Error-Controlled Real-Time Cut Updates for Multi-Resolution Volume Rendering

Rhadamés Carmona[a,*] and Bernd Froehlich[b]
[a] Universidad Central de Venezuela, Facultad de Ciencias, Centro de Computación Gráfica, 1041A-Caracas Venezuela, rhadames.carmona@ciens.ucv.ve
[b] Bauhaus-Universität Weimar, Fakultät Medien, 99423 Weimar, Germany, bernd.froehlich@uni-weimar.de
* Corresponding author at: Universidad Central de Venezuela, Facultad de Ciencias, Centro de Computación Gráfica, Venezuela. Tel./fax: +58 212 6934243.

Abstract
Multi-resolution techniques are required for rendering large volumetric datasets exceeding the size of the graphics card's memory or even the main memory. The cut through the multi-resolution volume representation is defined by selection criteria based on error metrics. For GPU-based volume rendering, this cut has to fit into the graphics card's memory and needs to be continuously updated due to the interaction with the volume such as changing the area of interest, the transfer function or the viewpoint. We introduce a greedy cut update algorithm based on split-and-collapse operations for updating the cut on a frame-to-frame basis. This approach is guided by a global data-based metric based on the distortion of classified voxel data, and it takes into account a limited download budget for transferring data from main memory into the graphics card to avoid large frame rate variations. Our out-of-core support for handling very large volumes also makes use of split-and-collapse operations to generate an extended cut in the main memory. Finally, we introduce an optimal polynomial-time cut update algorithm, which maximizes the error reduction between consecutive frames. This algorithm is used to verify how close to the optimum our greedy split-and-collapse algorithm performs.

Keywords: multi-resolution volume rendering, pre-integration, data-based metrics, out-of-core visualization, volume visualization.

1. Introduction

Large volume datasets are becoming more and more common in many application areas such as in the oil and gas industry, in medicine and also as a result of simulation runs. Direct volume rendering using 3D texture mapping is the current standard for volume visualization; however, large datasets may exceed the available memory of current graphics cards and often also the main memory size of host computers. For dealing with such large volume datasets, a number of real-time multi-resolution approaches have been developed [1, 2, 3]. These approaches trade resolution of the displayed datasets for interactivity. Often an octree-based data structure is used for representing the bricked volume at different resolutions. A cut-through of the octree is chosen according to a selection criterion, which may include object space [1, 4], data space [5, 6] and image space metrics [3, 7, 8, 9]. The challenge is to update the cut on a frame-to-frame basis and render the data with the best possible quality while the volume is interactively explored. In addition, a compatible out-of-core technique needs to preload and cache data required by future update operations on the cut.

We developed agreedy algorithm to transform one cut of an octree into another one through a set of split-and-collapse operations prioritized by a global error measure. This cut update algorithm considers the size of the available texture memory as well as a limit on the number of bricks that can be replaced in each frame. The error measure allows us to quantify the error of approximating the original data by a certain cut. It is based on the transfer-function-dependent data distortion introduced by a certain cut as well as object-space parameters. When roaming through a volume using a volume probe, moving an interest point inside the volume, editing the transfer function or changing the view, this error measure is updated in real-time and it guides the split-and-collapse operations towards a new cut. This is the first approach to perform an incremental, error-guided update of the cut-through a volume and is an alternative to more ad hoc approaches such as [4]. For paging from the hard disk, we use the same algorithm and the same error measure to generate an extended cut in main memory, which serves as a cache for the split-and-collapse operations that define the cut for the actual rendering process on the graphics card.

The split-and-collapse algorithm proceeds using a greedy heuristic; it does not generally find the optimal cut (i.e. the cut with minimum error under the constraints of texture memory and download budget). For polygonal scenes, Funkhouser et al. [36] showed that the optimal level of detail (LOD) selection algorithm is equivalent to a multiple choice knapsack problem and thus it belongs to the class of NP-complete problems. In that case, the rendering time (cost) varies per object and per LOD, and the goal is to select the LOD of each object such that the image quality (the benefit) is maximized while a given frame rate (budget) is achieved. In volume rendering, the goal is to optimize the volume representation on the graphics card by selecting a cut that provides the best quality according to a given metric. The contribution to global quality (benefit) of selecting a certain brick may vary, but the cost is always a single brick space of the graphics card's memory. This makes the problem different from the knapsack problem and we can show that an optimal solution can be computed in polynomial time by traversing the octree hierarchy bottom-up using memoization. Nevertheless, our implementation of the optimal algorithm still takes from several seconds up to a few hours to compute the optimal cut; thus its applicability is limited to serve as a benchmark for the greedy algorithms.

A key challenge was to enable the real-time operation of the split-and-collapse algorithm even though a data-based metric considering the transfer function is used. A complexity analysis reveals that the update operations of our incremental greedy algorithm mainly depend on the number of bricks in the working set and the download budget. For verification we integrated two variants of the incremental greedy algorithm and the optimal algorithm in our volume rendering framework, which uses GPU-based ray casting with pre-integration. Our results indicate that the split-and-collapse algorithm performs closely to the optimal algorithm with respect to our global error measure for interactive volume exploration tasks. The main contributions of this work include:

- A real-time greedy algorithm to perform an incremental, error-guided update of the cut-through of a multi-resolution volume hierarchy, which considers a data-based metric, object-space parameters as well as bandwidth and memory limitations.
- A new optimal polynomial-time cut update algorithm, which we used to confirm that the greedy algorithm performs close to the optimum.

Further contributions include our compatible out-of-core support, which maintains an extended cut in main memory using the split-and-collapse algorithm, and adaptations of well-known volume rendering techniques to the multi-resolution context. In particular, opacity-based adaptive sampling and early ray termination require keeping track of the accumulated opacity between bricks. We show how to perform this task by using frame buffer objects [28]. The combination of adaptive sampling and pre-integration needs a 3D pre-integration table, which we efficiently compute by reusing small integrals for each diagonal on each 2D table, as was introduced in [11], but also by reusing integrals between 2D tables.

2. Previous Work

The term "large" volume rendering is used for volumes which do not fit into memory (e.g. texture memory for texture-based volume rendering [6]). To deal with this limitation, the volume can be divided into sub-volumes or bricks, generally of equal size, so that every brick can be easily replaced by another one [14]. Since the bandwidth to texture memory is limited, several improvements have been considered. Multi-resolution schemes downsample the volume to generate several levels of detail (LODs). A selection criterion is used to determine the appropriate LOD for each volume area. LaMar et al. [1] used an octree to represent the volume hierarchy in LODs divided into bricks. The distance to a user-defined point of interest and the relation between projected angle and depth of field are used to determine the LOD for each area of the volume. Boada et al. [6] also focused on a similar approach as demonstrated by [1], but the selection considers the level of importance by a user-defined region of interest, brick homogeneity and the number of bricks.

Most recent work focuses on the error of each node to determine the desired cut under the hardware constraints. There are typically two kinds of metrics used: data-based metrics and image-based metrics. Data-based metrics include the distance from the brick to the eye, region or point of interest, the brick homogeneity and also the distortion of representing the original data by a coarser (downsampled) data [1], [4, 5]. The distortion is mostly measured using the mean squared error and signal-to-noise error [5, 9]. Image-based metrics take into account the visual perception of the rendered image. The ratio between voxel size and pixel size, the projected area of a brick and the occlusion belong to this category [3]. Transfer-function-based metrics can be categorized as data-based; they primarily consider classified voxel data instead of raw voxel data. In this case, the error is defined between the classified voxels of the source data and the classified voxels of the downsampled data. Guthe et al. [9] introduced a conservative estimation of the distortion error. They define the error as the sum of the maximum color error in RGB color space and the maximum opacity error over all possible combinations of classified voxels between the coarser block and the source block. More recent work computes the distortion in the perceptually-adapted CIELUV color space [3, 15]. Ljung et al. [15] approximate the root mean square between each coarse block and its source block by using a simplified 1D histogram of 10 segments. For each voxel in each histogram segment, its distortion is computed in CIELUV color space and then multiplied by its frequency to avoid redundant computing. Wang et al. [3] computed a more accurate approximation using a full histogram of 256 entries. In addition, they considered the covariance between both blocks, the mean value and the variance on each block in CIELUV color space. Several lookup tables including a 2D histogram per block, called a summary table, and a global table corresponding to the distance between every quantized pair of samples are used to update the distortion in real-time. However, updating the distortion during transfer function changes was reported to take several seconds. Ljung et al. [15], Wang et al. [3], Guthe et al. [9], and Gobbetti et al. [23] also considered occlusion in the selection process. However, neither explicitly considered the brick download budget into the graphics hardware in the LOD selection process, nor roaming (exploration of the volume using a volume probe or 3D lens). While our approach considers the transfer function, roaming and a download budget in the selection criterion, visibility estimation is not yet supported.

The main goal of cut selection algorithms is to find the appropriate resolution for each volume area, according to a selection criterion, which commonly includes the texture memory constraint. Based on a monotonic priority (i.e. a child's priority is not larger than that if its parent [19]) defined by one or more metrics) most cut selection algorithms use a priority queue to determine the desired cut [2, 3, 6, 16, 19]. The priority queue is initialized with the root node of the octree. In an iterative process, the node with highest priority is split, which means that such a node is removed from the queue and its children are reinserted into the queue. This process continues until the texture memory limit of *N* bricks is reached or no more refinements are possible. This algorithm is O(*N*Log*N*), since less than 2*N* nodes are inserted into the sorted queue, and each insertion is O(log*N*). The remaining nodes in the queue correspond to the selected bricks for rendering – the cut. However, there is no global error measure defined to quantify how the approximation is improved by each split. Our priority function makes use of the global multi-resolution distortion and results in lower cut selection errors than previous approaches. Moreover, we introduce a cut update algorithm which also considers the download budget in the refinement process for real-time rendering.

Our focus is on interactive roaming through large volumetric datasets which has the particular requirement that in each frame the selected cut may vary. Some areas might not be shown at all since a volume probe might only be covering a small part of the volume. The first systems in this area were the Octreemizer system introduced by Plate et al. [4] and the OpenGL Volumizer introduced by Bhaniramka et al. [10]. These systems support out-of-core techniques and roaming through the volume using volume probes and slices. Bricks are pre-paged during roaming by extrapolating the volume probe movement. However, there is no error analysis or a metric for guiding the refinement process described. Castani et al. [17] extended these ideas to support domain-specific exploration tasks in the oil and gas industry. In all these systems, the LOD selection methods are based on the actually visible regions of the volume defined by the volume probes and the slices. The transition of the selected LODs between two consecutive rendering frames is based on simple heuristics. The work reported in this paper focuses precisely on this point and suggests a split-and-collapse algorithm, which is guided by an error measure.

