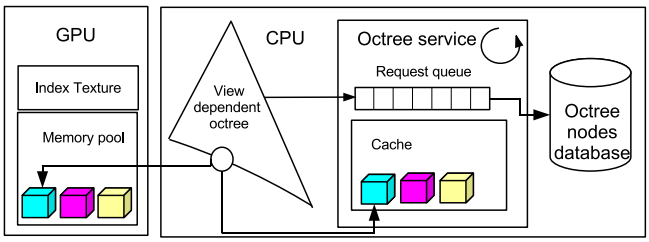


Fig. 2 Method overview. At runtime, an adaptive loader, executing on the CPU, updates a view- and transfer function-dependent working set of bricks maintained on GPU memory by asynchronously fetching data from an out-of-core coarse-grained octree representation. A compact indexing structure that spatially organizes the current working set is exploited by an efficient stackless GPU raycaster for image generation.

In our approach, we separate the creation and maintainance of the rendering working set, which is performed on the CPU, from the actual rendering, which is fully performed on the GPU based on an efficient encoding of the current working set representation (see figure 2). In order to maximize CPU and bandwidth efficiency, we employ a coarse-grained volume decomposition, which allows us to amortize decision costs over a large number of rendered voxels and to efficiently update the GPU representation with few calls.

The original volumetric model is decomposed into small cubical bricks, which are then organized into a coarse octree structure maintained out-of-core. The octree contains the original data at the leaves, and a filtered representation of children at inner nodes. Each node also stores the range of values, as well as, optionally, precomputed gradients. In order to efficiently support runtime operations that require access to neighboring voxels, such as linear interpolation or gradient computations, blocks are made self-contained by replicating neighboring samples. One layer is replicated for linear interpolation support, while two layers are replicated for additionally using central differences to compute gradients at rendering time. At runtime, an adaptive loader updates a view- and transfer function-dependent working set of bricks incrementally maintained on CPU and GPU memory by asynchronously fetching data from the out-of-core octree. The working set is maintained by an adaptive refinement method guided by suitably computed node priorities (see section 4). At each frame, a compact indexing structure, which spatially organizes the current working set into an octree hierarchy, is encoded in a small texture. This structure is not a multiresolution data representation, but simply spatially organizes the leaves of the current view-dependent representation into an octree with neighbor pointers. The inner nodes of this structure simply contain pointers to children, and only the leaves refer to volume data nodes stored in the memory pool. The spatial index structure is exploited by an efficient stackless GPU raycaster, which computes the volume rendering integral by enumerating non-empty bricks in front-to-back order, adapting sampling density to brick resolution, and stopping as soon as the accumulated opacity exceeds a certain threshold, updating both the frame- and depth-buffer (see section 5).

Using an occlusion query mechanism designed to reduce GPU stalls, feedback from the renderer is exploited by the loader to avoid refinement and data loading of occluded zones (see section 6).

4 Generating a view- and transfer-function dependent working set

At each frame, an incremental refinement procedure constructs the current view-dependent working set by refining a sorted set of visible non-empty nodes initialized with the octree root. In the most basic case, the set is sorted by decreasing projected screen-space size of voxels, but we will see in section 6 how visibility information can also be incorporated in the process. Empty nodes are terminal ones in the refinement process, as well as nodes which fall outside of the view frustum. As in [21], for isosurface rendering, a node is considered non-empty if the isovalue is within the range spanned by the minimum and maximum value of the cell. In the case of volume rendering, as in [18], summed-area tables of the transfer function opacity are used to determine if the block is empty by taking the difference of the table entries for the minimum and maximum block values. The refinement procedure stops when all nodes are considered adequately refined, no data is currently available in-core to perform a refinement, or no more space is available in the GPU cache to contain a further subdivision. In order to hide out-of-core data access latency, all data access requests are performed asynchronously by a separate thread, and refinement continues only if data is immediately available.

