Vol. 13, No. 2: 21–37

Simple Empty-Space Removal for Interactive Volume Rendering

Vincent Vidal
INRIA-Evasion

Xing Mei
CASIA-NLPR/LIAMA

Philippe Decaudin
INRIA-Evasion

Abstract. Interactive volume rendering methods such as texture-based slicing techniques and ray-casting have been well developed in recent years. The rendering performance is generally restricted by the volume size, the fill-rate and the texture fetch speed of the graphics hardware. For most 3D data sets, a fraction of the volume is empty, which will reduce the rendering performance without specific optimization. In this paper, we present a simple kd-tree based space partitioning scheme to efficiently remove the empty spaces from the volume data sets at the preprocessing stage. The splitting rule of the scheme is based on a simple yet effective cost function evaluated through a fast approximation of the bounding volume of the non-empty regions. The scheme culls a large number of empty voxels and encloses the remaining data with a small number of axis-aligned bounding boxes, which are then used for interactive rendering. The number of the boxes is controlled by halting criteria. In addition to its simplicity, our scheme requires little preprocessing time and improves the rendering performance significantly.

1. Introduction

Interactive volume rendering methods play an important role in scientific visualization and computer graphics [Hadwiger et al. 06]. Due to the rapid

1086-7651/06 $0.50 per page

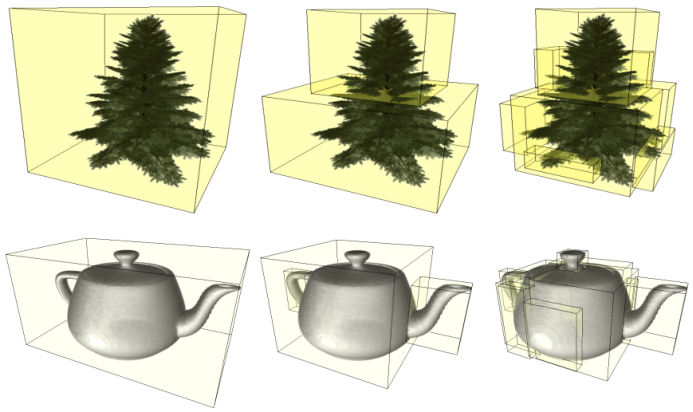

Figure 1. An illustration of our empty-space removing method. Empty voxels are progressively removed from the original volume with a kd-tree based spatial partitioning scheme.

improvement of graphics hardware, several efficient algorithms have been proposed in the last few years, such as slice-based methods with 2D textures or 3D textures [Cabral et al. 94, Engel and Ertl 02] and GPU-accelerated ray casting [Kruger and Westermann 03, Stegmaier et al. 05]. They produce good visualization results at interactive framerates. For slice based methods, a stack of planes are generated to resample the volume data by texture mapping, while for ray casting methods, some fragment operations such as texture fetches and arithmetic calculations are performed on each pixel of the rasterized bounding box of the volume. If a large fraction of the volume contains no information of interest, these methods will generate lots of fragments that do not contribute to the final image. Unnecessary operations on these void fragments will bring down the rendering performance significantly. Several techniques have been proposed to solve this issue. For static volumes (with no time-varying data or dynamic transfer functions), one possible way to improve the performance is to remove most of the empty space in the volume at the preprocessing stage before sending it to the rendering pipeline.

This idea was first described by [Levoy 90] for ray casting methods. Spatially coherent information was encoded into an octree structure and was used to skip empty space along the marching ray. An efficient GPU implementation of the aforementioned technique was presented in [Kruger and Westermann 03]. This approach requires a multi-pass rendering process and can not be easily integrated with the slice-based methods. The octree structure has also been widely used in 3D texture based rendering as a multi-resolution representation [LaMar et al. 99]. However, since the splitting rule for the octree can not be adjusted flexibly with the volume content, many boxes are needed

for removing a significant amount of empty space. [Kähler et al. 03] proposed another hierarchical data structure known as an AMR (Adaptive Mesh Refinement) tree. The process of building an AMR tree is based on a complex clustering algorithm which also leads to large number of boxes. [Tong et al. 99] proposed an alternative approach to split the volume into equal-sized blocks along one selected axis and trim each block to a smaller size with the bounding volume. The uniform space partition along one axis made the approach inefficient for unevenly distributed volume data. [Li and Kaufman 03] presented another volume partitioning method called "Box Growing". Starting from seed texels, a set of boxes grow to find the connected regions with similar properties. The resulting box distribution is closely related to a non-trivial selection of the seed texels and the method might produce a large number of boxes.