Interactive volume roaming has some similarities with interactive terrain visualization. The challenge for terrain visualization is to dynamically update view-dependent triangle meshes and texture maps at high frame rates. The ROAM system [19] uses two priority queues to drive split-and-merge operations that maintain continuous triangulations while roaming through large terrain datasets. Our split-and-collapse algorithm is inspired by this approach and extends the general idea to the domain of large multi-resolution volumes.

Pre-integrated classification has been introduced to improve the rendering quality without impacting performance [11, 12, 13]. The main idea is to pre-compute the numerical integral for each pair of potential sample values and store them into a 2D map. During rendering, the ray is generally sampled at fixed intervals using pre-computed integrals for each pair of consecutive samples. This 2D map is constructed in O(n^3) steps. Lum et al. [11] introduced an incremental algorithm with O(n^2) complexity to update a 2D pre-integration table of n^2 entries. They observed that brute-force algorithms, which compute each entry (*i,j*) of the table by using a numerical method, repeat the integral calculation of many small ranges over the transfer function on each table diagonal. To avoid this redundancy, their algorithm incrementally builds an O(*n*) subrange integral table for each diagonal of the table, which contains all the necessary small integrals to calculate such a diagonal through compositing. To support adaptive sampling, we extend this approach to a 3D pre-integration table by reusing integrals between 2D tables.

3. Rendering Large Volume Datasets

Our research focuses on rendering large volumes on graphics cards with 3D texture mapping support. We are particularly interested in datasets larger than the graphics card's local memory and also in large datasets exceeding the size of the main memory. We make use of two levels of paging, from hard disk into main memory, and from main memory into texture memory as suggested in [4]. The main tasks include the selection of the cut that needs to fit into the graphics card's local memory and a strategy to select the set of cached bricks in main memory. We develop and describe the main components of our approach in the following order:

1. We briefly introduce the multi-resolution data representation.
2. We develop a selection criterion based on a global error metric that allows us to measure the average error per voxel incurred by representing the original volume data by a certain cut.
3. For minimizing this error, we improve upon the common greedy-style cut selection algorithm, which considers the memory constraint of the graphics card.
4. The availability of a global error metric suggests the development of an optimal cut selection algorithm, which can serve as a reference for greedy algorithms.
5. The error values for each brick change only slightly from frame to frame (e.g. due to viewpoint movement). Thus, we suggest an algorithm based on split-and-collapse operations on the nodes of the multi-resolution representation to update the cut on a frame-to-frame basis instead of selecting a new cut by starting from the root node. This algorithm considers the limited bandwidth into the graphics card in addition to the graphics card's memory constraint. We also introduce the optimal cut update algorithm, which achieves the maximal error reduction between pairs of frames.
6. Our global error metric considers the multi-resolution data distortion [3], which has to be recalculated if the transfer function changes. However, in our case, the distortion update must happen only for each node of the cut, which is typically a small subset of the entire dataset and allows us to perform this operation in real-time.

We describe each of these components in detail in the remainder of this section and carefully analyze the complexity of the algorithmic steps. The handling of out-of-core data and our contributions to rendering a multi-resolution representation are presented in Sections 4 and 5 respectively.

3.1 Data Representation

Our octree-based multi-resolution data structure is similar to approaches described in [1], [4] and [6]. The original volume is split into bricks containing, for example, 32x32x32 voxels each. Eight bricks on one level are downsampled to compute one brick on the next coarser level, which results in an octree hierarchy. Bricks on adjacent levels differ by a factor of eight in spatial extent. It is important to notice that bricks on all levels contain the same number of voxels, which allows for a simple memory management strategy.

3.2 Selection Criterion

For each brick, a priority has to be defined to determine the cut that is the best representation of the volume under the constraint of the texture memory size and bandwidth. Our priority function $E(b)$ combines two metrics: the distortion of the bricks in the transfer function domain $D(b)$ and the importance level $I(b)$, which is based on a region or point of interest and the viewpoint.

3.2.1 Distortion Metric

We define the distortion error $D(b)$ for a brick b as the sum of the distortion errors between each voxel value f_i of the source data and its approximation by voxel value b_i in brick b:

$$D(b) = \sum_{i=1}^{n(b)} D(f_i, b_i), \tag{1}$$

where $n(b)$ is the number of source voxels approximated by brick b. $D(b)$ is computed in CIELUV color space after applying the transfer function to the raw voxel values:

$$D(f_i,b_i) = d(\ L(rgb(f_i)\alpha(f_i)),\ L(rgb(b_i)\alpha(b_i))\ ), \tag{2}$$

where $rgb(v)$ is the RGB color associated with voxel v, $\alpha(v)$ is its opacity, L is a function that converts an RGB color into LUV color in CIELUV color space and $d(a,b)$ is the Euclidean distance between two colors a and b in CIELUV space [15].

The distortion $D(b)$ depends on the approximation of the values in the source data by values in an inner brick of the octree and the transfer function. Thus, the distortion for each brick needs to be recomputed if the transfer function changes. This process is accelerated by using summary tables as suggested by [3]. Storing only the non-zero entries of these sparse tables results in only a small memory overhead – about 4% of the dataset size for our datasets.

The per-voxel average distortion of approximating the source voxels by the cut can be defined as follows:

$$D(cut) = \frac{1}{\sum\limits_{b\in cut} n(b)} \sum_{b\in cut} D(b)\,, \tag{4}$$

where $\Sigma n(b)$ is the number of source voxels approximated by the cut. The distortion $D(b)$ is generally monotonic; moreover, since the parent is a coarser representation of the volume, its distortion is generally higher than the sum of the distortions of its children. Thus, the monotonicity is not only satisfied between the brick and each of its children, but is also satisfied between the brick and its finer representation, covering the same extent. We call this property *block-based monotonicity*. This property ensures that the split operation does not increase the global error $D(cut)$, which simplifies the design of selection algorithms. Only in exceptional cases, the block-based monotonicity condition is not satisfied. In these cases, the distortion of the parent can be replaced by the sum of the distortions of its children to ensure block-based monotonicity.

3.2.2 Importance Level

For real world applications, the user usually focuses the interest on specific subvolumes [1, 4, 18]. A region of interest (*ROI*) or point of interest (*PI*) is typically used to define priorities for specific volume areas. We define a priority $R(b,PI)$ based on the distance from the brick to the point of interest as follows:

$$R(b, PI) = diag(b) \,/\, (\, diag(b) + d(b,PI)\,), \tag{5}$$

where $diag(b)$ is the length of a brick's diagonal and $d(b,PI)$ is the minimum distance of the brick to the *PI*. Such a distance can be efficiently computed in object space, since the brick is simply an axis-aligned box. Note that $R(b,PI)\in(0,1]$ and coarser bricks closer to the *PI* have higher priority to be refined. The distance $d(b,PI)$ is zero if *PI* is inside *b*. We can also base the priority on the *ROI* instead of the *PI*. In this case, $d(b,PI)$ in (5) is replaced by the average of the minimum distance between the *ROI* and the brick, and the minimum distance between the brick and the *ROI* center.

A priority based on the distance to the viewpoint is defined as in (5) by replacing $d(b,PI)$ by $d(b,eye)$, where *eye* is a 3D point representing the viewpoint. We weigh both priorities to define the importance level as:

$$I(b) = (1-t)R(b,x) + tR(b,eye), \tag{6}$$

where $t\in[0,1]$, and x represent the *ROI* or the *PI*. In our experience, we found t=1/4 to be a good choice giving more priority to the *PI* or *ROI* distance. Note that $I(b)\in(0,1]$ and higher values of $I(b)$ define higher priority. The importance level and the distortion can be combined to finally define the error $E(b)$:

$$E(b)=D(b)I(b). \tag{7}$$

Since $diag(b)>diag(child(b))$ and $R(b,x)\geq R(child(b),x)$, the importance level $I(b)$ defined by (6) is monotonic, thus making our priority function $E(b)=D(b)I(b)$ also monotonic. Moreover, since $D(b)$ is block-based-monotonic and $I(b)$ is monotonic, $E(b)$ is also block-based monotonic.

A global average error per voxel can be derived from (4) with the goal of comparing the accuracy of the selection algorithms in the error minimization.

$$E(cut) = \frac{1}{\sum\limits_{b\in cut} n(b)} \sum_{b\in cut} E(b)\,. \tag{8}$$

3.3 Greedy-Style Cut Selection Algorithms

The common greedy algorithm based on splitting the node with highest error [2, 3, 6] is not always the best choice to reduce the average error *E*(*cut*), since the number of child nodes may vary and the number of bricks in the queue is limited by the texture memory size (*N* bricks). We introduce two different approaches, which produce better results. The first one improves the greedy algorithm by selecting the node with highest error reduction per child brick with respect to *E*(*cut*) instead of the node with the highest error. The second one performs an exhaustive search for the cut with the smallest error, and yields the optimal cut.

A nearly optimal algorithm must consider how much the error is reduced by each split operation. However, not every node has eight children nodes. For real-world applications, datasets may be of different size along the three dimensions or they may be constructed of bricks such that their number is non-power-of-two along one or more axes. Moreover, transparent areas can be excluded from rendering since they do not contribute to the final image. Thus, each split may add a different number of bricks into the cut. The best node for splitting is the one with highest average error reduction (*ER*) per child node which can be computed in the following way:

$$. ER(b) = \frac{E(b) - \sum_{b_i \in children(b)} E(b_i)}{|\,children(b)\,|} \tag{9}$$

Therefore, the priority queue used in the selection algorithm is conveniently sorted by *ER*(*b*) instead of *E*(*b*). Although the selection algorithm using *ER*(*b*) typically achieves higher error reduction than the selection algorithm using *E*(*b*), both approaches are only greedy techniques.

