

Gradient Octrees: A new Scheme for Remote Interactive Exploration of Volume Models

Lazaro Campoalegre, Isabel Navazo, Pere Brunet

Research Group on Modelling, Visualization, Interaction and Virtual Reality, MOVING
Universtat Politècnica de Catalunya
Barcelona, Spain
lazaroc@lsi.upc.edu

Abstract—Remote exploration of medical volume models is nowadays a challenging problem. Interactive visualization algorithms must be able to send and render in real-time the volume model at the maximum possible visual quality, while adapting to network bandwidth limitations and to hardware constraints in the client device. In this paper we present a transfer function-aware scheme for the remote interactive inspection of volume models in client-server architectures with the objectives of supporting multi-resolution, avoiding gradient computations in the client device and sending a very limited amount of information through the network. Gradient Octrees can be progressively transmitted to the clients in a strongly compact way while achieving a minimum loss of visual quality as compared to state of the art ray-casting renderings. Visual volume understanding can be complemented by showing 2D sections of the original volume data on demand. The approach is discussed and compared to standard ray-casting results on several examples.

Keywords—Volume Visualization; Volume Compression; Progressive Transmission; Octrees;

I. INTRODUCTION

Present systems for the acquisition of medical images are providing better and better resolution images which end up in bigger amounts of data. However, hospitals are now more interested in tele-medicine and tele-diagnostic solutions. Requirements for medical clinical sessions and new tele-medicine applications are usually including low-end (and sometimes mobile) visualization and interaction devices. In this context, the main present challenge in remote exploration of medical volume models can be formulated as a constrained optimization problem: visualization algorithms must be able to send and render in real-time the volume model at the maximum possible visual quality, while adapting to network bandwidth limitations and to memory and hardware constraints in the client device. Although techniques like bricking, multiresolution or compression have been proposed, efficient rendering of large volumes is still a challenging problem, [1]

Two main approaches have been proposed for visualizing volume medical models in low-end devices. The first one, inspired on thin-client architectures, uses a remote server

to render the model and to stream the images to the client through an internet connexion, [2] [3]. The second approach uses graphic capabilities of the device clients in order to render the volume model in client GPUs. Streaming rendered images can become too slow in cases when too many client requirements compete, while sending the volume model to the clients can be also too slow and overflow the client memory storage. Sending a compressed volume model to the client can be a good compromise, as discussed by a number of recent papers. However, this is a problem with conflicting requirements. Server-client model transmission requires good compression techniques, being as lossless as possible and having a high compression rate, to obtain a reduced data flow through the network. Progressive transmission of multi-resolution volumes is also essential, to produce approximate representations of the volume in the client from the very beginning in networks with limited bandwidths. Finally, decompression should be ideally performed in the GPU client in an efficient way and with a minimum loss of visual quality. To the best of our knowledge, no present approach succeeds in fulfilling all these requirements, and there is room for improvements.

In this paper, we present a transfer function-aware scheme for remote interactive inspection and expressive visualization of volume models in client-server architectures with the objective of supporting multi-resolution, avoiding gradient computations in the client device and sending a very limited amount of information through the network, see Figure 1. The main contributions of our approach are:

- An octree, multi-resolution volume data representation for client-server architectures, based on a predefined set of transfer functions.
- A GPU-oriented encoding of the proposed hierarchical data structure with explicit volume gradient information in octree nodes, to avoid gradient computations during GPU ray-casting.
- An efficient compression algorithm which encodes gradients and material properties, together with a data transmission scheme which is able to progressively

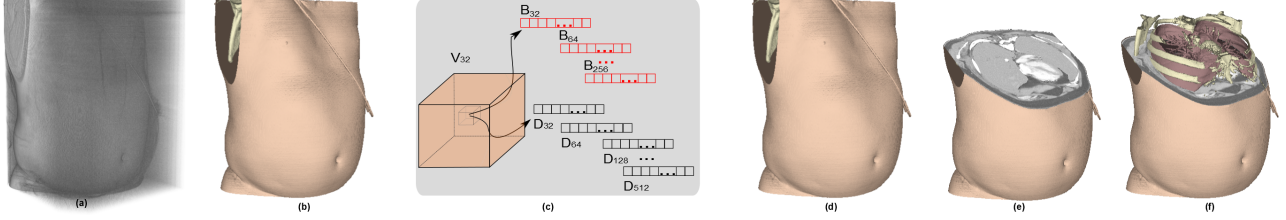


Figure 1. (a) We start from an initial volume model V and a set of transfer functions S , to create a (b) segmented volume model $V_E(S)$. Gradient Octrees (c) are multi-resolution representations of $V_E(S)$ which can be progressively sent to client devices in a very compact way. Clients are able to reconstruct a copy of the Gradient Tree and ray-cast it in their GPU to interactively inspect volume structures, (d). Clients can perform planar sections (e) and render them with high-res information from the initial volume V , even adding offset structures in front of the section plane (f).

send deeper octree levels with a minimum amount of octree structure information (one bit per son node). A recovering algorithm that succeeds in reconstructing a full copy of the octree in the client from the received information.