In this paper, we present a simple empty space removing method for interactive rendering of static volumes. Our first concern is to cull a large number of empty voxels from the volume with a small number of boxes. The rendering efficiency not only depends on the number of empty voxels filtered out, but also on the granularity of the partitioning. Too many boxes will increase the burden of the rendering algorithm. For slicing-based rendering, this will lead to expensive back-to-front box sorting and context switching. And for raycasting, the box traversal cost (boxes/ray intersections) will become non-negligible. Our second concern is to keep the preprocessing time low so that (re)processing the data will not be prohibitive. We choose the flexible kd-tree structure as our hierarchical representation for the volume data. Starting with one bounding box that encloses the volume, the method recursively splits the volume and builds up the kd-tree. The splitting plane for each node of the tree is chosen with a cost function, which estimates how many empty voxels can be removed from the node by the splitting. The cost function relies on the computation of the bounding volume for any given axis-aligned region, which can be accelerated with a 3D-summed-volume table. We then traverse the kd-tree to get a set of axis-aligned bounding boxes (AABBs) enclosing the regions of interest and render them. Due to the efficient space removing scheme, our preprocessing method is fast and the rendering performance is improved significantly. We obtain a hierarchical structure which progressively fits the non-empty voxels according to a specified traversal depth. This depth can be deduced from some intuitive user defined parameters (maximum number of boxes, percentage of empty space to remove...) or even selected interactively.

The kd-tree structure is not new for volume rendering. [Subramanian and Fussell 90] proposed to accelerate volume ray tracing with a kd-tree structure. The main difference between their method and our method is the partitioning rules: they employ the median-cut method to generate a well balanced tree with a small depth limit, which brings benefits for fast tree traversal and ray tracing, while we focus on removing the empty space, which is more

important for interactive rendering methods. In parallel with the Box Growing scheme [Li and Kaufman 03] for volume subdivision, [Li et al. 03] suggested converting all the grown boxes into a kd-tree for quick visibility sorting. The kd-tree structure was not used for the generation of the sub-volumes.

2. Empty Space Removing Preprocess

Our preprocessing method is divided into two phases: the kd-tree generation phase and the subvolume list generation phase. In the first phase, we build the kd-tree hierarchical structure from the volume data set with a recursive top-down algorithm. In the second phase, we traverse the tree to get a list of all the non-empty subvolumes. The subvolume list is represented as a set of AABBs, which can be easily rendered with slice-based methods or ray casting methods. In this section, we describe the two phases in detail.

2.1. Kd-Tree Generation

We assume that the original volume data V is stored in a $l_x \times l_y \times l_z$ 3D array where l_x, l_y, l_z is the volume resolution in the X, Y, Z direction respectively. The value of voxel (i, j, k) in volume V is denoted as $V(i, j, k)$. In a preparation step, the voxels in V are classified into "empty" and "non-empty" ones with a classification threshold specified by the user. We generate a copy of the volume V, denoted as V_1, and store the classification results in it:
If voxel (i, j, k) is empty $V_1(i, j, k) = 0$, otherwise $V_1(i, j, k) = 1$.
Since V_1 provides enough information on the spatial distribution of the data in V, we work on V_1 to remove the empty voxels.

Our kd-tree building process starts with an axis-aligned region that encloses the entire volume V_1. We first check if the region is divisible or not according to some specified criteria. If it is divisible, the region is divided into two subregions and labeled as an interior node. The two subregions are inserted into the tree as the child nodes, and their bounding boxes are adjusted to a smaller size to discard the empty voxels in the parent node. The subdivision procedure is then repeated recursively on the two child nodes. If the current region is not divisible (ie., one of the halting criteria is satisfied), then the region is labeled as a leaf node, and the recursion is terminated.

The pseudocode for the tree building process is presented in Algorithm 1: In the **TreeBuilding** function block, we initialize the root node with the entire volume V_1 and build the kd-tree hierarchical structure by calling the **NodeSplitting** function, which splits each interior node of the tree in a recursive way.