3.4 Optimal Cut Selection Algorithm

The optimal cut (hereafter referred to as *cut** in equations and OC in the narrative) is the one which minimizes the error *E*(*cut*), constrained by the texture memory size. Thus, it contains at most *N* bricks. The greedy algorithm considers only the reduction of the global error for the next split operation and is therefore a heuristic, which generally will not yield the global optimum. Obtaining the optimal cut requires an exhaustive search. One approach for finding the optimal cut can be implemented by backtracking. Starting from a cut containing only the root node, the backtracking approach would split each node of the cut at each recursive level, as long as the texture memory constraint is satisfied. After splitting one node, the updated cut is compared to the current best cut found with respect to the error *E*, replacing the current best cut if required. Also, a recursive call follows with the updated cut. After the recursive call, the status of the cut is restored by collapsing the previously refined node, and another node is then split. One recursion level is finished if all nodes of the cut have been considered for splitting. At the end of the backtracking process, the current best cut corresponds to the optimal cut.

For this optimization problem, backtracking requires exponential time, which makes it unusable even as a reference for our heuristic approach. We suggest an alternative, the polynomial-time algorithm, which computes an OC for each possible budget of exactly m bricks if it exists, with $1 \leq m \leq N$. The globally OC with a brick budget of less than or equal to *N* bricks is the one with the lowest error among this set of up to *N* OCs. A cut with exactly *N* nodes may not exist or may not be the globally OC. If multiple cuts with the lowest error exist, any one of them could be used.

We use a recursive depth-first traversal of the octree to compute the set of OCs at the root node bottom-up. A leaf node represents an OC itself, with a cost of one brick. In each inner node the set of OCs is constructed by combining the sets of OCs of its children. Also, the inner node itself is considered an OC consisting of one brick. Combining the sets of OCs may be implemented by an octary Cartesian product up to over 8 sets, where each set contains up to *N* OCs. Since we are interested in OCs with at most *N* bricks, we nest binary Cartesian products, permitting us to immediately discard cuts requiring more than *N* bricks. During the combination process, several cuts may require the same budget of bricks; however, only the one incurring the smallest error – the optimal one – is kept. The algorithm is illustrated in pseudo code in Algorithm 1. Fig. 1 shows an example of the combination process. For simplicity, Fig. 2 shows a trace of the algorithm for a small binary tree. In the pseudo code and examples we do not average the error per voxel as in (8), since the number of voxels approximated by a cut is constant. For simplicity, we use $\bar{E}(cut) = \sum E(b)$ instead.

```
Function DFS(node b) → Cut[1..N]
  Cut[1..N] Result
  // b itself is a cut consisting of one node
  Result[1] = Cut(b,1,b. $\bar{E}$)
  if IsLeaf(b) = false then
    for each child bi of b do
      Result = Combine(Result, DFS(bi))
  return Result
```

```
Function Combine(Cut[1..N] A, B) → Cut[1..N]
  Cut[1..N] C
  for i=1 to N do for j=1 to N do
    if A[i] exists and B[j] exists then
      if (i+j ≤ N) and (C[i+j] does not exist or
          A[i].$\bar{E}$+B[j].$\bar{E}$< C[i+j].$\bar{E}$) then
        C[i+j] = A[i] ∪ B[j]
  return C
```

Alg. 1: Depth-first search algorithm and combination process. Each cut is a record (*list*,*m*,$\bar{E}$). *Cut*(*b*,*m*,$\bar{E}$) constructs a cut containing only node *b*, with *m*=1 and the corresponding error $\bar{E}$. The union of two cuts is the union of their list of nodes, and the sum of their fields *m* and $\bar{E}$ respectively. The pseudo code has not been optimized to make it easier to understand.

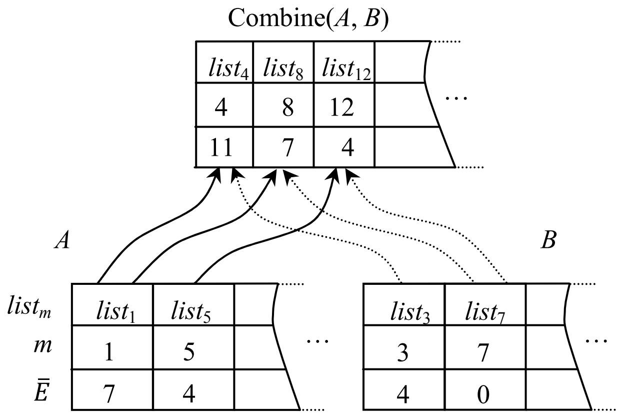


Fig. 1: An example of the combination process. In this data structure, one optimal cut is represented by a record ($list_m$,*m*,$\bar{E}$), where *m* represents the number of bricks, $\bar{E}$ the error and $list_m$ the list of *m* nodes of the cut. Each optimal cut in *A* is joined with each optimal cut in *B*. Joining two optimal cuts requires adding their fields *m* and $\bar{E}$, and the union of two lists. In this example, two joins require *m*=8 bricks, but only the cut with the smaller error is kept. Resulting cuts are discarded if they contain more than *N* bricks.

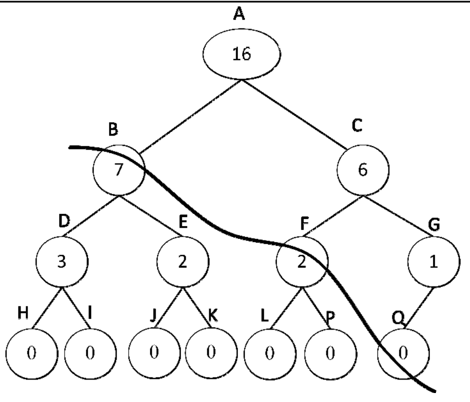
Node	**Optimal cuts (*$list_m$*, *m*, $\bar{E}$) , with *N*= 3**
H	(H,1,0) 
I	(I,1,0)
D	(HI,2,0), (D,1,3)
J	(J,1,0)
K	(K,1,0)
E	(JK,2,0), (E,1,2)
B	~~(HIJK,4,0)~~, (HIE,3,2), ~~(DJK,3,3)~~, (DE,2,5), (B,1,7)
L	(L,1,0)
P	(P,1,0)
F	(LP,2,0), (F,1,2)
Q	(Q,1,0)
G	(Q,1,0), (G,1,1)
C	(LPQ,3,0), ~~(LPG,3,1)~~, (FQ,2,2), ~~(FG,2,3)~~, (C,1,6)
A	~~(HIELPQ,6,2)~~, ~~(HIEFQ,5,4)~~, ~~(HIEC,4,8)~~, ~~(DELPQ,5,5)~~, ~~(DEFQ,4,7)~~, ~~(DEC,3,11)~~, ~~(BLPQ,4,13)~~, **(BFQ,3,9)**, (BC,2,13), (A,1,16)

Fig. 2: A complete trace of our optimal cut selection algorithm for the shown tree. Each node is identified by a letter A…Q. Optimal cuts are represented by black tuples. Gray tuples are discarded, because they exceed the texture memory size (*N*=3), or they are not optimal (i.e. there exists another cut requiring *m* bricks rooted in the same node but with a lower $\bar{E}$). The sequence of nodes on the left side corresponds to the depth-first traversal sequence of the algorithm. At root node A, there are 3 optimal cuts – one per each possible brick budget. The overall optimal cut is the cut rooted in node A with lowest error. In the example, it corresponds to (BFQ, 3, 9).

3.4.1 Optimality

The set of OCs of a parent node can be efficiently computed from the sets of OCs of its branches, which indicates that the problem exhibits optimal substructure. Also, the subproblems are overlapping, since an OC for a node can be part of different OCs of its ancestors; thus memoization – storing the OCs of each node – is an effective approach for avoiding computing an OC more than once. Optimal substructure and overlapping subproblems make dynamic programming as applied here an ideal solution [20]. The proof that the algorithm finds the optimal solution at the root node can be reduced to demonstrating that the OC obtained in each node b for each budget size m ($1 \leq m \leq N$) is a combination of OCs of its branches (or the node b itself if $m=1$).The following proof by contradiction considers that the cut of at least one of the branches of an OC in an inner node is not optimal.

Definitions:

- Let $cut(m,b)$ be a cut rooted in b with m nodes and error $\bar{E} = \bar{E}(cut(m,b))$, and let $cut^*(m,b)$ be the OC rooted in b, with m nodes and error $\bar{E}^* = \bar{E}(cut^*(m,b))$.

Hypothesis:

- $cut^*(m,b)$ is optimal, as in there does not exist another cut rooted in b with m bricks with lower error.
- The m nodes of $cut^*(m,b)$ are distributed among its max branches (at most 8 branches). The i-th branch, which is rooted in child node b_i, contains a cut $cut(m_i,b_i) \subseteq cut^*(m,b)$, with m_i nodes and error $\bar{E}_i = \bar{E}(cut(m_i,b_i))$. Thus, $m_1+m_2+\ldots+m_{max}=m$ and $\bar{E}_1+\bar{E}_2+\ldots+\bar{E}_{max}=\bar{E}^*$.

Proof by Contradiction:

- For some i, with $1 \leq i \leq max$: $cut(m_i,b_i)$ is not optimal.
- If for some i, $cut(m_i,b_i)$ is not optimal, then there exists another cut rooted in b_i with m_i bricks and error $\bar{E}_i'$ such that $\bar{E}_i' < \bar{E}_i$. So, we can build a cut $cut(m,b)$ rooted in b with $\bar{E}' = \bar{E}_1+\bar{E}_2+\ldots+\bar{E}_i'+\ldots+\bar{E}_{max}$ and $\bar{E}' < \bar{E}^*$. Thus, there exists a cut $cut(m,b)$ such that $\bar{E}(cut(m,b)) < \bar{E}(cut^*(m,b))$ and therefore $cut^*(m,b)$ is not optimal (contradiction).

3.4.2 Complexity Analysis

The depth-first traversal visits each node of the octree. In each node the sets of OCs of the up to eight branches need to be combined. For every combination of two sets of OCs, up to N^2 joins of cuts may be computed, but a new set with at most N OCs is created. A join operation of two cuts, $cut(m_1,b_1)$ and $cut(m_2,b_2)$, calculates the cut size m_1+m_2, the union of two cuts $cut(m_1,b_1) \cup cut(m_2,b_2)$ and the error sum $\bar{E}(cut(m_1,b_1)) + \bar{E}(cut(m_2,b_2))$. Thus, the cost for each join is dominated by the cost of generating the union of two cuts, which requires O(N) steps if the node lists of both cuts are copied. If references to the node lists are used, the join can be performed at a constant cost (i.e. O(1)). With K being the number of nodes in the octree hierarchy, the total complexity of the algorithm is bound by O(KN^2). Since $N \ll K$, O(K^3) is a coarser upper bound, which shows that the algorithm can be executed in polynomial time in K (size of the problem, representing the number of candidate nodes for selecting the optimal cut).