At the end of the refinement process, all nodes in the current working set are present both in the CPU and GPU cache, the CPU cache being used as a larger level-2 cache that uses system memory to avoid disk accesses for recently used blocks. Both memory pools are managed using a LRU policy. The GPU texture cache is devised as a large preallocated 3D texture managed as a pool of blocks, or two of them when using precomputed gradients. In our current implementation, the value texture is configured as a 16bit single channel texture, while the gradient texture is a RGBA8

texture with normalized gradients in the RGB components and gradient norm in the A component. During incremental refinement, the GPU texture cache is incrementally updated with `glTexSubImage` calls to move data. Due to temporal and spatial coherence, the number of updated nodes per frame is generally small.

5 GPU rendering

The incremental refinement procedure defines a cut of the octree hierarchy that is considered adequate for the current frame and stores all data blocks associated to non-empty leaves in GPU memory. In order to render an image with a single-pass GPU raycaster, the fragment shader must be able to efficiently enumerate in front-to-back order for each fragment all the blocks pierced by the associated view ray. We reach this goal by using an octree with neighbor structure to spatially index the current leaf blocks, and using this structure to accelerate ray traversal.

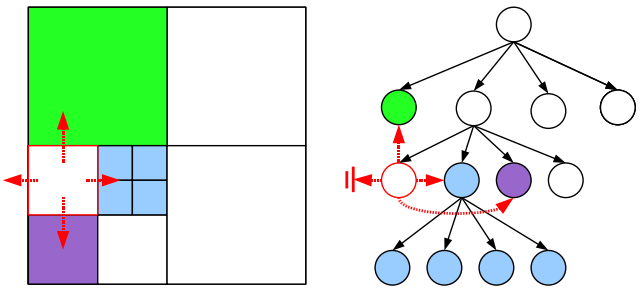


Fig. 3 Octree with neighbor pointers. Neighbor pointers link each leaf node of the octree via its six faces directly to the corresponding adjacent node of that face, or to the smallest node enclosing all adjacent nodes if there are multiple ones.

5.1 Spatial index construction

The octree with neighbors structure augments a branch-on-need octree with links, so that a direct traversal to adjacent nodes is possible (see figure 3). In this structure, neighbor pointers directly link each leaf node of the octree via its six faces to the corresponding adjacent node of that face, or to the smallest node enclosing all adjacent nodes if there are multiple ones. We create such a structure on-the-fly at each frame directly from the view-dependent octree, and encode it into a 3D texture that acts as a spatial index.

The layout of the spatial index texture is designed to encode the minimum amount of data required for octree traversal (see figure 4). Similarly to Octree Textures on GPU [13], we use a 8 bit *RGBA* texture encoding information in the *RGB* component pointer information, and various kinds of tags in the *A* components. With this encoding, we can potentially address $16M$ nodes or data blocks, which is several orders of magnitude larger than what is needed.

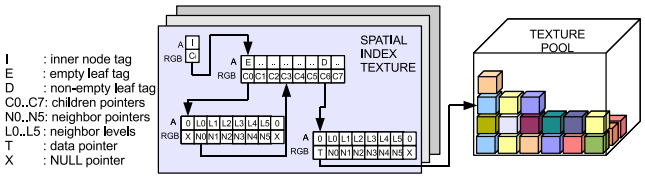


Fig. 4 Spatial index and memory pool textures. Each octree node is encoded in 8 of consecutive texels arranged in the x direction.

The octree structure is encoded using a tagged pointer per node. The *A* component of the tagged pointer determines the node kind, and it is $A = 1.0$ for inner nodes, $A = 0.5$ for for data nodes, and $A = 0.0$ for empty nodes. Inner nodes used the *RGB* component of the tagged pointer to point to the first of 8 children arranged consecutively in the x direction. Leaf nodes use the *RGB* component of the tagged pointer to point to leaf information, which consists in a data pointer (if the node is not empty), and 6 consecutive texels storing pointers to neighbors. The last leaf data texel always contains a NULL pointer, used in our traversal code to simplify the handling of rays that do not exit from the current box and thus should not continue to neighbors. Neighbor pointers use their *A* value to encode the octree level of the neighbor, which can be the same as the level of current node or coarser. The level information is all that is required in our traversal algorithm to rapidly compute the bounding box information of neighbors during traversal.