Algorithm 1. (Pseudocode for the kd-tree building process)

```
struct TreeNode {
    bb              ▷Axis-aligned Bounding Box of the node
    left_ptr        ▷pointer to the left child
    right_ptr       ▷pointer to the right child
    depth           ▷current node depth in the tree
}

function NodeSplitting(TreeNode* node)
    if node is divisible then
        plane ← SplittingPlaneSelection(node)
        SubRegionGeneration(node, plane)
        NodeSplitting(node→left_ptr)
        NodeSplitting(node→right_ptr)
    end
end function

function TreeBuilding(TreeNode* root)
    Initialization(root)
    NodeSplitting(root)
end function
```

2.1.1. Cost Function

For any kd-tree subdivision, the key part is to define a reasonable cost function. Many cost functions have been proposed for building kd-trees in ray-tracing acceleration techniques. The most popular cost function for such techniques is given by the surface area heuristic (SAH) [MacDonald and Booth 90]. Its first purpose is to estimate ray intersection and traversal costs, which does not directly lead to empty-space detection. In particular, to enable SAH to favor splittings that remove more empty space, the cost function should be biased with a user-defined factor. This biasing factor can not be used to exactly determine which plane removes most empty voxels. Furthermore, it is difficult to extend SAH to handle volume data efficiently.

The key element of our method is the introduction of a simple cost function based on bounding volume computations which maximizes the empty space removed from the volume associated to the current node. We simply define the cost function C for the splitting candidate plane p in the node n as follows:

$$C(n,p) = BV(n_{left}(p)) + BV(n_{right}(p)) \tag{1}$$

where n_{left}, n_{right} are the two child nodes created by the plane p, and BV is a scalar function which calculates the bounding volume of the non-empty

voxels for a given node. Our goal now becomes finding the plane with the minimum cost value.

The proof that this cost function maximizes the empty space removed from n is straightforward: for any splitting plane, all the non-empty voxels are grouped into the two child nodes, which are stored for further subdivision. The volume of the empty space that can be removed from the current node, denoted as EV, is calculated by eqn. (2):

$$\begin{aligned} EV(n,p) &= BV(n) - (BV(n_{left}(p)) + BV(n_{right}(p))) \\ &= BV(n) - C(n,p) \end{aligned} \tag{2}$$

Therefore minimizing the cost function is equivalent to maximizing EV, that is, removing most empty space from the current node. A simple 2D example is presented in Fig. 2: based on our cost function, p_2 is a better choice than p_1 and p_3 for the current node.

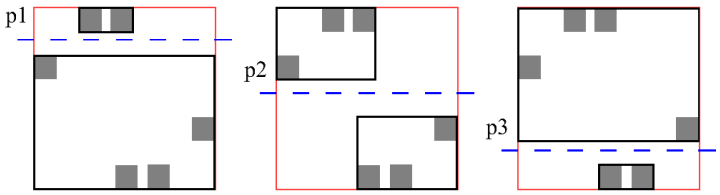

Figure 2. A 2D illustration for the cost function. The regions with non-empty voxels are depicted in gray. The bounding boxes for the original node and the two child nodes are marked with red and black respectively. The splitting planes are denoted as dotted lines.

We should point out that at each step we get a *local* optimum. If one wants to produce a kd-tree that removes the most empty space for a given tree depth, it is necessary to perform a global optimization that finds the splitting plane set $\{p_i\}$ that minimizes $\sum_i C(n_i, p_i)$. But this is a time-consuming process. In practice, even without global optimization, the algorithm is able to remove a significant amount of empty space with a low tree depth, as shown in Section 3. A limitation of the local-only approach is that it cannot remove the holes inside the object: The splitting planes needed to extract those holes are not locally favored by the cost function (for that purpose, a global optimization is required).

2.1.2. Fast Bounding Volume Computation

We can see from eqn. (1) that the cost function is based on the bounding volume (BV) computation of the two child nodes. Since the cost function needs to be computed for each candidate plane, a fast method to find the

bounding box of a given axis-aligned region would be necessary. For a region defined in the range $[i_L, i_R] \times [j_L, j_R] \times [k_L, k_R]$, our problem is to find the following boundaries along each axis respectively:

- $[i_1, i_2] \subset [i_L, i_R]$ along the X axis
- $[j_1, j_2] \subset [j_L, j_R]$ along the Y axis
- $[k_1, k_2] \subset [k_L, k_R]$ along the Z axis

such that any voxel that stays outside $[i_1, i_2] \times [j_1, j_2] \times [k_1, k_2]$ is empty.

One straightforward method is to scan all the non-empty voxels in the node and keep updating the boundary values during the scan. This method would be computationally expensive and impractical for large size volumes. [Tong et al. 99] employed min-max arrays to accelerate the boundary computation in a fixed direction, which is not suitable for our work since the min-max arrays need to be regenerated for each individual node of the tree.