3.5 Greedy-style Cut Update Algorithm: The Split-and-Collapse Approach

Interactive changes of the transfer function, viewpoint or interest area generate different cuts, which may require a large number of new bricks to be loaded into texture memory on a frame-to-frame basis. The limited bandwidth between main memory and texture memory requires that the number of exchanged bricks per frame has to be limited to M (maximum downloads) bricks in order to maintain interactive frame rates. Therefore, in some cases, only an approximation of the desired cut can be generated if the download constraint is considered. Unfortunately, the greedy cut selection algorithm always starts from the root node to perform the refinement process. This approach may generate quite coarse representations if the download limit were to be directly considered. The split-and-collapse algorithm addresses this problem by starting from the most recent cut stored in texture memory instead of starting from the root node. Special considerations are required during interactive transfer function changes, since the most recent cut may not be valid for the new transfer function. We will first introduce the split-and-collapse algorithm for volume roaming and discuss the extra processing required for interactive transfer function editing in the next sub-section.

The goal of the split-and-collapse algorithm is to find a set of split-and-collapse operations evolving the most recent cut into the next cut such that the global error *E*(*cut*) is as small as possible under the hardware-dependent constraints, *N* and *M*. This approach is similar to the incrementally greedy algorithm introduced for interactive terrain rendering [19]. It starts by selecting the node of the cut with highest priority for splitting. The split can be performed if it does not violate *M* or *N*. If it exceeds *M*, such a split cannot be applied in this frame and the algorithm finishes. If only *N* is exceeded, a collapse operation (i.e. replacing eight siblings in the cut by their parent) needs to be performed to enable the split. The node of the cut, whose parent has the lowest priority, is the best candidate for collapsing. For full trees, a collapse operation releases seven bricks, which enables a split consuming space for seven bricks. For incomplete trees, more than one collapse operation may be required to enable a single split and a single collapse may enable more than one split.

To efficiently perform these operations and to exploit frame-to-frame coherence, we maintain two priority queues. The first queue contains the nodes of the cut representing the splitable nodes (*S*), sorted by priority in descending order. The second queue contains the collapsible parents (*C*), sorted by priority in ascending order. A node *x* is a collapsible parent if all its children belong to the cut (see Fig. 3). The nodes of the cut, which are leaf nodes, cannot be refined; however, these nodes are not considered before splitting all coarser nodes, which is generally not feasible for large datasets.

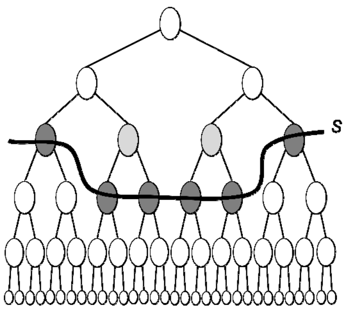


Fig. 3: Nodes of the working set. A binary tree is shown for simplicity. Dark grey nodes represent *S* and therefore the current cut. Light gray nodes represent *C*.

The sets *C*, *S* and *DL* have to be updated by each split or collapse operation, where *DL* represents the list of new bricks to be downloaded into texture memory for the next rendering frame. The algorithm finishes if any of the following conditions are satisfied in this order:

- The error of the next candidate for splitting is zero. In this case, the error cannot be further reduced, and the dataset is displayed at the highest quality for the given transfer function.
- The next candidate for splitting exceeds the download constraint. In this case, the split cannot be performed in this frame, not even by collapsing nodes.
- The next candidate for splitting S_0 exceeds the memory constraint and one of the following two conditions is met: the priority of the next collapsible parent is higher or equal than the priority of S_0 or collapsing exceeds *M*.

The split-and-collapse algorithm converges to the same cut that would be generated by the greedy selection algorithm if the priority function is monotonic [19]. The convergence may require more than one frame by considering the download constraint.

Based on the observation that splitting the node *b* with highest error *E*(*b*) does not always achieve the highest error reduction, we use a priority function based on the error reduction per split operation. We actually use the error reduction per child brick, because we want to achieve the best overall error reduction within our budget of *M* downloadable bricks. We define the priority *P*(*b*) for brick *b* as *P*(*b*)=*ER*(*b*).

The complexity of the split-and-collapse algorithm is dominated by sorting the sets *S* and *C*, which is O(*N*log*N*). Thus, the complexity of the algorithm depends on the texture memory size rather than the size of the dataset. The priority of the nodes in *S* and *C* is updated if any viewing parameter (or the transfer function) has changed from the previous frame. Any change of the priorities invalidates the current sorting of *S* and *C*, requiring them to be sorted again. However, it is not necessary to perform a full sorting of both sets, since the number of splits or collapses that can be performed per frame is limited by *M*. This observation suggests partial sorting of both sets. Partial sorting can be performed using the partial quick-sort algorithm in O(*N*+*M*log*M*) [21]. Since *M*<<*N*, it is evident that this algorithm is more efficient than classical sorting algorithms, such as quick-sort, merge-sort and heap-sort. Both *S* and *C* are split into two structures: a small set (A) containing not more than M sorted nodes, and an unsorted list (B) containing the remaining $N-M$ nodes. If a node needs to be inserted into the split structure AB, it is inserted in A in O(log$M$), and the last node of A is moved to B and also to O(log$M$), keeping A with no more than M nodes. For removing a node b, it requires O(log$M$) operations if $b \in A$, and O(1) if $b \in B$.

The split-and-collapse algorithm only performs $O(M)$ insert or remove operations on S, C and DL, each of which require O(log$M$) operations. Thus, the algorithm is bound by O(Mlog$M$), which is a much lower bound than the O(Nlog$N$) incrementally greedy algorithm introduced for roaming of terrains [19]. If the priority of the bricks does not change between two consecutive frames (i.e. the viewpoint, *PI* and *ROI* did not move, and the transfer function did not change), partial sorting does not need to be performed, except if A eventually becomes empty.

The split-and-collapse algorithm updates the cut on a frame-to-frame basis using a greedy heuristic, which does not necessarily result in the best error reduction between pairs of frames. To obtain the optimal cut update, we extend the optimal cut selection algorithm to include the bandwidth constraint. Thus, the optimal cut update algorithm has to additionally keep track of the number of downloads during the bottom-up search and also discards OCs that exceed the download constraint given by M. The maximum number of OCs in each node increases to up to N x M, and the complexity of the optimal algorithm is bound by O(KN^2M^2).

3.6 Split-and-Collapse and Transfer Function Changes

In our implementation, fully transparent nodes are not part of the cut, since they do not contribute to the final image. Bandwidth and texture memory released by these bricks are invested for non-transparent bricks, improving the multi-resolution approximation. However, by editing the transfer function, fully transparent nodes may become translucent or opaque and vice versa, which invalidates the current cut. Thus, some additional processing is necessary on the cut before the split-and-collapse algorithm can be performed.

When a new transfer function is generated, transparent nodes are removed from the cut, and new non-transparent bricks are inserted to obtain a valid cut. We use a two-stage approach to determine which non-transparent bricks to insert into the cut. The first stage performs bottom-up, starting from the nodes of the remaining cut until the root node is reached. In this stage, the nodes of the cut and all their ancestors are marked with a flag. The second stage performs top-down, visiting all non-transparent nodes from the root, until a node of the cut is reached, or an unmarked node is found. Each unmarked node is inserted into the cut to ensure a representation for each non-transparent volume area.

The insertion of new bricks into the cut may eventually exceed the hardware constraints. Thus, the collapsing of nodes needs to be performed until the constraints are satisfied. The split-and-collapse algorithm can continue with the refinement process, if the download constraint is not exceeded.

In a preprocessing step, we compute a min-max octree [22] to quickly determine which bricks are non-transparent and which bricks are transparent. We quantize the absorption function into $n-1$ sub-intervals (e.g. 255 subintervals) if a new TF is given or the absorption function is edited. Let $t(i)$ be a function, such that $t(i)=1$ if the interval $(i-1,i)$ is not fully transparent, and 0 otherwise. Then, we incrementally construct a single global table T of n integers, from $i=0$ to $i=n-1$ such that $T(i)=T(i-1)+t(i)$. Any brick with a (min,max) interval is fully transparent if $T(min)=T(max)$, and non-transparent otherwise. Note that the table T is computed in only O(n) steps whenever the absorption function changes, while testing if a brick is non-transparent or transparent only requires to compare two entries in T.

4. Out-of-core Support

We need to consider out-of-core techniques for dealing with large datasets, which fit only partially into main memory. The entire dataset may be stored on local disks or network locations. Part of the main memory needs to be reserved as a brick cache, which holds the bricks required for the current and future rendering frames. The out-of-core techniques need to be compatible with the way the split-and-collapse algorithm selects bricks for rendering. There are basically two aspects to consider: which bricks to load into main memory cache and which bricks should be replaced by newly loaded bricks. It is important that the actual loading of bricks does not stall the rendering process to keep it responsive to the user input. Thus, the brick loading should be handled in a separate loader thread concurrently running with the rendering thread [4, 18, 23]. While the main process renders the i-th frame, the loader can page bricks required for frame i+1, or pre-page bricks potentially required in future frames.

Because the bricks around the cut will be requested in future split-and-collapse operations, the paging algorithm must keep, load and replace these bricks in some way. We keep the nodes in the cut S, as well as all their ancestors in main memory. Since the bricks coarser than the cut only represent an additional fraction of the memory requirement for the cut (about 1/8 + 1/64 + 1/512 +… ≈1/7th of N for complete trees), the memory space required for the cut and inner nodes is about 8/7N bricks. The remaining cache size is used for paging finer nodes into main memory.

The paging algorithm operates in a similar way as the split-and-collapse algorithm. It uses an extended cut S^+, which is in general finer than the cut S used for rendering (see Fig. 4). The nodes of S^+ and also corresponding ancestors (inner nodes) are kept in main memory cache. S+ is updated by split-and-collapse operations. To perform a collapse operation, the set of collapsible parents C^+ is also required. The node in S^+ with highest priority (head(S^+)) is selected for splitting, and only if the split cannot be completed (due to the limited cache size), collapsing is considered. After splitting a brick b, it is kept in main memory. The nodes of C included in C^+ are not selected for collapsing in order to keep the cut S in main memory. Within this scheme, collapsing any node in C or in C^+ will never produce a page fault. Moreover, finer nodes than the rendering cut S are loaded and kept in main memory in the same priority order they would be required by the incremental selection algorithm. Thus, the bricks are paged, pre-paged and replaced according to the same priority $P(b)$, which is a more consistent strategy than other common approaches based on paging on demand using a simple LRU page replacement policy as in [16], [17] and [23].