The on-the-fly construction of the index texture is fast, since our structure is coarse-grained and the view-dependent tree is composed of only a few thousands leafs. Once the structure is constructed in a memory area, the GPU texture is updated using a `glTexSubImage` call, and the fragment shader implementing volume raycasting is activated by rendering a quad.

5.2 Spatial index traversal

The spatial index structure is exploited by an efficient raycasting algorithm, which computes the volume rendering integral by enumerating non-empty bricks in front-to-back order and adapting sampling density to brick resolution. The traversal algorithm is a streamlined octree extension of an efficient stackless ray traversal method for kd-trees [7, 16], which reduces costly texture memory accesses by computing neighbor information on-the-fly (see figure 5). In our approach, children and neighbor bounding boxes are implicitly computed on-the-fly by the shader without any additional access to texture memory by exploiting the regular structure of the octree. The basic concept behind the stackless traversal is to start by performing a down traversal of the octree for locating the leaf node that contains the current sampling position, which, at the start, is the position at which the ray enters the volume. Then, the leaf node is processed by accumulating color and opacity by stepping through the associated brick if it contains data, or simply skipping the node if it is empty. If the ray does not terminate because maximum

opacity is reached, the algorithm determines the face and the intersection point through which the ray exits the node. Then traversal continues by following the neighbor pointer of this face to the adjacent node, eventually performing a down traversal to locate the leaf node that contains the exit point, which is now the entry point of the new leaf node. This approach has the important advantage of not requiring a stack to remember nodes that still need to be visited, since the state of the ray only consists of its current node and its entry point.

```
fragment.color=float4(0,0,0,0);
fragment.depth=FAR;
// Start at octree root
node_ptr = float3(0,0,0); octree_level=0;
box_min=float3(0,0,0); box_dim=float3(1,1,1);
while (!is_null(node_ptr) and color.a<1) {
  // Find leaf containing current sampling point
  P = ray.start+ray.dir*t_min;
  node = tex3d(spatial_index, node_ptr);
  while (is_inner(node.w)) {
    box_dim/=2; box_mid=box_min+box_dim;
    s=step(P,box_mid); box_min+=s*box_dim;
    child_offset=dot3(s,float3(1,2,4))*texel_sz;
    node_ptr=node.xyz+float4(child_offset,0,0,0);
    node=tex3d(spatial_index, node_ptr);
    ++octree_level;
  }
  // Clip ray to box and find exit face
  (box_t_max, exit_face_idx, exit_dir) =
     box_clip(ray, t_min, t_max, box_min, box_dim);
  // If non-empty block, access data and accumulate
  if (!is_empty(node.w)) {
    data_ptr=tex3d(spatial_index, node.xyz);
    (fragment.color, fragment.depth) =
      accumulate(fragment.color,
                 ray, t_min, box_t_max,
                 data_ptr, box_min, box_max);
  }
  // If ray exits from current block, move to neighbor
  neighbor_offset=float3(1+exit_face_idx,0,0)*texel_sz;
  neighbor=tex3d(spatial_index, node.xyz+neighbor_offset);
  node_ptr=neighbor.xyz;
  octree_level=neighbor.w;
  box_dim=exp2(-octree_level);
  box_min=trunc(box_min/box_dim)*box_dim;
  t_min=box_t_max;
}
```

Fig. 5 Stackless octree traversal on the GPU. The code minimizes memory accesses by computing visited boxes on the fly.

In order to implement this approach, the information a node should provide in addition to tags and pointers consists in its bounding box, which is used for locating points during a down traversal, and exit ray positions during neighbor traversal. All our computations are done in texture coordinates, and we assume that the octree subdivides the unit cube. During all out traversal steps, we maintain a current box, initialized with the unit cube, as well the current octree level, initialized at 0.