The paper is organized as follows. After reviewing the previous work in next Section, Section III presents an overview of the proposed approach and algorithms. Sections IV and V are devoted to the generation of the Gradient Octree in the server and to the rendering algorithms and interaction facilities in the client. Finally, Section VI presents and discusses a number of examples while showing the main interaction paradigms and comparing with standard ray-casting renderings. Conclusions and future work are listed in Section VII.

II. PREVIOUS WORK

With the improvement of capture devices, medical volume models have grown continuously during the last years. The amount of memory of modern GPUs is also growing, but unfortunately the increasing rate of the size of volumetric datasets is even much higher. Although techniques like bricking, multiresolution or compression have been proposed, the efficient rendering of large volumes is still challenging [4], [5]. This fact is especially true when the applications must be run in devices with low performance GPUs, transmission of high volume datasets both through the network and (in any case) between the CPU and the GPU of the client being also a bottleneck.

Bricking techniques subdivide large volumes into several *blocks*, named *bricks*, in such a way each block fits into GPU. Bricks are stored in main memory, then they are sorted either in front-to-back or back-to-front order respect to the camera position, depending on the rendering algorithm [4], [6]. A second approach is the use of -hierarchical-multiresolution models [7], [8], [9]. Both techniques may require a significant amount of texture transfers. In [10], an adaptive out-of-core technique for rendering massive scalar datasets within a single-pass GPU raycasting framework is presented. It however requires a significant memory cost and

GPU performance. Note that the whole model should be transmitted through the network.

Compression techniques consist on decreasing the size of volumetric data by applying a specific encoding algorithm [1]. In some approaches compression is combined with a brick or a hierarchical partition of the original volume to facilitate the compression process and a level-of-detail rendering/transmission. The compact representation can be loss-less or lossy. Medical applications require a loss-less compression when rendering models at their highest resolution. We briefly comment the compression techniques more closely related with our approach and we refer the reader to [1] for a detailed coverage of this topic.

Vector quantification is a lossy compression method [11]. The volume dataset is represented as indexes into a small codebook of representative blocks. This scheme is suitable for a CPU-based ray-casting render. Schneider and Westermann [12] proposed a Laplacian pyramid vector quantification approach that allows relatively fast volume decompression and render on the GPU. However this method does not support GPU linear filtering capabilities and its rendering cost increases when using high zoom factors.

Wavelet transforms offer considerable compression ratios in homogeneous regions of a volume while they conserve the detail in the non-uniform ones [13]. Ihm and Park [14] proposed an effective 3D $16 \times 16 \times 16$ block-based compression/decompression wavelet scheme. Guthe *et al.* [15] proposed an alternative hierarchical wavelet representation which allows to store very large volume datasets in compact -lossy- data structures.

Octree-based, GPU-friendly ray-casting algorithms have been proposed in [5], [9] and others. In [5] the octree leaves represent bricks containing 16^3 or 32^3 voxels, while octree nodes in [9] contain 64-bit child descriptors, both being too verbose for progressive transmission purposes. More recently, Suter *et al.* [16] have proposed a multiscale volume representation based on a tensor approximation within a GPU-accelerated rendering framework. It can be better than wavelets at capturing non-axis aligned features at different scales. Gobbetti *et al.* [17] have proposed a different multi-

resolution compression approach using sparse representation of voxel blocks based on a learned dictionary. Both approaches obtain good compression ratios, but they are lossy and require a heavy pre-process. In contrast, Gradient Octrees are compact, progressive transmission is efficient and they succeed in producing good visualizations in low-end clients.

III. OVERVIEW OF THE PROPOSED APPROACH

Our main objective is to compress an initial volume model V , send it to the client device, and generate visualizations in this client device which have good visual quality when compared with Direct Volume Rendering (DVR) of V see 1. More specifically, our goal is to design a hierarchical, multi-resolution volume data structure supporting compact progressive transmission to the clients together with decompression and high-quality rendering in the client GPU. We start from the observation that in the clinical practice DVR visualizations are mostly based on well established transfer functions which enhance specific tissues and organs. Transfer functions are able to assign a color $c(p)$ and an opacity value $o(p)$ to any point p in V according to its properties. Transfer functions (TF 's in what follows) can be based on the density value of the scalar field V [18] or they can be functions of densities, gradient values and other attributes [18]. Our compression scheme is based on a predefined set S of transfer functions, $S = \{T_1, T_2, \dots, T_n\}$, each T_k in S being able to classify a certain specific material in V . In our approach, the goal is to minimize the loss of information and quality when using transfer functions from the set S , at the cost of having removed volume information that could have been relevant for TF 's not in S . Our approach is lossy in a selective way, as it tries to maximize the visual quality when using any of the transfer functions in S .