The loader thread and the main thread need to synchronize if the priority of the bricks changes (i.e. during view changes, editing of the transfer function and roaming with a volume probe or a *PI*). Thus, the main thread sends a message to the loader thread, which has to update the priority of the nodes in S^+ and C^+, and it also performs partial sorting of both sets before the paging process can be continued. The loader thread only remains idle if the main memory cache is full and the priority of the next node to be split is smaller than the priority of the next node to be collapsed, ensuring that nodes with lower priority should not replace nodes with higher priority.

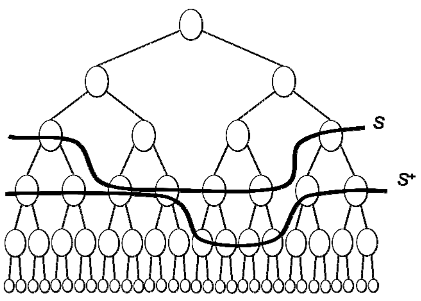


Fig. 4: Nodes of the working set. S represents the rendering cut, and S^+ represents the extended cut. All nodes in the subtree from the root to S^+ are cached in main memory.

The split-and-collapse algorithm, which is updating the cut on the graphics card, needs to only be slightly adapted such that nodes chosen for splitting do not generate page faults. This can be achieved by considering page faults during the partial-sort such that the sorted subset of S (A_s) contains only nodes for which the children are available in main memory. The partial-sort of S needs to be performed in each frame, since the children of each node of S can be loaded into or removed from main memory between frames. Collapse operations will never generate page faults, since we keep the ancestors of the cut in main memory.

5. Rendering

We use GPU-based ray casting in combination with pre-integration for rendering multi-resolution datasets. The bricks are rendered in front-to-back order. For each brick, the front faces of the corresponding bounding box are rasterized to generate the fragments, which represent the rays passing through each brick [24]. Each brick can be sampled with a constant or adaptive sample distance, starting at the brick's front faces. For each pair of consecutive samples, we use pre-integration and the integrals are composited using the 'under' operator [25]. Intervals crossing brick boundaries are clipped to the respective brick. Pre-integrated values are fetched from a 3D pre-integration table to adapt for varying sample distances.

We support adaptive sampling in combination with early ray termination to accelerate the rendering process. A common approach is to adapt the distance between samples to the level of detail of the traversed bricks [1, 6, 7, 16]. However, we found that this scheme pronounces the difference between LODs (see Fig. 5b). As an alternative, we adapt the sampling step according to the accumulated opacity along the ray (see Fig. 5c). While the opacity increases, the influence of the remaining samples decreases along the ray. Based on this observation, Danskin et al. [26] introduced the β-acceleration approach, which considers the accumulated opacity multiplied by the volume sample to determine the sampling step and the level of detail for the next sampling. Thus, the level of detail and the sampling step can increase and decrease during the ray traversal. In our approach, the level of detail of each volume area is defined by the selection algorithm, and the sampling step h increases exponentially from h_{min} to h_{max} according to the accumulated opacity:

$$h = h_{min}+(h_{max}-h_{min}).opacity^{n}. \tag{10}$$

In our experiments we found that $h_{min}=1$, $h_{max}=8$ and $n=3$ worked well for most of our datasets. This approach requires keeping track of the accumulated color and opacity for each pixel. For bricked volumes this can be efficiently achieved if there is access to the currently used framebuffer or framebuffer object (FBO). Most OpenGL® implementations allow access to the FBO, which is the current render target, by binding it as a texture [27, 28]. However, the accessed data may not be the most current due to caching. In our case, we need to access the accumulated opacity for a certain pixel. If the fetched opacity is not the most recent information, the ray can generate more sampling steps than necessary and thus this approach is conservative. Early ray termination is similarly enabled by keeping track of the accumulated opacity inside of the brick.

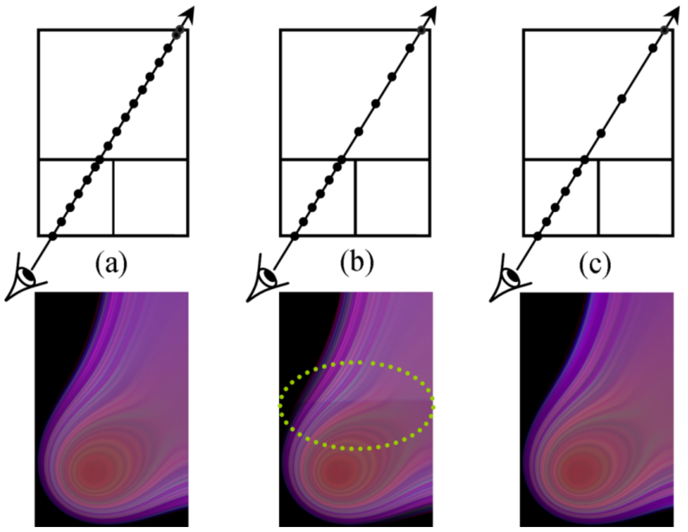


Fig. 5: Multi-resolution GPU-based ray casting. (a) Fixed sampling. (b) Adaptive sampling based on LOD. (c) Adaptive sampling based on opacity. In all cases, the last ray segment inside a brick is clipped to the brick boundary.

The fast algorithm of Lum et al. computes a 2D pre-integration table in O(n^2) by reusing small integrals on each table diagonal. This idea can also be applied to the generation of 3D pre-integration tables, which would otherwise require O(n^4) operations [13]. The naive approach would use such a 2D algorithm for each quantized value of h. However, we also found that some integrals can be obtained from already computed values by simple compositing. The color integral for a pair of unnormalized samples (f,b) separated by a fixed distance h [12] is computed in the following way:

$$C(f,b,h)=\frac{h}{(b-f)\Delta}\int_{f\Delta}^{b\Delta} C(\beta)\tau(\beta)\exp\left(-\frac{h}{(b-f)\Delta}\int_{f\Delta}^{\beta}\tau(\lambda)d\lambda\right)d\beta\,, \tag{11}$$

with $\Delta=1/n$. For the l-th table diagonal, with $l<n$, the following color integrals have to be calculated: $C(0,l,h)$, $C(1,l+1,h)$, $C(2,l+2,h)$, …, $C(n-2-l,n-2,h)$, $C(n-1-l,n-1,h)$. Note that $b-f$ equals l for all entries on the l–th diagonal. Thus:

$$C(i,i+l,h)=\frac{h}{l\Delta}\int_{i\Delta}^{(i+l)\Delta} C(\beta)\tau(\beta)\exp\left(-\frac{h}{l\Delta}\int_{i\Delta}^{\beta}\tau(\lambda)d\lambda\right)d\beta\,. \tag{12}$$

with $0 \le i \le n-1-l$. It has been demonstrated how a simple table of $2n$ entries, called a subrange integral table, is computed in linear time for each diagonal. Each entry of the diagonal can be computed by a simple compositing operation of two adjacent subrange integrals [11]. Because the 2D pre-integration table has about $2n$ diagonals, the table is calculated in $O(n^2)$.

If the sampling distance h varies due to the use of adaptive sampling, a 2D pre-integration table needs to be computed for each quantized value of h. Often adaptive sampling is performed by exponentially varying h with the level of detail [1, 7]. Thus we need to generate a 3D pre-integration table of $O(n^2m)$ entries (m=number of levels of detail), as also introduced in [29] for tetrahedral meshes. In this case, we noticed that the integrals on even diagonals for any coarse LOD with sampling distance h, can be computed by simple compositing of integrals from the next finer LOD with sampling distance $h/2$:

$$C(i,i+l,h) = C\left(i,i+\frac{l}{2},\frac{h}{2}\right) + \alpha\left(i,i+\frac{l}{2},\frac{h}{2}\right)C\left(i+\frac{l}{2},i+l,\frac{h}{2}\right). \tag{13}$$

Thus, subrange integral tables have to be computed only on odd diagonals, while for even diagonals (about 50% of the entire table) simple compositing is sufficient. During rendering, one 2D pre-integration table per LOD can be used. However, in our particular implementation, ray segments are clipped at brick boundaries requiring linear interpolation between consecutive 2D pre-integration tables. By storing 2D tables into a single 3D table, hardware-accelerated trilinear interpolation can be used to approximate integrals for arbitrary h values. Since h varies exponentially between 2D tables, the 3rd texture coordinate has to be adjusted accordingly before accessing the 3D table. For coarse levels of detail, the interpolation along the 3rd texture coordinate generates coarse approximations of integrals due to the large h step, which may produce noticeable artifacts at brick boundaries. These artifacts are significantly reduced by using a uniform 3D pre-integration table, varying h linearly between 0 and $h_{max}=2^m$ (e.g. h=0, 1, 2, 3, …, 2^m). In this case, if a common divider x exists for l and h, the color integral can be computed from already computed tables:

$$C(i,i+l,h) = C\left(i,i+\frac{l}{x},\frac{h}{x}\right) + \alpha\left(i,i+\frac{l}{x},\frac{h}{x}\right)C\left(i+\frac{l}{x},i+l,h-\frac{h}{x}\right). \tag{14}$$

In our tests, about 37% of the integrals can be obtained by simple compositing of previously computed values. However, storing and computing 2^m+1 tables are impractical. For real-world applications, we suggest to limit h_{max} to a smaller value ($h_{max} \ll 2^m$).

The 3D pre-integration table is computed in double precision and converted to single precision for compatibility with most of the current graphics hardware. During transfer function editing, a standard 2D pre-integration table is updated in real-time, and the volume is rendered with the corresponding constant sampling for immediate visual feedback.

In summary, for using pre-integration techniques in combination with adaptive ray-based volume sampling, we efficiently compute a 3D pre-integration table by reusing small integrals for each diagonal on each 2D table, as it was introduced in [11], but also, by reusing already calculated integrals between 2D tables.

6. Implementation and Results

Our implementation has been developed and evaluated under 32-bit Windows XP using Visual C++ 2005 with OpenGL® support. The reported results were achieved on a PC with a 2.4GHz dual core Intel processor, 2 GB of main memory, an NVidia Geforce 8800 GTS graphics card with 640MB on-board memory and a single SATA II hard disk.