A down traversal step from a node to the child containing a point P can be efficiently implemented by box subdivision, using a step function that compares P with the center of the current box. For each component of P, this function returns 0 if that component is less than the center's value, and 1 otherwise. The child box coordinates are thus obtained by translating the parent box origin by an amount $step_i \times child\_box\_dim_i$, where i is x, y or z. The values returned by the step function are also combined to access the proper children pointer inside the spatial index texture, which is stored at an offset $step_x * 1 + step_y * 2 + step_z * 4$ from the current pointed node. Each time the box is subdivided, the octree level is incremented by one.

Computing the bounding box of a neighbour relies on octree level tracking. When moving to a neighbor, the origin of the box is in a first step simply shifted in the direction of the box face from which the ray is exiting by translating it by $dir * box\_dim$, where the exiting face direction dir is either $\pm x$ or $\pm y$ or $\pm z$. This operation is sufficient to compute the neighbor box if it is at the same octree level. If, instead, the neighbor level is at a coarser level than the current node, the shifted box must be coarsened, an operatation that can performed effciently in closed form by first updating the box dimension to $box\_dim = 2^{-neighbor\_level}$ and then snapping the box origin to the $neighbor\_level$ grid by computing $box\_min = \lfloor box\_min / box\_dim \rfloor \cdot box\_dim$.

5.3 Adaptive sampling

The stackless traversal technique allows the fragment shader to enumerate all non-empty leaves pierced by a ray in front-to-back order. Each time the ray enters a leaf with data, the ray chooses a step size matching the local voxel density and accumulates color and opacity information depending on the active rendering mode. Entry and exit points within the block are determined during the octree traversal process. For isosurface rendering, we simply look for intervals that bracket the selected isovalue, while for semitransparent direct volume rendering, we have currently implemented a Phong illumination model with boundary enhancement and view-dependent transparency [3]. Stepping by a discrete number of intervals, which are directly associated to octree levels, enables the use of a compact precomputed 2D transfer function, using the octree level for the second dimension, where the transfer function opacity and color weighting are adjusted accordingly. Both the isovalue and the transfer function can be modified interactively. When the ray exits the block, the current opacity is checked, and, if it exceeds a certain threshold (0.99 in our tests), the ray terminates and the depth of the fragment is updated.

6 Incorporating visibility information

With the described techniques, we are able to perform fully adaptive GPU ray casting: as a matter of fact the structure provides support for empty space skipping, adaptive sampling and occlusion culling through early ray termination. Occlusion culling during ray accumulation is performed by early ray termination. This approach optimizes computations but is not optimal in terms of data management, since occluded areas are discovered only at rasterization time, after

the data is already part of the working set. In order to avoid wasting GPU memory resources and exploit bandwidth, we have incorporated a feedback mechanism in the system, that allows us to exploit visibility information gathered during rendering in the loader.

The basic principle of the method is to update at each frame the visibility status of the nodes in the graph, and, during the refinement cycle, only refine nodes that were marked as visible during the previous frame. Under this approach, the available GPU texture slots will be used mainly to refine nodes present in the visible part of the model, and load requests will not be posted for invisible ones. In order to gather node visibility information, we assume that the fragment shader write the depth of the last visited sample into the depth buffer. By issuing an occlusion query for the bounding box of non-empty leaves after volume rendering has finished, visibility information can be gathered by exploiting the rasterization hardware. If the occlusion query results indicates that the number of visible pixels is below a visibility threshold (4 pixels in our benchmarks), the node is marked not visible. The visibility information of leaves can then be propagated up to the root by considering an inner node visible only if at least one of its children is visible.