Given a volume model V (Figure 2-a), let us consider the segmentation induced by each transfer function T_k in S , Figure 2-b. The segmented region V_R^k (Figure 2-c) is defined by assigning all volume points p having an opacity $o(p) > t$ to the region R_k corresponding to T_k , where t is a given threshold. Alternatively, any other segmentation algorithm could be used. In V_R^k , voxels can be classified as *Uniform* voxels or *Edge* voxels. A voxel in V_R is *Uniform* **iff** its eight vertices belong to the same segmented region R_k . A voxel in V_R is an *Edge* voxel **iff** its eight vertices come from two different regions R_j and R_k .

Note that *Uniform* voxels in V_R^k are redundant, as V_R^k can be reconstructed from its set of *Edge* voxels. We therefore can define a compact auxiliary representation for V_R^k , named *Edge Volume* V_E^k (Figure 2-d). Uniform voxels in V_R^k are stored as Nil voxels in V_E^k , while Edge voxels contain an index to a look-up table storing attribute information of the materials pair (R_j, R_k) , plus the normalized gradient of V at the voxel center. We store gradients in order to avoid extra

computations in the client GPU. V_E^k is a lossy compressed representation of V based on T_k . Given a volume V and a set S , its S -based representation $V_E(S)$ can be defined (Figure 2, d and e) as the union of all V_E^k :

$$V_E(S) = V_E^1 \cup V_E^2 \cup \dots \cup V_E^n$$

This union operation is performed in a voxel per voxel basis. Union between two Nil voxels is Nil, whereas the union between a Nil and an edge voxel is an Edge voxel. An attempt to perform a union between two Edge voxels shows that V contains structures which are too narrow for the current voxel resolution. In this case, the resolution of the voxel model $V_E(S)$ should be increased.

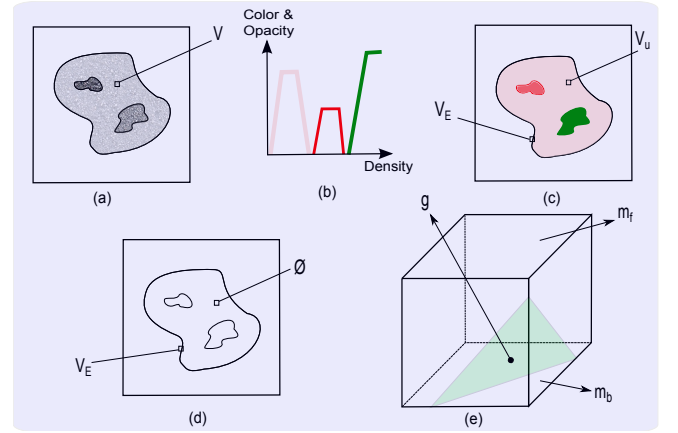


Figure 2. In (a), the initial volume model V . In (b) the set S of transfer functions which generate the segmented volume in (c). The Edge Volume model $V_E(S)$ (with an Edge and a Nil voxel) is presented in (d), whereas (e) shows the gradient and materials stored in an Edge voxel of $V_E(S)$ (a green triangle is shown for figure clarification).

Our approach starts with a preprocess stage in the server where the already described volume representation $V_E(S)$ is generated. Then, a compact Gradient Octree is created using V , $V_E(S)$ and the predefined set S , as detailed in Section IV. Then, it is progressively sent to the client as a sequence of arrays with compressed information. The client enlarges the received information in order to reconstruct a local copy of the Gradient Octree. Gradient decompression and ray-casting rendering is directly performed in the client GPU, as described in Section V. Interaction in the client drives its GPU visualization, sends queries to the server to request further levels of the Gradient Octree and sends planar section queries when interactive cuts of the volume must be filled out with original volume information to fully understand the original data in V . In this last case, the sections server generates a texture with the appropriate sliced information from V , and sends it to the client. Section V also details how these section textures are handled in the GPU. Interaction is autonomous and no further information is needed from the server, except when a new planar section is required.

IV. THE GRADIENT OCTREE

The Gradient Octree is a novel compact multi-resolution representation of $V_E(S)$. Its name comes from the fact that its Grey nodes contain information on the volume gradient at different resolutions. It is based on a standard octree space subdivision, being computed in a bottom-up way. The depth of the Gradient Tree is defined by the resolution of its *Edge* Volume model $V_E(S)$, the leaves of the Gradient Octree being voxels of $V_E(S)$. Nodes in the Gradient Octree can be either Grey or Nil. Grey nodes store gradient and material information, encoding 3D cubical regions that contains edge voxels of $V_E(S)$. Nil nodes represent 3D cubical regions which only contain Nil $V_E(S)$ voxels. Node information in any upper level of the tree is computed as the result of applying a simplification downsampling function to its eight son nodes.