We selected four datasets for evaluating our approach (see Fig. 6 and Table 1):

- VFCT: visible female computer tomography, from the Visible Human project ® [30].
- Angio: angiography dataset with an aneurism.
- IE: implicit equation $F(x,y,z)=0$.
- VF8b: grayscale-converted photos of the visible female, from the Visible Human project ®.

In our implementation, the bricks are loaded independently into texture memory; we do not use a texture atlas as in [7] or [23]. Our approach permits loading and rendering in interleaved mode and potentially in parallel [32]. We observed that the bandwidth increases with the brick size; however, the full bandwidth into the graphics card should not be used for downloading bricks, since it would significantly influence the actual rendering process. In our tests we use a small value of M on each dataset (see Table 1), forcing the execution of the split-and-collapse algorithm over multiple frames, even for small movements of the point of interest. While the actual bandwidth on this particular graphics card seems low for small brick sizes, this does not affect the algorithmic approach as long as one cannot replace the entire cut in graphics memory for each frame.

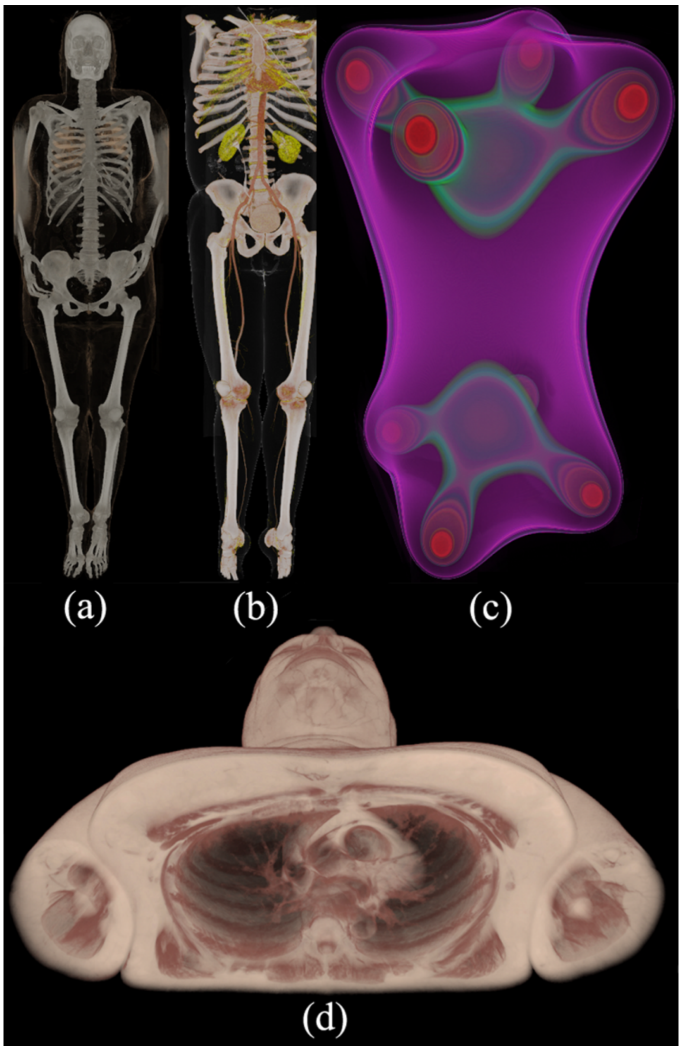


Fig. 6: Test datasets. (a) VFCT (b) Angio (c) IE (d) Visible female clipped by a volume probe.

	VFCT	**Angio**	**IE**	**VF8b**
Width (voxels)	512	512	992	2048
Height (voxels)	512	512	992	1216
Depth (voxels)	1734	1559	1984	5186
Source size (MB)	867	779.5	3723.88	12316.75
Octree size (MB)	1278.96	1146.57	4681.19	15768.75
Histo size (MB)	57.78	56.94	205.65	703.13
Histo overhead (%)	4.52%	4.97%	4.39%	4.36%
Brick size (voxels)	16^3	16^3	16^2x32	32^3
Brick size (KB)	8	8	16	32
M (bricks)	164	164	136	85

Table 1: Main parameters of the used datasets

Image quality has been compared for different priority functions $E(b)$ (see Fig. 7). The comparison between two images is measured by pixel-wise differences in CIELUV color space as in [3] and [15]. In this space, the Euclidean distance defines the distortion $D(a,b)$ between two colors a, b. A distortion $D(a,b)<6$ is not noticeable according to [3] and [15]. For $E(b)=I(b)$, the distortion is not perceptible around the interest point, but is dramatically noticeable around the legs and shoulders. If only the distortion is considered in the priority function (i.e. $E(b)=D(b)$), the difference in visual quality is not noticeable in most opaque areas (e.g. bones) since they have higher priority, and therefore, higher resolution. The distortion is quite homogeneous in pixels revealing only translucent areas (e.g. skin). If both metrics are combined, $E(b)=D(b)I(b)$, the quality at bones and around the interest point is mostly preserved, while the distortion increases in some other areas (e.g. thorax). However, the rendering does not show dramatic distortions as using only *I*(*b*) does.

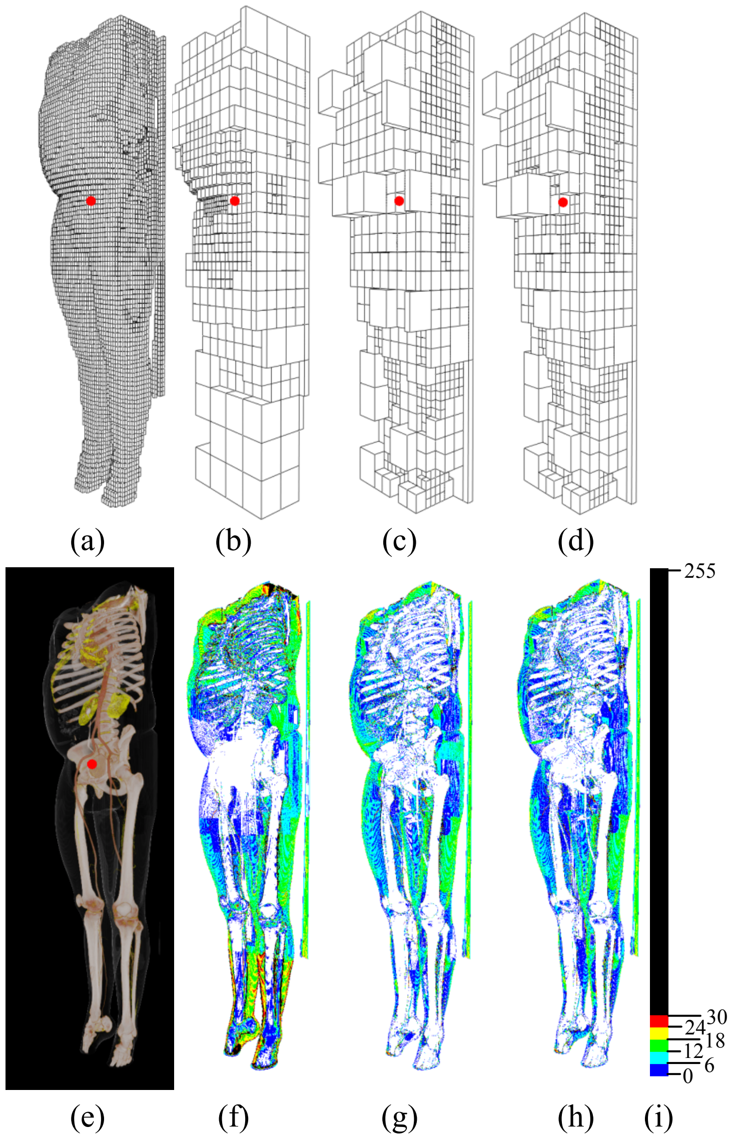


Fig. 7: Calculating the pixel-wise distortion for different priority functions. Selected bricks for rendering: (a) bricks at finest level of detail, using about 400MB of texture memory, (b,c,d) use 40MB of texture memory with different priority functions. (e) represents the reference image, generated by the cut (a). (f,g,h) show the pixel-wise distortion after applying the color map shown in (i). (b,f) *E*(*b*)=*I*(*b*), (c,g) *E*(*b*)=*D*(*b*), (d,h) *E*(*b*)=*D*(*b*)**I*(*b*). The greedy cut selection algorithm was used with *P*(*b*)=*E*(*b*). The red point represents the interest point, which is located at the aneurysm, inside of the body.

Using the pre-computed 2D histograms for the inner nodes of the octree and the min-max octree, we are able to update our global transparency table *T* and the distortion of each brick in the entire octree hierarchy in 0.1 to 0.4 seconds while the transfer function is being edited. [3] reported 5sec update time for this case on comparable hardware. However, the split-and-collapse approach only needs to update the distortion and priority in a small hull around the cut (i.e. nodes in *C*, nodes in *S* and the children of every node of *S*) which represents a small subset of the entire dataset. In our test, the update consumes less than 0.04 seconds, which supports changes of the transfer function in real-time.

The greedy and optimal cut selection algorithms were compared with respect to error (see Table 2). For this comparison, the point of interest was randomly moved through the volume for 100 frames, and the control points of the transfer function were modified for another 100 frames. The optimal algorithm is still computationally quite expensive and we had to keep the number of bricks in texture memory small to achieve reasonable computing times for the different datasets. Texture memory size for each dataset was selected to be 10MB for VFCT, 20MB for Angio, 40MB for IE and 80MB for VF8b. Two versions of the greedy algorithm were considered: the naive approach, which selects the node with the highest error for splitting $P(b)=E(b)$, and the improved approach, which selects the node with the highest error reduction per child node for splitting $P(b)=ER(b)$. In our tests, the error of the naive algorithm is only up to 3% higher than the error of the other algorithms for common transfer functions used for medical data (VFCT, Angio and VF8b). However, for transfer functions with high frequencies (e.g. TF of IE in Fig. 8b), the global error of the naive approach is up to 15% higher. The difference between the optimal algorithm and the improved approach is small (generally smaller than 0.2%), indicating that our improved greedy cut selection algorithm is a nearly-optimal approach. However, both greedy algorithms only consume between 1 and 4 milliseconds per frame, while the optimal algorithm takes up to 3 hours per frame to generate the optimal cut, making it only useful as a reference. We had also implemented the backtracking algorithm, which took several days even for a single frame and small texture memory size.