A straightforward implementation of this method, which would issue queries for all rendered nodes after the volume rendering call, is possible, but would be inefficient. It should be noted that, although queries are processed quickly using the rasterization power of the GPU, their results are not available immediately, due to the delay between issuing a query and its actual processing by the graphics pipeline, which would only occur when the rasterizers have finished with raycasting. For this reason, we exploit spatial and temporal coherence to schedule queries in a way that reduces end-to-end latency. Since we perform full-volume raycasting, the rendering procedure is separated into independent parts using a screen space subdivision. Given a budget of visibility queries per frame, the screen is recursively partitioned into tiles with the purpose of separating blocks for which visibility will be queried from all others (see figure 6). The partitioning uses a 2D binary space subdivision of the list of 2D rectangles that bound the block projection. Computing this set adds little overhead, since node bounding boxes are already projected onto the screen when computing node priorities. At each subdivision step, a split plane along one of the axes is placed at the position that separates the set into two equally sized subsets, defining two screen tiles. One of the two tiles is selected as containing the occlusion queried subset of octree nodes. The subdivision process continues only on that tile, until its size is below a certain threshold. At each frame, we take different decisions when selecting the half-spaces to ensure that we cover the entire set of octree nodes after a minimum number of frames. At the end of this process, we obtain a list of tiles partitioning the original viewport, as well as a list of blocks whose projection is entirely contained within one of the tiles, placed at the beginning of the list. The rendering process continues by instructing the graphics pipeline to raycast each of the tiles, and to perform occlusion queries for the selected subset of octree nodes just after issuing the rendering command of the first one (see figure 6). This approach reduces end-to-end latency because of the interleaving of the processing of occlusion queries in the first tile with the rasterization of the other tiles. Visibility information is incorporated in the system with a delay of $N+1$ frames, where N is the number of partitions required to cover all blocks. In practice, this number is very small, 2 to 4 in our tests, since we use a coarse-grained octree subdivision and the working set if made up of few hundreds to few thousands nodes.

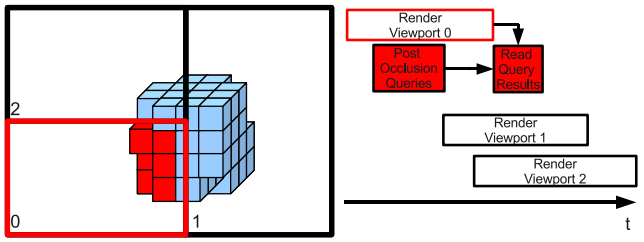


Fig. 6 Screen space subdivision. The screen is recursively partitioned into tile for the purpose of separating blocks for which visibility will be queried from other blocks, thus enabling interleaving of query processing with rasterization.

7 Implementation and results

An experimental software library and a rendering application supporting the technique have been implemented on Linux using C++ with OpenGL, and Cg 2.0. The octree is stored in an out-of-core structure, based on Berkeley DB, and data is losslessly compressed with the LZO compression library.

We have tested our system with a variety of high resolution models. In this paper, we discuss the results obtained with the inspection of a large volumetric model containing two high resolution X-Ray CT datasets of biological specimens [1]. The overall volume has a resolution of $2048 \times 1024 \times 1080$ with 16 bit/sample, and has been embedded in a 2048^3 cubical grid.

All tests have been performed on a Linux PC with a dual 2.4GHz CPU, 4GB RAM, a GeForce8800 Ultra graphics board and SATA2 disks storing the out-of-core models. The construction of the octree with precomputed gradients from the source datasets was performed using a granularity of 32^3 for octree bricks with 1 layer overlap. The preprocessor was instructed to filter out data by discarding all blocks with a value lower than 6400, in order to discard most of the noisy empty space. Data preprocessing took 95 minutes to com-

[1] Source: Digital Morphology Project, the CTLab and the Texas Advanced Computing Center, University of Texas, Austin