The downsampling function must consider two different cases. Nodes in the upper level having only Nil son nodes are assigned a Nil node type. Otherwise, when at least one son node is not Nil, the gradient of the parent node is computed by applying a *SetGradient* function to the gradients of its non-Nil son nodes. One option for *SetGradient* could be to average the gradients of the son nodes, but this algorithm produces artifacts in upper tree levels when region boundaries with almost opposite gradients are merged. In our implementation, we concluded that the best option was to use a *selection* algorithm. We estimate an importance weight for each son gradient vector $g.s1, \dots, g.sn$, and then copy the most important gradient to its parent's gradient. This is then repeated for the material information: we copy the materials pair of the most important son to its parent. After several experiments, we concluded that importance weights could be estimated in a suitable way as the width of the visibility cone of the node cube when observed from different view directions. As the exact computation of visibility cones may be slow, we have finally used a more simple heuristics: we compute the average of the son gradient vectors after removing outlier sons (outliers are sons being too different from their average) and estimate importances by the scalar product between the node's gradient and this average.

A Gradient Octree of depth d can be represented as d different lists of nodes, one of them per each tree level. Any octree level l is represented as a list of its Grey nodes. Grey nodes contain their gradient vector, an index to the materials pair in the materials look-up table and eight indexes to their son nodes. Indexes to Nil son nodes are set to zero, while indexes to Grey son nodes point to their location (component) in the list of the next octree level. The cardinality M_l of the level l list is exactly the number of Grey nodes at this octree level l . For the sake of simplicity, we represent the Grey node list corresponding to any tree level l by two separate arrays with M_l elements each: the octree array O_l and the data array D_l . O_l stores indexes

to the eight son nodes, while D_l stores node gradient and materials index.

However, lists at coarser levels of the tree are too small and it is wise not to store them. The final Gradient Octrees representation is shown in Figure 3 and consists on a small volume model V_{32} and two sets of per-level arrays, O_l and D_l (arrays B_l , in red in Figure 3, will be described later). For the sake of clarity, octree levels in Figure 3 and in what follows will be identified by their resolution. The example shown in Figure 3 corresponds to the complete octree representation of a volume V of resolution $R = 512$, with gradient and materials stored in the data array D_{512} . Data arrays of coarser octree levels (D_{256} , D_{128} , D_{64} and D_{32}) store gradient and materials data of Grey octree nodes at these levels. But, instead of storing data at coarser levels than D_{32} , we directly address D_{32} from the $32 \times 32 \times 32$ volume array V_{32} . V_{32} elements are either Nil (if the corresponding Grey node level 32 is Nil) or contain a two-byte index i_{32} to the data of this Grey node in the D_{32} array. This index i_{32} also identifies the component of O_{32} which stores the information (indexes) to access the data of its sons in D_{64} and O_{64} , Figure 3. Pruning Gradient Trees at resolution 32 saves texture addressing steps in the client GPU at a very limited overhead, the size of V_{32} being only $32 \times 32 \times 32 \times 2 = 65$ Kbytes. In other words, we will start by sending V_{32} and D_{32} to the client, which represent a volume of resolution 32^3 that can already be ray-casted. Having a low-resolution volume model V_{32} in the client GPU will also facilitate ray traversal at rendering time, as shown in Sections V and VI. To refine the octree one further level up to resolution 64, arrays O_{32} and D_{64} must be sent to the client. We use a two-byte encoding for O_l indexes when the number of grey nodes in D_{2l} is below 2^{16} , and a three-byte encoding otherwise. This process must be repeated for all subsequent levels, the deepest level in the example in Figure 3 being reached when O_{256} and D_{512} are sent to the client.

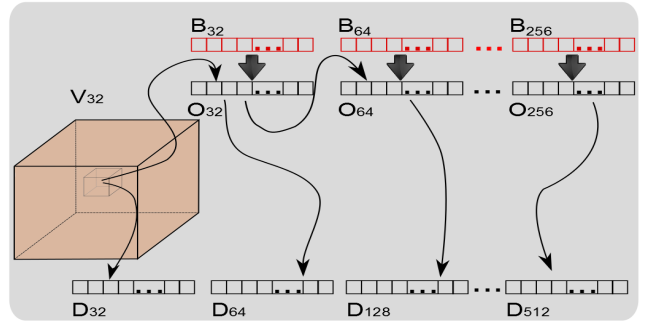


Figure 3. Encoding Gradient Octrees on a compact volume V_{32} , a set of index arrays O_l and a multi-resolution set of node data arrays D_l . O_l octree arrays are not send to the client. They are reconstructed from the more compact set of arrays B_l with one bit/node.