Dataset (texture memory) →		**VFCT (10MB)**	**ANGIO (20MB)**	**IE (40MB)**	**VF8b (80MB)**
Error Changing PI	(a) Naive	1.4428	1.3982	9.8849	0.0450
	(b) Improved	1.3988	1.3608	9.1839	0.0447
	(c) Optimal	1.3967	1.3598	9.1687	0.0447
	(b) vs. (a)	3.05%	2.67%	7.09%	0.69%
	(c) vs. (a)	3.20%	2.75%	7.25%	0.75%
	(c) vs. (b)	0.16%	0.08%	0.16%	0.05%
Error Editing TF	(a) Naive	5.5929	4.3251	25.8419	3.0308
	(b) Improved	5.4458	4.2660	22.4131	3.0038
	(c) Optimal	5.4433	4.2636	21.9592	3.0033
	(b) vs. (a)	2.63%	1.37%	13.27%	0.89%
	(c) vs. (a)	2.68%	1.42%	15.02%	0.10%
	(c) vs. (b)	0.05%	0.06%	2.03%	0.02%

Table 2: Comparing the error $E(cut)$ for the three different selection algorithms.

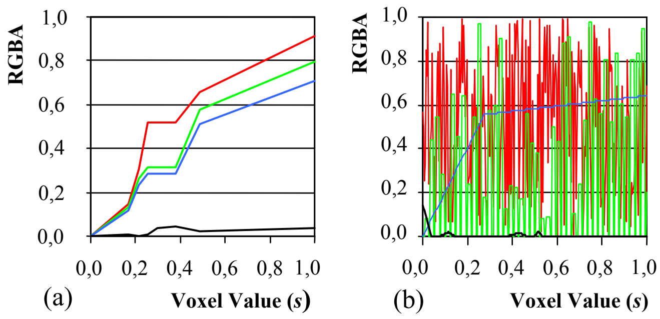


Fig. 8: Transfer function of VF8b (a) and IE (b). The darkest line corresponds to the absorption function $A=\tau(s)$. (b) Red and green channels were randomly generated to obtain a high frequency transfer function. The absorption function reveals semi-transparent surfaces and semi-transparent volume areas, which is challenging for multi-resolution volume rendering systems.

The error reduction per frame for the cut update algorithms was also compared (see Table 3). Here, the number of bricks M that can be downloaded in every frame is limited. For comparison purposes, we take the result of the optimal algorithm as the base of 100% of error reduction while the initial cut that the cut update algorithms start from represents the 0%. With this measure, the difference between the naive approach and the improved approach is very noticeable; the improved approach is up to 70% better in error reduction than the naive approach. On average, the optimal algorithm performs between 6% and 18% better than the improved approach, and up to 87% better than the naive approach. Fig. 9 shows a visual comparison between the naive approach and the improved approach for the Angio dataset. In this test case, the optimal algorithm generates the same cut as the improved approach.

Changing PI		Input Error	Output Error	Reduced Error	%Reduc. Error
VFCT	(a) Naive	1.5555	1.4683	0.0872	74.7%
	(b) Improved		1.4465	0.1090	93.4%
	(c) Optimal		1.4388	0.1167	100.0%
IE	(a) Naive	10.0569	9.8958	0.1611	34.8%
	(b) Improved		9.6462	0.4107	88.6%
	(c) Optimal		9.5934	0.4635	100.0%
Editing TF		**Input Error**	**Output Error**	**Reduced Error**	**%Reduc. Error**
VFCT	(a) Naive	5.6433	5.5913	0.0520	33.7%
	(b) Improved		5.4981	0.1452	94.2%
	(c) Optimal		5.4892	0.1540	100.0%
IE	(a) Naive	26.2268	26.0345	0.1923	13.1%
	(b) Improved		25.0328	1.1940	81.6%
	(c) Optimal		24.7629	1.4639	100.0%

Table 3: Comparing incremental selection algorithms with respect to our error metric $E(cut)$ for the case that the point of interest (PI) is continuously moved through the volume and during editing of the transfer function (TF).

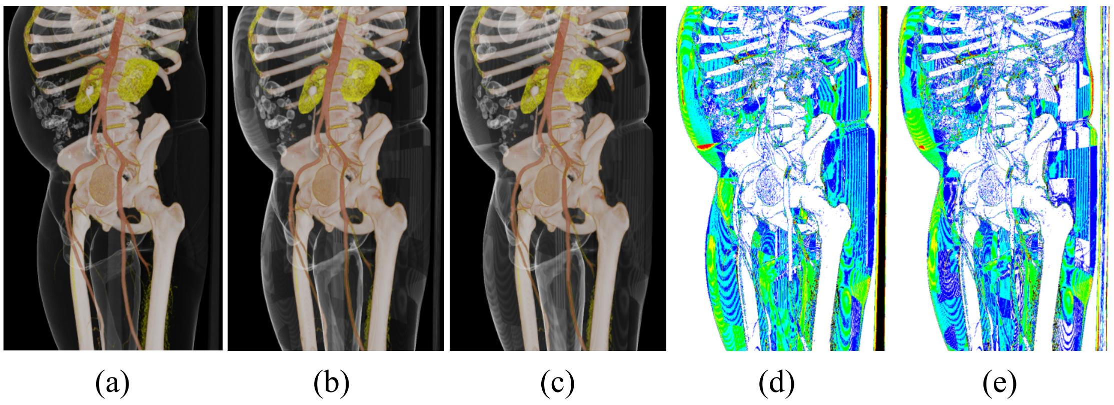


Fig. 9: Comparing cut update selection algorithms. (a) Reference image at full resolution. (b,c) Images generated with (b) naive approach and (c) the improved approach. (d) Pixel-wise error in CIELUV color space for the naive approach. $E(cut)$ =10.84 and the average pixel distortion is 9.834. (e) Pixel-wise error for the improved approach. $E(cut)$=10.69 and the average pixel distortion is 7.975. Both approaches start from a fixed cut, and the resulting images (b,c) were generated after applying the corresponding split-and-collapse algorithm toward the next frame.

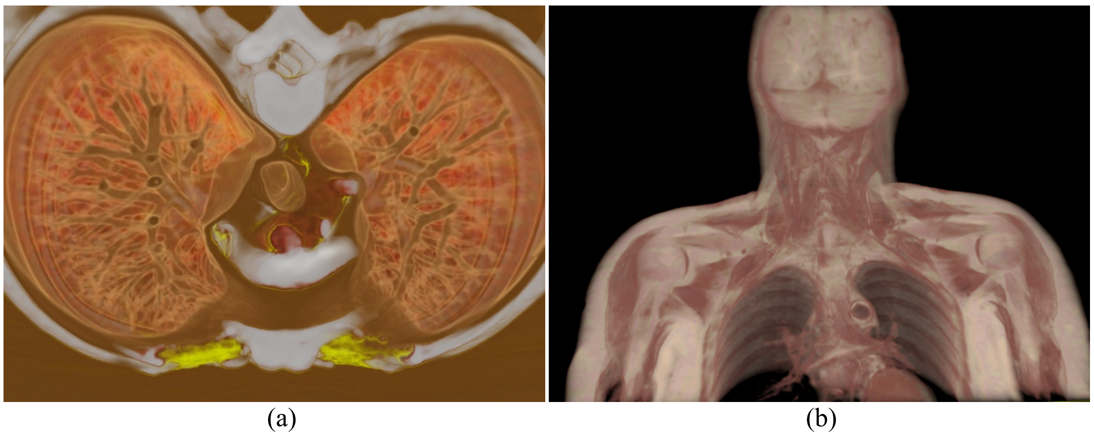


Fig. 10: The views used for evaluating the influence of adaptive sampling and early ray termination. (a) Pulmonary area of the Angio dataset, viewed from the top. (b) VF8b dataset. In both cases, the ROI has been zoomed to fit into the viewport of 1024x768 pixels.

We evaluated and compared the described acceleration techniques in our multi-resolution framework. Fig. 10 shows two images generated with adaptive sampling based on opacity and early ray termination.

Quantitative results are shown in Table 4, comparing the frame rate (frames per second) for all cases. Our implementation of early ray termination improves the frame rate typically between 20% and 50%. Adaptive sampling based on LOD is typically faster than adaptive sampling based on opacity. However, the former pronounces the difference between adjacent bricks with different LODs (see Fig. 5), and images generated with the latter are very similar to constant sampling.

Dataset	Early Ray Termination	Constant	Adaptive Opacity	Adaptive LOD
Angio	No	3.66 fps	8.26 fps	7.15 fps
	Yes	9.06 fps	11.48 fps	14.65 fps
VF8b	No	7.66 fps	8.84 fps	18.88 fps
	Yes	7.83 fps	9.38 fps	20.25 fps

Table 4: Frame rates for different acceleration technique configurations.

Table 5 shows the calculation times of the 3D pre-integration tables for different levels of quantization (n) of the transfer function. We selected two extreme transfer functions for this test: one with high frequencies (Fig. 8b), and a simple, almost linear function (Fig. 8a). We requested a precision of seven exact decimals for the computation of the small integrals using adaptive Simpson integration [33]. The optimization introduced in this work reduces the computing time by about 40% for the full exponential table, and by about 35% for the uniform table. Even though the TF of the IE dataset contains higher frequencies than the TF of VF8b, the computing time is shorter for IE because the function support is shorter (i.e. sub-domain of the TF where the absorption is not zero). Pre-integration tables exceeding 2GB of contiguous memory could not be computed in this test due to platform limitations.

TF	*N*	Exponential Table kn^2 entries			Uniform Table $2^k n^2$ entries		
		Naive	Optim.	%Impr.	Naive	Optim.	%Impr.
IE (k=7)	64	0.0434	0.0255	41.19	1.1300	0.7550	33.19
	128	0.1703	0.1024	39.88	3.4770	2.2340	35.75
	256	0.5800	0.3568	38.48	10.9740	7.1810	34.56
	512	2.3410	1.4480	38.15	42.7290	28.1910	34.02
	1024	9.5290	5.8920	38.17	--	--	--
	2048	38.9140	24.3940	37.31	--	--	--
VF8b (k=9)	64	0.0975	0.0526	46.10	88.828	53.719	39.52
	128	0.3035	0.1783	41.25	208.674	136.875	34.41
	256	0.9878	0.5920	40.07	497.388	326.157	34.43
	512	3.7490	2.2440	40.14	--	--	--
	1024	14.6820	8.8280	39.87	--	--	--
	2048	59.2190	35.5540	39.96	--	--	--

Table 5: Comparing the naive approach to the optimized approach for calculating a 3D pre-integration table. Computing time is given in seconds.