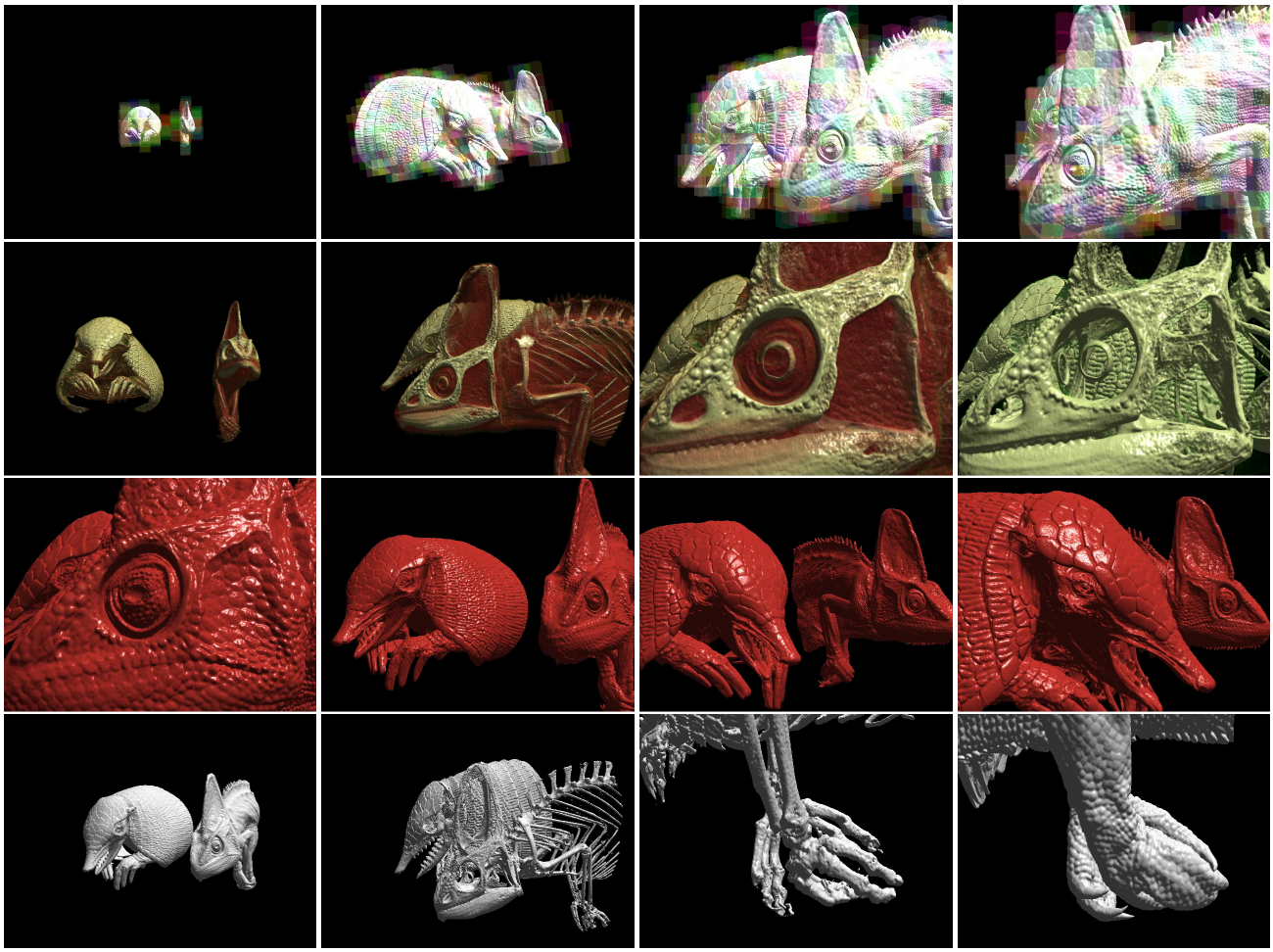


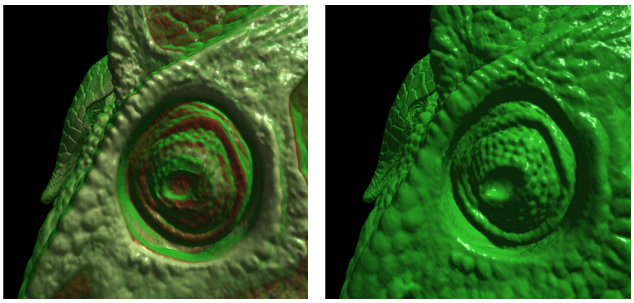
Fig. 7 Real-time inspection. These images, taken from the accompanying video, show successive instants of interactive exploration of the test CT dataset. The overall volume has a resolution of $2048 \times 1024 \times 1080$ with 16 bits/sample.

plete on a single CPU and produced an octree database with an on-disk size of 4.1 GB.