As previously explained, Gradient Octrees are generated in the server and must be transmitted to the clients. However, and for transmission purposes, it is possible to compact the O_l arrays of the Gradient Octree in a lossless way. Instead of sending the arrays $O_{32}, \dots, O_{256}$, we send a set of arrays $B_{32}, \dots, B_{256}$ with one byte per component as shown in red in Figure 3. The components in B_l simply represent the type of each son in one bit (0 if Nil, 1 if Grey). Note that this is equivalent to store and transmit only one bit per node. We compute the compressed arrays B_l in the server, send them to the client, and reconstruct again the set of octree arrays O_l in the client from the information in B_l . Let us represent the j -index to D stored in the $O_l[i]$ element as $O_l[i][j]$. The algorithm is based on the following invariant, known by both server and client: taking into account that indexes in $O_l[i][j]$ ($i=1..M_l, j=0..7$) are either nil or relevant indexes to son nodes, the subsequence of relevant indexes is the sequence of Natural numbers $\{1, 2, \dots, M_{l+1}\}$ when $O_l[i][j]$ is traversed through an outer i loop and an inner j loop. Then, in the encoding step and for the i -Grey element in the list represented in O_l , the j -bit of the component $B_l[i]$ is one **iff** $O_l[i][j]$ is relevant, and it is zero **iff** $O_l[i][j]$ is nil. The client can reconstruct O_l from B_l by simply traversing all bits in B_l and counting. First, all $O_l[i][j]$ are initialized to Nil. Then, every 1 in the j -bit of $B_l[i]$ is translated into an increment of one in a global counter of Grey nodes which is then assigned to $O_l[i][j]$. Instead of sending indexes, we simply send bits.

Moreover, for transmission and storage purposes, Grey node data (gradient vectors and materials) is compressed to 3 bytes. Consequently, D_l arrays contain elements of 3 bytes each. Gradient vectors are lossily encoded by their dominant orientation and by the quantized value of the two remaining coordinates. Let's consider a cube centered at the origin, having a 512×512 grid in each of its six faces. Given a gradient vector, it stabs a grid cell in one of the cube faces. We encode the gradient by storing the index (0, ..., 5) of the stabbed face and the two coordinates of the grid cell in the cube face. As these two coordinates require $9 + 9$ bits, we use the remaining 6 bits to store the cube face and the node materials as a single value $v = f + 6 * m$ where f is the index of the face and m is an index to a look-up table of (up to ten) material pairs. We could consider a much higher variety of transfer functions and material pairs by increasing from three to four the number of bytes per element in the D_l arrays, but we have experimentally found that supporting 10 different boundaries between segmented regions is reasonable in most practical cases.

V. RAY-CASTING AND INTERACTION IN THE CLIENT

Progressive transmission of the Gradient Octree to the client device consists on initially sending the volume V_{32} , its data array D_{32} and the materials look-up table, followed by the transmission of a progressive sequence of array pairs B_l ,

D_{2l} representing individual levels of the octree for $l=32, 64, 128, \dots, R/2$ where R is the resolution of $V_E(S)$. For every received level, the client generates a B_l -driven, increasing sequence of indexes to create a local copy of the array of indexes to son nodes, O_l . Note that O_l indexes point simultaneously to D_{2l} and O_{2l} . The volume V_{32} is sent to the GPU as a 3D texture, whereas arrays O_l and D_l are encoded as 1D textures.

We use a standard ray casting algorithm in the client GPU, the only difference with classical algorithms being that rays traverse a virtual volume model with the same resolution R as $V_E(S)$, instead of traversing $V_E(S)$ itself. In Figure 4, an example with resolution $R = 512$ is shown. Ray-casting proceeds as usual by advancing along rays r from the observer with a uniform sampling of the volume along r . Then, for each sample s of r addressing a virtual voxel (i, j, k) of $V_E(S)$, its volume information is found in D_l . Our approach is based on known octree addressing

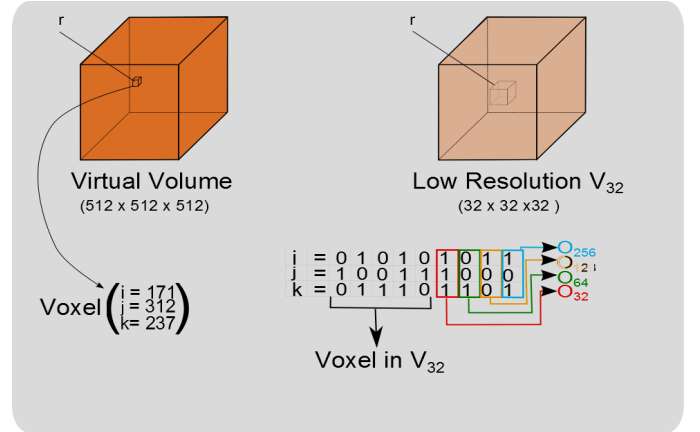


Figure 4. The virtual volume (left) and the low-res volume texture V_{32} (right), as used by the ray-casting algorithm in the client GPU. Octree addressing is based on the binary representation of voxel coordinates in the virtual volume.

properties. The search of any initial voxel (i, j, k) is directly driven by the base-2 representation of i, j and k , as shown in Figure 4. In our case, their first 5 binary digits point to the corresponding voxel in the low-res texture V_{32} . The index i_{32} found in this V_{32} voxel element points to the low resolution data in D_{32} (which we don't use if a higher resolution is required) and also to the array of its eight son indexes in O_{32} . A well-known property of binary octree subdivision ensures that next "three bit columns" s_l in the binary representation of i, j, k are in fact son indexes. Subtree traversal from the low-res voxel in V_{32} to the virtual voxel data is based on the recursion equation,

$$i_{2l} = O_l[i_l][s_l]$$

for $l=32, 64, 128, \dots, R/2$

Son indexes point to deeper octree levels. The final index i_{512} points to the high-res data in D_{512} , but tree traversal can stop earlier if the virtual voxel is void and any index i_l in the chain is found to be zero.