We instead employ a summed-volume table for fast boundary computation. We first consider how to get the boundary value i_1: starting from i_L along the X axis, we increase i by 1 each step. i_1 is the first position that guarantees the region $R_i = [i_L, i] \times [j_L, j_R] \times [k_L, k_R]$ is not empty, which can be expressed as:

$$i_1 = \min\{i \mid i \in [i_L, i_R] \ \&\ R_i \text{ is not empty}\} \tag{3}$$

In the worst case, we need to test if R_i is empty or not for $i_R - i_L + 1$ times. Therefore if we can find a quick way to determine if an axis-aligned region of the volume is empty or not, the computation for i_1 can be greatly accelerated. The summed-volume table solves this problem in a simple way.

The summed-volume table, denoted as V_2, is a 3D array that is the same size as V_1. The value for the element (i, j, k) in V_2 is defined as

$$V_2(i, j, k) = \sum_{u=1}^{i} \sum_{v=1}^{j} \sum_{w=1}^{k} V_1(u, v, w) \tag{4}$$

The summed-volume table, first introduced in [Glassner 90], is a natural 3D extension of the summed-area table for 2D images [Crow 84] (see Figure 3). $V_2(i, j, k)$ records the total number of the non-empty voxels in region $[1, i] \times [1, j] \times [1, k]$. In practice V_2 is built within one pass through the 3D grid in a sequential way:

$$\begin{aligned} V_2(i, j, k) = \quad & V_1(i, j, k) + V_2(i, j, k-1) \\ & + (V_2(i-1, j, k) - V_2(i-1, j, k-1)) \\ & + (V_2(i, j-1, k) - V_2(i, j-1, k-1)) \\ & - (V_2(i-1, j-1, k) - V_2(i-1, j-1, k-1)) \end{aligned} \tag{5}$$

To compute $V_2(i,j,k)$, we need to fetch the current cell value $V_1(i,j,k)$ and seven previously computed V_2 values. For special cases where the index values $i-1$, $j-1$ and $k-1$ are not available, the corresponding V_2 values are set to zero. Note that eqn. (5) only requires the V_1 value for the current cell, which means V_2 can be built directly from V_1 in the same array. $V_2(i,j,k)$ is computed and stored at position (i,j,k), while the old $V_1(i,j,k)$ value can be discarded safely since it is no longer used for further computation. Therefore no new volume needs to be created for V_2. Once this table is built, the number of the non-empty voxels in an arbitrary axis-aligned region $R = [u_1+1, u_2] \times [v_1+1, v_2] \times [w_1+1, w_2]$ (denoted as $Num(R)$) can be computed with eight lookups in V_2:

$$\begin{aligned} Num(R) = \quad & (V_2(u_2,v_2,w_2) - V_2(u_2,v_2,w_1)) \\ & - (V_2(u_1,v_2,w_2) - V_2(u_1,v_2,w_1)) \\ & - (V_2(u_2,v_1,w_2) - V_2(u_2,v_1,w_1)) \\ & + (V_2(u_1,v_1,w_2) - V_2(u_1,v_1,w_1)) \end{aligned} \tag{6}$$

If $Num(R) = 0$, we know that region R is totally empty. A simple analysis would reveal that eqn. (5) is just a special case of eqn. (6).

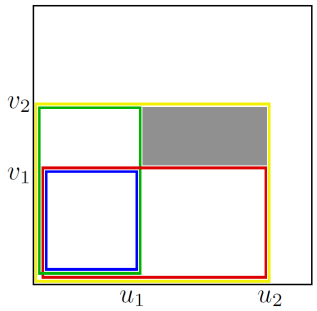

Figure 3. Illustration of the summed table in 2D [Crow 84]: *Sum* of the gray rectangle elements $= Sum(u_2,v_2)$ (elements outlined in yellow) $- Sum(u_1,v_2)$ (elements outlined in green) $- Sum(u_2,v_1)$ (elements outlined in red) $+ Sum(u_1,v_1)$ (elements outlined in blue). Eqn. (6) is the extension of this formula to 3D.

With eqn. (3) and (6), i_1 can be found in $O(i_R - i_L + 1)$ time cost. Boundary values i_2, j_1, j_2, k_1, k_2 are computed in a similar way, then BV is computed from these boundary values.

2.1.3. Splitting Plane Selection

With the cost function (1), the proper splitting plane can be approximated with a greedy algorithm: a set of candidate planes are generated uniformly

along one axis and the plane that gives the lowest cost is selected. To make sure that no node will be stretched too far in one direction, we choose to split along the longest axis of the bounding box. This also reduces the number of candidate planes since they are only generated along one axis instead of three. The number m of candidate planes depends on the length of the split axis l_{max} and the user-defined distance Δl between the planes: $m = \lfloor \frac{l_{max}}{\Delta l} \rfloor$. In practice, we set Δl to be 4 for 256^3 data sets and 8 for 512^3 data sets. Therefore the number of the candidate planes for each node would not exceed 64 in both cases.