While the tests shown in Table 5 updates k or 2^k two-dimensional tables, we can limit the step width to h=8, such that no more than nine 2D tables are required, reducing the memory consumption and the response time. Moreover, if the transfer function is being edited, we switch to constant sampling mode, which requires a single pre-integrated 2D table of 256x256 entries to ensure interactive response. Such a table is computed in about 0.05 seconds in our tests.

7. Conclusions and Future Work

The split-and-collapse algorithm is the first interactive approach for an error-controlled cut update of a multi-resolution volume representation, which considers the necessary constraints for real-world applications. We also developed an optimal polynomial-time cut update algorithm and used it to benchmark the greedy split-and-collapse algorithm. The results provide empirical evidence that the split-and-collapse cut update algorithm performs very close to the optimal algorithm for a variety of test datasets. Furthermore, we suggest that out-of-core handling can be better supported by an extended cut in main memory instead of the commonly used LRU strategy. For using pre-integration in combination with adaptive sampling in a hierarchical volume representation, a 3D pre-integration table is necessary. We show that such a 3D pre-integration table can efficiently be computed by reusing already calculated integrals between 2D tables in addition to reusing small integrals for each diagonal of each 2D table.

Our implementation of the split-and-collapse algorithm in our multi-resolution volume rendering framework supports GPU-based ray casting including opacity-based adaptive sampling, early ray termination and pre-integration. The system performs in real time for common interaction tasks, including transfer function changes, roaming and viewpoint changes. The complexity analysis shows that the split-and-collapse algorithm and the out-of-core techniques do not depend on the dataset size, but on the actual working set size, which enables our framework to handle in principle arbitrarily large datasets. Compression techniques (e.g. wavelet tree [9]) have the potential to make better use of the available bandwidth of the out-of-core layer. Multi-resolution rendering artifacts can be ameliorated by using a one-brick wide transition zone as suggested in [35]. These smooth transitions require contiguous levels of detail of adjacent bricks and thus a restricted octree cut, which could be enforced by considering this constraint during the split-and-collapse operations.

The split-and-collapse algorithm could be used in other contexts as well, where a continuously changing approximation of a multi-resolution dataset is required. The algorithm can also be extended to work for 4D datasets, where it is crucial to use the available memory and bandwidth resources in the best possible way. However, animated transitions between two temporal cuts would need to be performed to enable smooth animations. Particularly in this case but also for static volumes, temporal occlusion coherence [34] can be considered by the split-and-collapse algorithm to exclude occluded areas from rendering and focus the resolution on visible areas.

Acknowledgment

We are thankful to Jorge Bernadas for all his suggestions regarding the optimal polynomial-time cut update algorithm. The second author thanks the VRGeo consortium for its continuous support. We also thank the associate editor and the reviewers for their patience and numerous suggestions.

References

[1] E. LaMar, B. Hamann and K.I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. Proc. IEEE Visualization '99; 1999, p. 355-362.

[2] P. Ljung, C. Lundström and A. Ynnerman. Multiresolution Interblock Interpolation in Direct Volume Rendering. Proc. EUROGRAPHICS/IEEE-VGTC Symposium on Visualization '06; 2006, p. 256-266.

[3] C. Wang, A. García and H.-W. Shen. Interactive Level-of-Detail Selection Using Image-Based Quality Metric for Large Volume Visualization. IEEE Transactions on Visualization and Computer Graphics, Vol. 13, No. 1; 2007, p. 122-134.

[4] J. Plate, M. Tirtasana, R. Carmona and B. Froehlich. Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes. Proc. EUROGRAPHICS/IEEE TCVG Symposium on Visualization '02; 2002, p. 53-60.

[5] C. Wang, J. Gao, and H.-W. Shen. Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. Proc. EUROGRAPHICS Symposium on Parallel Graphics & Visualization '04; 2004, p. 23-30.

[6] I. Boada, I. Navazo and R. Scopigno. Multiresolution Volume Visualization with Texture-based Octree. The Visual Computer, vol. 17; 2001, p. 185-197.

[7] P. Ljung. Adaptive Sampling in Single Pass, GPU-based Ray Casting of Multiresolution Volumes. Proc. EURO-GRAPHICS/IEEE International Workshop on Volume Graphics '06; 2006, p. 39-46.

[8] H. Zhou, M. Chen, and M.F. Webster. Comparative Evaluation of Visualization and Experimental Results Using Image Comparison Metrics. Proc. IEEE Visualization '02; 2002, p. 315-322.

[9] S. Guthe and W. Straßer. Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. Computers & Graphics, vol. 28, no. 1; 2004, p. 51-58.

[10] P. Bhaniramka and Y. Demange. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets. Proc. IEEE Symposium on Volume Visualization and Graphics '02; 2002, p. 45-54.

[11] E. Lum, B. Wilson and K. L. Ma. High-Quality Lighting and Efficient Pre-Integration for Volume Rendering. Proc. EUROGRAPHICS/IEEE TCVG Symposium on Visualization '04; 2004, p. 25-34.

[12] K. Engel, M. Kraus and T. Ertl. High Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware '01; 2001, p. 9-16.

[13] S. Roettger and T. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. Proc. IEEE Symposium on Volume Visualization and Graphics '02; 2002, p. 23-28.

[14] R. Grzeszczuk, C. Henn and R. Yagel. Advanced Geometric Techniques for Ray Casting Volumes. Course Notes No. 4, Annual Conference on Computer Graphics (SIGGRAPH '98), 1998.

[15] P. Ljung, C. Lundström, A. Ynnerman and K. Museth. Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets. Proc. IEEE Symposium on Volume Visualization and Graphics

'04; 2004, p. 25-32.
[16] S. Guthe, M. Wand, J. Gonser and W. Straßer. Interactive Rendering of Large Volume Data Sets. Proc. IEEE Visualization '02; 2002, p. 53-60.
[17] L. Castani, B. Lvy and F. Bosquet. Volume Explorer: Roaming Large Volumes to Couple Visualization and Data Processing for Oil and Gas Exploration. Proc. IEEE Visualization ’05; 2005, p. 247-254.
[18] P. Ljung, C. Winskog, A. Persson, K. Lundström and A. Ynnerman. Full Body Virtual Autopsies using a State-of-the-art Volume Rendering Pipeline. IEEE Transactions on Visualization and Computer Graphics, vol. 12, No. 5; 2006, p. 869-876.
[19] M. Duchaineau. M, Wolinsky. D.E. Sigeti, M.C. Miller, C. Aldrich and M.B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. Proc. Visualization '97; 1997, p. 81-88.
[20] T. Cormen, C. Leiserson, R. Rivest and C. Stein. Introduction to Algorithms, 2nd edition, MIT Press & McGraw-Hill, ISBN 0-262-03293-7; 2001.
[21] C. Martínez. Partial Quicksort. Proc. 6th ACM-SIAM Workshop on Algorithm Engineering and Experiments (ALENEX) and the first ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics (ANALCO), SIAM; 2004, p. 224-228.
[22] W. Li, K. Mueller, and A. Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-Based Volume Rendering. Proc. IEEE Visualization '03; 2003, p.317-324.
[23] E. Gobbetti, F. Marton and J.A. Iglesias. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. The Visual Computer, vol. 24; 2008, p. 797-806.
[24] J. Krüger and R. Westermann. Acceleration techniques for GPU-based Volume Rendering. Proc. IEEE Visualization '03; 2003, p. 287-292.
[25] N. Max. Optical Models for Direct Volume Rendering. Visualization in Scientific Computing, Springer; 1995, p. 35-40.
[26] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. Proc. Workshop on Volume Visualization '02; 1992, p. 91-98.
[27] R. Kaehler, J. Wise, T. Abel and H.C. Hege. GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations. Proc. Volume Graphics; 2006, p. 103-110, 144.
[28] H.-F. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig and B. Froehlich. Ray Casting of Trimmed NURBS Surfaces on the GPU. Proc. IEEE Symposium on Interactive Ray Tracing; 2006, p. 151-160.
[29] M. Kraus, W. Qiao and D. Ebert. Projecting Tetrahedra without Rendering Artifacts. Proc. IEEE Visualization '04; 2004, p. 27-34.
[30] Visible Human Project®. http://www.nlm.nih.gov/research/visible/visible_human.html.
[31] N. Mark and G. Jean-loup. The Data Compression Book. 2nd Edition, M&T Books, ISBN:1-55851-434-1, New York; 1995.
[32] C. Rezk-Salama. Volume Rendering Techniques for General Purpose Graphics Hardware. Thesis dissertation, Erlangen-Nürnberg University, Germany; 2001.
[33] R. Burden and D. Faires. Numerical Analysis. 7th edition, Brooks/Cole, ISBN: 0-534-38216-9; 2000.
[34] J. Gao, H-W. Shen, J. Huang, and J.A. Kohl. Visibility Culling for Time-Varying Volume Rendering Using Temporal Occlusion Coherence. IEEE Visualization '04; 2004, p. 147-154.
[35] R. Carmona and B. Fröhlich. Reducing Artifacts between Adjacent Bricks in Multi-resolution Volume Rendering. Springer-Verlag, Advances in Visual Computing, LNCS 5875; 2009, p. 644-655.
[36] T. A. Funkhouser and C. H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. Proc. ACM SIGGRAPH '93; 1993, p. 247-254.

Vitae

Rhadamés E. Carmona is an associate professor with the Computer Science department at Central University of Venezuela (UCV). He received his M.S. and Ph.D. degrees from UCV in 1999 and 2010 respectively. His research interests include scientific visualization, imaging and visual computing.

Bernd Froehlich is currently a full professor with the Media Faculty at Bauhaus-Universität Weimar, Germany. From 1997 to 2001 he held a position as senior scientist at the German National Research Center for Information Technology (GMD), where he was involved in scientific visualization research. From 1995 to 1997 he worked as a Research Associate with the Computer Graphics group at Stanford University. Bernd received his M.S. and Ph.D. degrees in Computer Science from the Technical University of Braunschweig in 1988 and 1992, respectively. He has served as a program co-chair for the IEEE VR, 3DUI, IPT/EGVE and VRST conferences. He is also a co-initiator of the 3DUI symposium series and is the winner of the 2008 Virtual Reality Technical Achievement Award. His research interests include real-time rendering, visualization, 2D and 3D input devices, 3D interaction techniques, display technology and support for collaboration in co-located and distributed virtual environments.