Let's assume that ray r is crossing voxel $i = 171$, $j = 312$, $k = 237$ in the virtual volume, Figure 4-left. In this case, Figure 4-right, the octree search starts in a voxel of V_{32} and is then driven by four son indexes: $s_{32} = 7$, $s_{64} = 1$, $s_{128} = 4$ and $s_{256} = 5$ which recursively generate the indexes i_{64} , i_{128} , i_{256} and i_{512} . Reaching the deepest level information in a Gradient Octree of resolution $R = 512$ involves a maximum of six texture queries, to V_{32} , O_{32} , O_{64} , O_{128} , O_{256} and finally to D_{512} .

After retrieving high-res data in D_{512} , materials and the gradient vector are decompressed on the fly in the GPU. The gradient face f , the two grid-coordinates of the gradient and the material index are decoded through simple bitwise operations and by one division by 6 to decouple f and the material index. Final gradient decompression involves specific computations for each of the six possible values of f , where the coordinate in the direction of the f normal vector is assigned a value 1 or -1 and the other two coordinates are linearly dependent on the coordinates of the grid cell within the cube face, followed by gradient normalization. Obviously, everything also works when lower resolution virtual volumes are considered.

Interaction in the client is efficient and does not depend on further communication with the server, except when new planar sections are requested. Users can play with the camera by rotating around the volume and zoom in. They can also render the volume at different resolutions, if necessary.

Planar sections are managed by the ray-casting algorithm in a straightforward way. Section textures T of the original volume V are sent to the client, on request. The GPU shader receives the normalized section plane equation $ax + by + cz + d = 0$ as four uniforms (a, b, c, d) , together with the texture T . The alpha channel of texels which correspond to external air is set to zero, to ensure transparency through air in external areas, as discussed in next Section. Then, as the rendering algorithm is casting a ray, the signed distance to the section plane is computed for each sample. Samples on the plane (computed with a tolerance ϵ) are assigned the corresponding value of the texture T . Opacity in non-transparent texels is set to the maximum value, to ensure that structures behind the plane will be occluded. Samples with a negative signed distance are processed in the standard ray-casting way. Moreover, samples having a positive signed distance s_d are only considered when some selected structures have been assigned a non-zero offset distance o_d . In this case, the ray-casting algorithm processes Grey octree nodes if $s_d < o_d$ and otherwise skips them.

VI. RESULTS AND DISCUSSION

We have tested the proposed scheme with a number of volume models V and transfer function sets S . Results presented here correspond to two 512-resolution models. Initial volumes are between 121 and 128 Mbytes, as shown in Tables I and II. Tests have been run on a server with 6 GB of RAM, Intel Core 2 Duo at 3.16 GHz and a client with 4 GB of RAM, Intel Core 2 Duo at 3.06 GHz and Nvidia GeForce GTX 280.

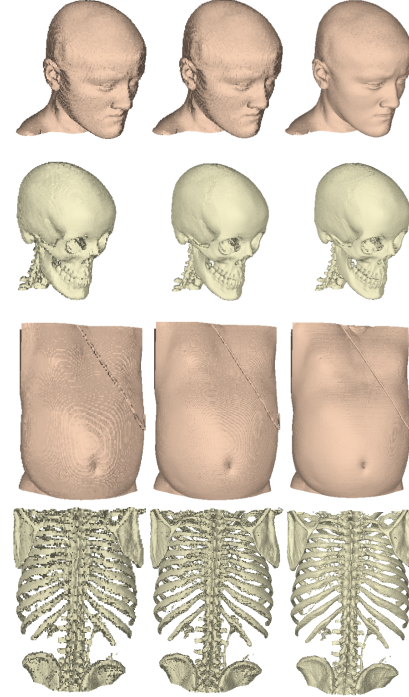


Figure 5. Progressive model transmission and multi-resolution rendering. The three columns show the head and the thorax models with two different transfer functions each, at resolutions of 128, 256 and 512.