2.1.4. Halting Criteria

In our algorithm, the node splitting process is recursively performed on the newly generated subregions until one of the following criteria is satisfied:

1. The node volume is smaller than some minimum threshold.
2. The best splitting plane can not remove enough empty space from the node.

For criteria 1, the minimum volume threshold needs to be adjusted for different volume resolutions. In practice we set this value to be 10% of the original volume. For criterion 2, we set the minimum percentage of empty space that should be removed from the node by the split plane to be 5%, which works well for all our examples. Fixing a maximum depth is necessary for a complete kd-tree building process. Since our algorithm normally stops at a reasonable depth ($4 \sim 7$ in the examples), we set this maximum depth to be a large value such as 10 so that it is seldom used for recursion termination.

2.2. Subvolume List Generation

Following the kd-tree building phase, we traverse the tree to get a list of the non-empty nodes. Each non-empty node contains an axis-aligned subvolume that encloses some regions of interest. The straightforward way is to traverse the tree to collect all the leaf nodes. Some small leaf nodes might enter the list without bringing much gain for the performance. It would be helpful for the users if the total number of the subvolumes, M, can be controlled and adjusted during the tree traversal process. Therefore we provide an alternative traversal scheme for the generation of the subvolume list. Starting from the root node, we want to produce at most M subvolumes. If $M = 1$, the root node is included in the list, and the scheme stops; if $M > 1$, we first compute the number of the leaf nodes M_l and M_r for its two child nodes. If $(M_l + M_r) \leq M$, all the leaf nodes are included in the list as the final results; otherwise the

maximum number of subvolumes that the two child nodes can produce should be limited to $M_l^{'} = \lceil M \cdot \frac{M_l}{(M_l + M_r)} \rceil$ and $M_r^{'} = \lceil M \cdot \frac{M_r}{(M_l + M_r)} \rceil$ respectively. The scheme then operates on two child nodes with the new limitation numbers $M_l^{'}$ and $M_r^{'}$ in a recursive way. The size of the subvolume list is successfully controlled in the scheme. In practice, different M values can be tested for various examples by the users to find a suitable subvolume list with enough empty space removing and good rendering efficiency.

For adjacent subvolumes, their AABB computation on the common boundary faces is adjusted to be consistent with each other to avoid discontinuous artifacts during rendering. In the volume rendering stage we render the subvolumes using a slice-based method. At this stage, raycasting may also be used; the kd-tree structure is then used to accelerate the intersection computations of the rays with the non-empty subvolumes.

2.3. Algorithm Summary

The whole preprocessing algorithm can now be summarized in the following four steps:

1. Build the binary volume V_1.
2. Build the summed-volume table V_2 from V_1.
3. Build the kd-tree T.
4. Generate the subvolume list by traversing T.

Given the volume resolution l_x, l_y, l_z, we briefly discuss the running time cost of each step. Without losing generality, we assume $l_x = l_y = l_z = l$. For step 1 and 2, V_1 and V_2 are both built in $O(l^3)$. For step 3, the splitting process for the root node is considered. The number of the plane candidates is proportional to l. For each candidate plane, the bounding volumes of its left and right child nodes are computed in $O(l)$ with the summed-volume table V_2. Therefore the proper candidate plane for the root node is found in $O(l^2)$. If the depth of the generated kd-tree is denoted as d_{max}, the computation cost for building the whole kd-tree would not exceed $O(2^{d_{max}} \cdot l^2)$. Then in step 4, the subvolume list can be generated in a $O(2^{d_{max}})$ tree traversal. We can see that the overall time cost for the algorithm is mostly determined by the $O(l^3)$ step 1, 2 and the $O(2^{d_{max}} \cdot l^2)$ step 3. Since the kd-tree building process naturally stops at an early stage, the tree depth d_{max} is usually low ($4 \sim 7$ in our examples), which means step 1 ,2 are more computationally expensive than step 3, especially for large size volumes. Therefore the algorithm is basically a $O(l^3)$ process. Finally we present the total memory cost of the algorithm:

only one copy of the original volume is maintained for V_1 and the subsequent V_2, as shown in Section 2.1. The memory usage for the dynamical kd-tree and the subvolume list is related to the tree depth d_{max} and the halting criteria, which is negligible compared to the summed-volume table.