We evaluated the rendering performance of the technique on a number of interactive inspection sequences. The qualitative performance of our adaptive GPU ray-caster is illustrated in an accompanying video. Representative frames are shown in figure 7. Because of video recording constraints, the sequence is recorded using a window size of 640×480 pixels. In all recorded sequences, we used a 1 voxel/pixel accuracy to drive the adaptive renderer. As shown in the video, the system is fully interactive, and it is possible to translate, rotate, and scale the model as well as to change rendering mode, transfer functions, and isovalue parameters.

The average frame rate of the DVR sequence varies between 12 Hz and 30 Hz, with an average of 16 Hz. A few occasional frames have a delay of 200 ms, which correspond to cases in which many textures have to be updated in response to rapidly varying view conditions. These frame-rate jitters could be avoided by introducing a texture upload budget and stopping refinement when this budget is exceeded. In the case of isosurfaces, the frame rate is higher, 20 Hz on average, with peaks of up to 40 Hz. We repeated the same tests on a 1024×1024 window, and obtained an average slowdown of a factor of 3, roughly corresponding to the increase in number of pixels.

The higher performance of isosurface rendering is due to the simplicity of the inner accumulation loop, that has only to bracket the isovalue and accesses the gradient texture only once per fragment to shade the detected surface. By contrast, the direct volume rendering code requires an additional texture look-up for implementing the transfer function and accumulates more samples per fragment in the case of semitransparent materials. This latter fact is also reflected in the higher texture memory needs of semitransparent volume rendering, caused by the decreased effectiveness of visibility culling. During the entire inspection sequences, the resident set size of the application is maintained within the 600 MB of pre-allocated cache size by the out-of-core data management system. Both the isosurface and volume rendering sequences have a minimum texture memory occupation of about 475 octree bricks, corresponding to the first few frames with



(a) 2505 blocks without occlusion culling, 1334 blocks with occlusion culling

(b) 2455 blocks without occlusion culling, 1038 blocks with occlusion culling

Fig. 8 Occlusion culling feedback. Direct volume rendering images with different transfer functions, rendered on a 1024×1024 window. Visibility culling reduces the working set by about 50% in the semitransparent case, and by about 60% when surfaces get more opaque.

a view from a distance. However, the isosurface rendering sequence had an average memory occupation of 1560 bricks and a peak of 1820, while the direct volume rendering sequence had a average memory occupation of 1890 bricks and a peak of 2450.

The occlusion query mechanism has proved to be able to reduce the size of the working set, especially when using isosurface rendering or transfer functions with moderate to high opacity. A simple illustration of the benefits of visibility feedback is given in figure 8, which shows two direct volume rendering images with different transfer functions, rendered on a 1024×1024 window. Visibility culling reduces the working set by about 50% in the semitransparent case, and by about 60% when surfaces get more opaque.

8 Conclusions

We have presented an adaptive out-of-core technique for rendering massive scalar datasets within a single-pass GPU raycasting framework. The method separates the adaptive incremental maintenance of the rendering working set, which is performed on the CPU, from the actual rendering, which is fully performed on the GPU by a stackless raycaster that traverses a spatially indexed version of the current working set maintained in texture memory. Our results demonstrate that the resulting method is able to interactively explore gigavoxel datasets on a desktop PC. Besides optimizing and improving the proof-of-concept implementation, we plan to extend the presented approach in a number of ways. In particular, we are currently working on incorporating compression in the GPU representation to reduce GPU memory costs, as well as techniques to further reduce the working set through a more aggressive visibility culling based on tighter bounding volumes. We are also exploring ways to exploit the capability of our system to perform a full-volume raytacing to produce higher quality images that incorporate more advanced shading effects.