Rows in Tables I and II correspond to resolutions among 32 and 512, showing the number of Grey octree nodes, the size of textures B_k and D_k in Kbytes, the amount of information streamed over the network and the client storage requirements for the Gradient Octree. As shown in Table I, the most complex case (thorax with skin, bones and lungs) requires 100 Kbytes for sending the octree data at resolution 32, plus 165 Kbytes for the 64-res octree level and 738 Kbytes for the 128-res octree level. The octree at 256-resolution requires sending $100 + 165 + 738 + 3217 = 4.22$ Mbytes, whereas the full octree at the highest resolution requires 13.5 Mbytes, equivalent to 3.3% and 13.5 % of the initial data size respectively. For the model with skin and bones in Table I, these ratios are 2.4% and 10%, whereas the percentage values for the model in Table II are 2% and 8.4%. High-res sections of the initial volume V , which are sent on demand, require less than 200 Kbytes each. This is

a drastic improvement over thin-client schemes which must send one image per frame. In short, compressed volumes at a 256-resolution require between 2 and 3.3 percent of the initial data, whereas these percentages grow up to values between 8.4 and 13.5 at the highest resolution.

We succeed in sending the tree structure O_k in a lossless way and with only one bit per node, through a sequence of compact arrays B_k . To the best of our knowledge, this outperforms previous octree-based schemes, including [5], [9], [16] and [17]: we send one bit instead of pointers. Moreover, compressing gradients and materials in three bytes is efficient, supports GPU decompression and suffers from a very limited loss in visual quality, as shown in Figure 7.

Progressive model transmission and multi-resolution rendering is shown in the different columns of Figure 5. Some artifacts appear in resolutions lower than 256, but these coarse octrees are only used for approximate model rendering during progressive transmission.

Figure 6 shows some snapshots of the interaction results in the client with different transfer functions. Images show planar sections with high-res information of the initial volume V , the right column presenting two images with offset structures in front of the section plane.

Preprocess times in the presented examples have always been between three and five minutes with a non-optimized code. Inspection and ray-casting in the client is fully interactive with frame rates between 20 fps and 30 fps.

To discuss the loss of visual quality, we have compared the ray-casting of the Gradient Tree in the client with the results of a direct ray-casting of the initial volume V . This loss turns out to be small, as depicted in Figure 7, being acceptable in practical cases.

Our compression results are better than similar client-server schemes for volume rendering, and compare favorably to Marching Cubes (MC) based approaches. While these last schemes must send an average of three triangles per voxel in segmented volumes like $V_E(S)$, we succeed in encoding octree nodes at a rate of 3 Bytes/node, which is much lower than using MC triangles.

Although our tests have been performed on a client with a PC architecture, in the very next future we plan to migrate the decompression and interactive rendering algorithms to mobile devices.

It could be considered that one limitation of the proposed approach is that the visualization in the client is restricted to the materials and medical structures which have been pre-defined in the set S of transfer functions. However, this should be acceptable in most practical cases, as medical doctors usually adopt standard transfer functions to enhance specific tissues and organs. Anyway, we are enhancing visual volume understanding by showing 2D sections of the original volume data on demand and by a number of additional interactive tools. Any strong volume compression

Tissues: Skin, Bones, Lungs					
Resolution	Grey Nodes	B(kb)	D(kb)	Stream(kb)	Mobile Storage(kb)
32	11489	0	34.5	100	100
64	51126	11.5	153.3	165	337.3
128	229203	51.1	687	738	1504.6
256	995610	229	2988	3217	6652
512	4123820	996	12372	13368	28308
Tissues: Skin, Bones					
Resolution	Grey Nodes	B(kb)	D(kb)	Stream(kb)	Mobile Storage(kb)
32	8816	0	26.4	92	92
64	38028	8.8	114	123	255
128	170394	170	510	548	1422
256	737342	737	2211	2381	6291
512	2988561	2989	8967	9704	26655

Table I
THE THORAX MODEL AT RESOLUTION 512³. THE UNCOMPRESSED MODEL SIZE IS 128MB.

must be lossy and specific details are always lost. Our proposal focuses on respecting features which are visible with the TF 's of the predefined set S and on displaying real sections of the initial volume, on demand.

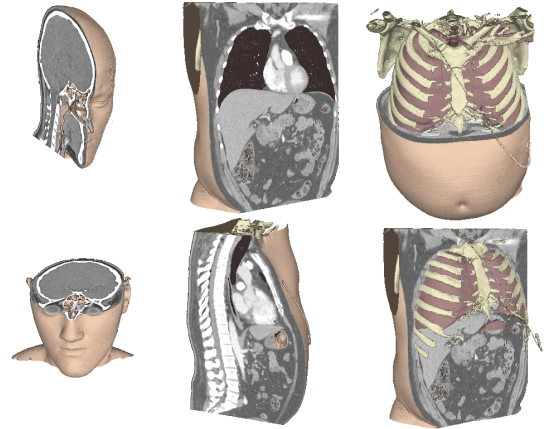


Figure 6. A few snapshots of the interaction in the client with different transfer functions. Images show planar sections with high-res information of the initial volume V , and two images with offset structures in front of the section plane.