3. Examples

Dataset	Preprocessing time (seconds)		Level of subdivision 1				Level of subdivision 2				Level of subdivision 3			
			# Subvolumes	Space removed	Rendering time (fps)		# Subvolumes	Space removed	Rendering time (fps)		# Subvolumes	Space removed	Rendering time (fps)	
	256^3	512^3			256^3	512^3			256^3	512^3			256^3	512^3
spruce (A)	0.52	4.18	1	0%	88	30	2	29%	103	35	8	48%	128	42
teapot (B)	0.53	4.25	1	0%	85	28	3	34%	103	35	13	52%	177	59
shrub (C)	0.57	4.57	1	0%	88	29	2	19%	98	34	10	52%	137	47
knot (D)	0.49	3.99	1	0%	98	34	3	24%	104	37	10	50%	133	41
palm (E)	0.46	3.75	1	0%	101	35	2	67%	338	112	6	74%	401	133
skull (F)	0.55	4.42	1	0%	91	31	4	22%	119	43	7	27%	127	45
eiffel (G)	0.51	4.11	1	0%	105	37	2	63%	222	74	7	75%	349	112
charact. (H)	0.55	4.48	1	0%	100	35	2	54%	240	79	4	76%	464	158

Table 1. Performance results for the examples with increasing number of subvolumes as shown on Figures 1, 4 and 5. For each example, we test on two grid resolutions: 256^3 and 512^3. The number of the subvolumes, the percentage of the empty space removed from the intial volume, the average rendering time (in frames per second) from six orthogonal viewpoints are recorded with a screen resolution of 800×600.

We applied our preprocessing method on several examples as shown in Figures 1, 4 and 5. Our initial aim was to accelerate the interactive rendering of plants and trees represented by the volume data, such as models A, C and E. Many empty regions exist in this kind of volume data, especially around the trunks of the trees. We also applied the method on some more general shapes (models B, D, F, G and H). Our test platform is a Pentium IV 2.40GHz PC equipped with a NVIDIA GeForce 8800 GTX graphics card. For each example, we test the data set on two different grid resolutions: 256^3 and 512^3. We record the following performance results with an increasing number of the subvolumes for comparison: the preprocessing time (for steps 1, 2 and 3), the percentage of the empty space removed from the volume, and the average rendering time from six orthogonal view directions with a slice-based method. The screen resolution is 800×600. Note that for each example, the numbers

of the subvolumes are adjusted respectively to produce satisfactory results (following the method described in Section 2.2). We do not provide the time spent on subvolume list generation since it is negligible compared to the tree building process. The results are presented in Table 1.

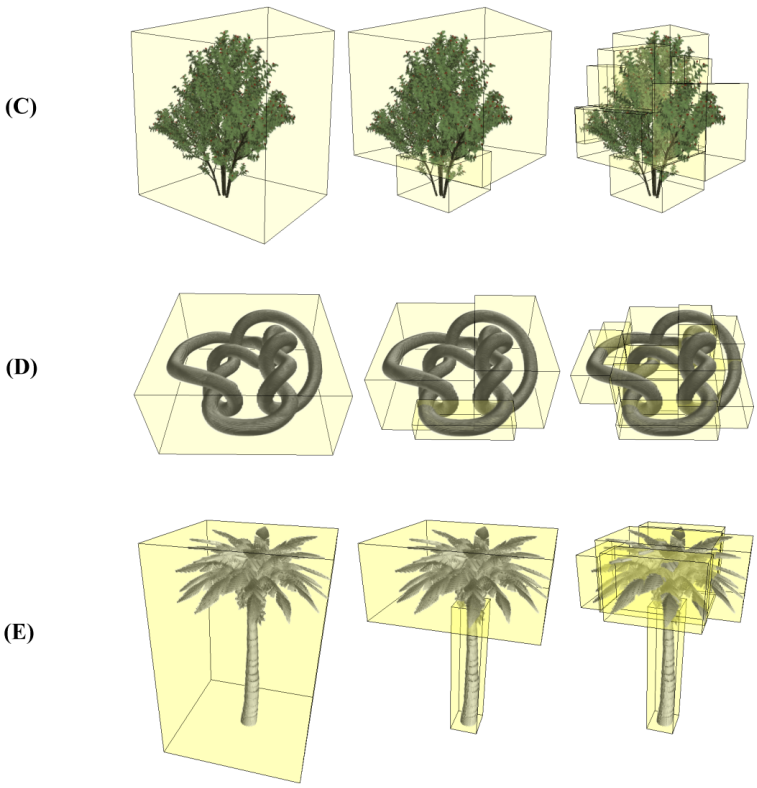

Figure 4. Display of the subvolume list of data sets C to E in Table 1. See also Figures 1 and 5.