Acknowledgments. This work is partially supported by the Italian Ministry of Research under the CYBERSAR project and by the EU Marie Curie Program under the 3DANATOMICALHUMAN project (MRTN-CT-2006-035763).

References

1. Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W.: Coherent hierarchical culling: Hardware occlusion queries made useful. Computer Graphics Forum **23**(3), 615–624 (2004)
2. Boada, I., Navazo, I., Scopigno, R.: Multiresolution volume visualization with a texture-based octree. The Visual Computer **17**(3), 185–197 (2001)
3. Bruckner, S., Gröller, M.E.: Style transfer functions for illustrative volume rendering. Computer Graphics Forum **26**(3), 715–724 (2007)
4. Engel, K., Hadwiger, M., Kniss, J., Rezk-Salama, C., Weiskopf, D.: Real-time Volume Graphics. AK-Peters (2006)
5. Govindaraju, N.K., Sud, A., Yoon, S.E., Manocha, D.: Interactive visibility culling in complex environments using occlusion-switches. In: 2003 ACM Symposium on Interactive 3D Graphics, pp. 103–112 (2003)
6. Guthe, S., Strasser, W.: Advanced techniques for high quality multiresolution volume rendering. Computers & Graphics **28**, 51–?58 (2004)
7. Havran, V., Bittner, J., Sára, J.: Ray tracing with rope trees. In: L.S. Kalos (ed.) 14th Spring Conference on Computer Graphics, pp. 130–140 (1998)
8. Hong, W., Qiu, F., Kaufman, A.: Gpu-based object-order raycasting for large datasets. In: Eurographics / IEEE VGTC Workshop on Volume Graphics, pp. 177–186 (2005)
9. Kaehler, R., Wise, J., Abel, T., Hege, H.C.: Gpu-assisted raycasting for cosmological adaptive mesh refinement simulations. In: Eurographics / IEEE VGTC Workshop on Volume Graphics, pp. 103–110 (2006)
10. Kraus, M., Ertl, T.: Adaptive texture maps. In: Graphics Hardware 2002, pp. 7–16 (2002)
11. Krueger, J., Westermann, R.: Acceleration techniques for GPU-based volume rendering. In: Proc. Visualization, pp. 287–292 (2003)
12. LaMar, E.C., Hamann, B., Joy, K.I.: Multiresolution techniques for interactive texture-based volume visualization. In: IEEE Visualization '99, pp. 355–362 (1999)
13. Lefebvre, S., Hornus, S., Neyret, F.: Octree Textures on the GPU, pp. 595–613. Addison-Wesley (2005)
14. Li, W., Mueller, K., Kaufman, A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), p. 42. IEEE Computer Society, Washington, DC, USA (2003)
15. Ljung, P.: Adaptive sampling in single pass, gpu-based raycasting of multiresolution volumes. In: Eurographics / IEEE VGTC Workshop on Volume Graphics, pp. 39–46 (2006)
16. Popov, S., Günther, J., Seidel, H.P., Slusallek, P.: Stackless kd-tree traversal for high performance gpu ray tracing. Computer Graphics Forum **26**(3), 415–424 (2007)
17. Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., Strasser, W.: Smart hardware-accelerated volume rendering. In: Proc. VISSYM, pp. 231–238 (2003)
18. Scharsach, H.: Advanced GPU raycasting. In: Proceedings of the 9th Central European Seminar on Computer Graphics, p. 69?76 (2005)
19. Stegmaier, S., Strengert, M., Klein, T., Ertl, T.: A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In: Eurographics/IEEE VGTC Workshop on Volume Graphics, pp. 187–195 (2005)
20. Vollrath, J.E., Schafhitzel, T., Ertl, T.: Employing complex gpu data structures for the interactive visualization of adaptive mesh refinement data. In: Eurographics / IEEE VGTC Workshop on Volume Graphics, pp. 55–58 (2006)
21. Wilhelms, J., Gelder, A.V.: Octrees for faster isosurface generation. ACM Transactions on Graphics **11**(3), 201–227 (1992)