Despite its simplicity, our algorithm performs well on all the examples: It removes significant empty space from the original volume with just a small number of subvolumes (less than 16). We can see from these examples that the subvolumes divide the initial object in an even manner: After the first building steps, compact non-empty regions are embedded directly in a small number of large subvolumes which are then naturally refined in later steps. Thereby we avoid a common shortcoming of partitioning algorithms which is to produce in the first steps some regions of small size that need to be brought together to reduce the number of subvolumes that fit the non-empty spaces.

We can also observe that the rendering speed is significantly improved,

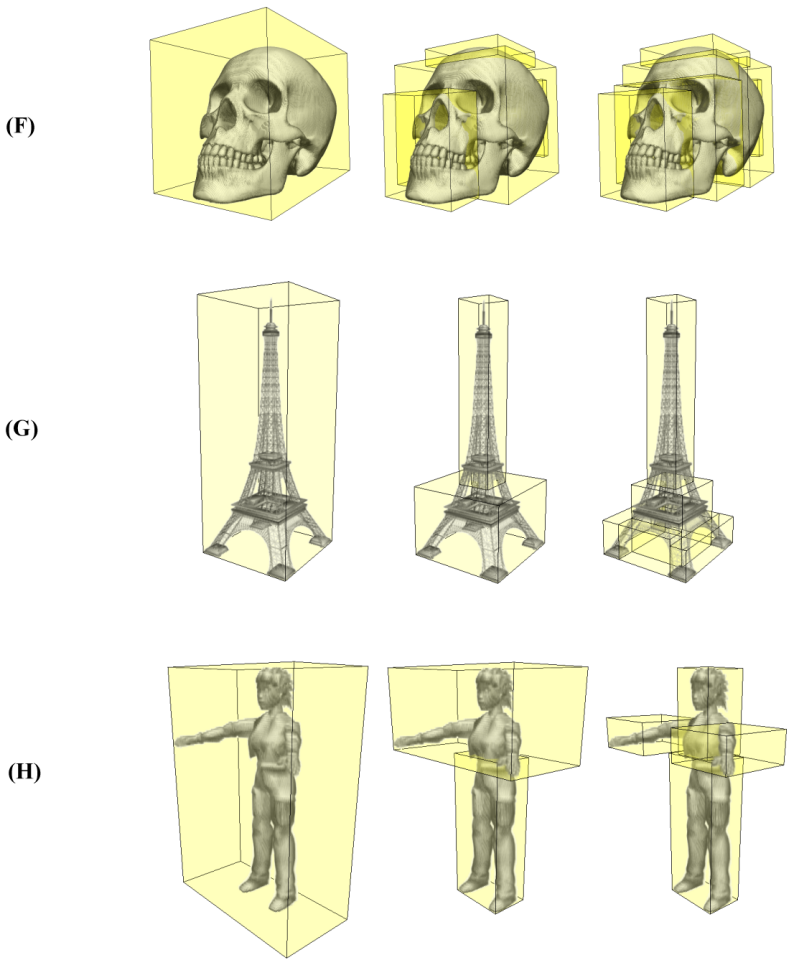

Figure 5. Display of the subvolume list of data sets F to H in Table 1. See also Figures 1 and 4.

especially for models E, G and H. Furthermore, it requires little preprocessing time for our examples: about half a second for 256^3 data sets, and about 4 seconds for 512^3 data sets, which is consistent with our algorithm analysis.

4. Discussion

As stated in the introduction, in this paper we consider static volumes with a fixed transfer function. In the examples presented in the previous section, the

volumes are defined by 3D textures that store the color and opacity of each voxel, and they are rendered directly with a slice-based algorithm (no transfer function is applied). This can be seen as a limitation since interactive transfer function editing is often required by volume visualization applications. Since the preprocessing time of our method is quite low (especially for volumes of resolution less than or equal to 256^3), an option to alleviate this limitation can be to recompute the subvolume list when significant changes to the transfer function occur. If the volume resolution is too large to allow fast update of the non-empty subvolumes list, one could rebuild this list using a filtered version of the volume at a lower resolution in order to get a good approximation of the list. The exact list can then be computed from the full resolution volume when transfer function editing stops.

Another limitation, already mentioned in Section 2.1.1, is that our algorithm cannot remove the empty space inside a voxelized object, since the splitting rule is based on a direct estimation of outer bounding volume which cannot capture the holes in the object. For those objects with considerable internal empty space such as the skull model (F) in Table 1, the speedup is not significant since the internal volume still needs to be sampled and rendered. For fuzzy objects like trees, this is not a serious limitation. However, a global optimization would be necessary for the generation of the kd-tree if one needs to alleviate this problem.

For future improvement we could consider reducing the size of the 3D texture with the generated subvolumes following the texture packing algorithm in [Kähler et al. 03]. And we could also consider applying our method to some larger size data sets, which may be initially divided into several bricks to fit into the GPU memory.

Finally we note that our method (except the fast bounding volume computation part) can be extended to build kd-trees that isolate empty regions for 3D meshes. In this case, the bounding volumes are estimated by accumulating the AABBs of the mesh faces.

Acknowledgments

The authors would like to thank Jamie Wither and Laks Raghupathi for proofreading. We would also like to thank the anonymous reviewers for their valuable comments. Philippe Decaudin is supported by a grant from the Marie-Curie project REVPE MOIF-CT-2006-22230 from the European Community. Xing Mei is supported by LIAMA NSFC 60073007 and the Marie-Curie project VISITOR MEST-CT-2004-8270 from the European Community.

References

[Cabral et al. 94] Brian Cabral, Nancy Cam, and Jim Foran. "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware." In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pp. 91–98, 1994.

[Crow 84] Franklin C. Crow. "Summed-area tables for texture mapping." In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 207–212, 1984.

[Engel and Ertl 02] K. Engel and T. Ertl. "Interactive high-quality volume rendering with flexible consumer graphics hardware." Eurographics '02, State of the Art Report, 2002.

[Glassner 90] Andrew S. Glassner. "Multidimensional sum tables." In *Graphics Gems*, pp. 376–381. Academic Press, 1990.

[Hadwiger et al. 06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-Time Volume Graphics*. A. K. Peters, 2006.

[Kähler et al. 03] Ralf Kähler, Mark Simon, and Hans-Christian Hege. "Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies." *IEEE Transactions on Visualization and Computer Graphics* 9:3 (2003), 341–351.

[Kruger and Westermann 03] J. Kruger and R. Westermann. "Acceleration Techniques for GPU-based Volume Rendering." In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 Conference*, pp. 38–42, 2003.

[LaMar et al. 99] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. "Multiresolution Techniques for Interactive Texture-based Volume Visualization." In *VIS '99: Proceedings of the 10th IEEE Visualization 1999 Conference*, pp. 355–362, 1999.

[Levoy 90] Marc Levoy. "Efficient ray tracing of volume data." *ACM Trans. Graph.* 9:3 (1990), 245–261.

[Li and Kaufman 03] Wei Li and Arie Kaufman. "Texture Partitioning and Packing for Accelerating Texture-based Volume Rendering." In *GI '03: Graphics Interface*, pp. 81–88, 2003.

[Li et al. 03] Wei Li, Klaus Mueller, and Arie Kaufman. "Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering." In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 Conference*, pp. 317–324, 2003.

[MacDonald and Booth 90] David J. MacDonald and Kellogg S. Booth. "Heuristics for ray tracing using space subdivision." *The Visual Computer* 6:3 (1990), 153–166.

[Stegmaier et al. 05] Simon Stegmaier, Magnus Strengert, Thomas Klein, and Thomas Ertl. "A simple and flexible volume rendering framework for graphics-hardware-based raycasting." In *Fourth International Workshop on Volume Graphics*, pp. 187–241, 2005.

[Subramanian and Fussell 90] K. R. Subramanian and Donald S. Fussell. "Applying space subdivision techniques to volume rendering." In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pp. 150–159, 1990.

[Tong et al. 99] Xin Tong, Wenping Wang, Waiwan Tsang, and Zesheng Tang. "Efficiently Rendering Large Volume Data Using Texture Mapping Hardware." In *Proceedings of Joint Eurographics/IEEE TVCG Symposisum on Visualization*, 1999.

Information:

Vincent Vidal, INRIA-Evasion, Inovallée 655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
(vidal.vince@gmail.com)

Xing Mei, CASIA-NLPR/LIAMA, 95 Zhongguancun East Road, Beijing 2728# P.O. 100080, China
(xmei@nlpr.ia.ac.cn)

Philippe Decaudin, INRIA-Evasion, Inovallée 655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
(http://www.antisphere.com)

Received November 14, 2007; accepted in revised form June 4, 2008.