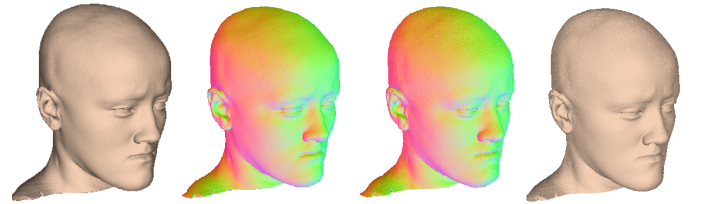


Figure 7. Ray casting of the original volume V with one of the transfer functions in S (left), compared to Gradient Tree ray casting in the client device (right). The two central images show gradient vectors without and with the proposed gradient compression.

VII. CONCLUSIONS AND FUTURE WORK

We have presented a transfer function-aware scheme for the remote interactive inspection of volume models in

Resolution	Grey Nodes	B(kb)	D(kb)	Stream(kb)	Mobile Storage(kb)
32	5641	0	16.8	81.8	81.8
64	29541	5.6	88.5	94.1	178.1
128	135964	29.5	405	434	877
256	577888	135	1731	1866	3891
512	2390093	577	7170	7747	16402

Table II

THE HEAD MODEL WITH SKIN AND BONES AT RESOLUTION 512x512x485. THE UNCOMPRESSED MODEL SIZE IS 121MB.

client-server architectures. Our approach aims at supporting progressive transmission, avoiding gradient computations in the client device and sending a very limited amount of information through the network. Our multi-resolution volume representation, Gradient Octrees can be progressively transmitted to the client in a compact way while achieving a minimum loss of visual quality as compared to state of the art ray-casting renderings. To offer users the possibility of fully understanding the volume data, our scheme supports planar volume sections which are shown with high-resolution volume information, besides interactive extrusion of specific structures. Interaction is autonomous and no further information is needed from the server, except when a new planar section is required.

In our future work we would like to investigate local algorithms for octree-based gradient filtering, by taking into account gradient information in the parent node. Besides this, a good avenue for further research is to optimize thresholds during the generation of the edge volume (V_E^k) generation to minimize the visual error between the ray-casting renderings of V and V_E^k for any viewing direction and for any T_k in S .

ACKNOWLEDGMENT

This project has been supported by the TIN2010-20590-C02-01 Project of the Spanish Government. L. Campoalegre has been supported by a UPC PhD grant. The authors would like to thank the anonymous reviewers for their useful comments and Eva Monclús for her technique support.

REFERENCES

- [1] M. Balsa Rodriguez, E. Gobbetti, J. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter, "A survey of compressed gpu-based direct volume rendering," in *Eurographics State-of-the-art Report*, May 2013.
- [2] F. Lamberti and A. Sanna, "A solution for displaying medical data models on mobile devices," *WSEAS Trans. Information Science & Applications*, vol. 2, pp. 258–264, 2005.
- [3] S.-J. Jeong and A. E. Kaufman, "Interactive wireless virtual colonoscopy," *The Visual Computer*, vol. 23, no. 8, pp. 545–557, 2007.
- [4] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel, *Real-time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.
- [5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering," *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 15–22, 2009.
- [6] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, and T. Ertl, "Hierarchical visualization and compression of large volume datasets using GPU clusters," *Parallel Graphics and Visualization*, pp. 41–48, 2004.
- [7] I. Boada, I. Navazo, and R. Scopigno, "Multiresolution volume visualization with a texture-based octree," *The Visual Computer*, vol. 17, no. 3, pp. 185–197, 2001.
- [8] M. Weiler, R. Westermann, C. D. Hansen, K. Zimmerman, and T. Ertl, "Level-of-detail volume rendering via 3D textures," *Proc of Volviz*, pp. 7–13, 2000.
- [9] S. Laine and T. Karras, "Efficient sparse voxel octrees," *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pp. 55–63, 2010.
- [10] E. Gobbetti, F. Marton, and J. A. I. Guitián, "A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7-9, pp. 797–806, 2008.
- [11] P. Ning and L. Hesselink, "Vector quantization for volume rendering," *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pp. 69–74, 1992.
- [12] J. Schneider and R. Westermann, "Compression domain volume rendering," *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 39, 2003.
- [13] S. Muraki, "Approximation and rendering of volume data using wavelet transforms," *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pp. 21–28, 1992.
- [14] I. Ihm and S. Park, "Wavelet-based 3D compression scheme for interactive visualization of very large volume data," *Computer Graphics Forum*, vol. 18, pp. 3–15, 1999.
- [15] S. Guthe and W. Strasser, "Real-time decompression and visualization of animated volume data," *VIS '01: Proceedings of the conference on Visualization '01*, pp. 349–356, 2001.
- [16] S. K. Suter, J. A. Iglesias Guitián, F. Marton, M. Agus, A. Elsener, C. P. Zollikofer, M. Gopi, E. Gobbetti, and R. Pajarola, "Interactive multiscale tensor reconstruction for multiresolution volume visualization," *IEEE TVCG*, vol. 17, no. 12, pp. 2135–2143, 2011.
- [17] E. Gobbetti, J. A. Iglesias Guitián, and F. Marton, "Covra: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks," *Computer Graphics Forum*, vol. 31, pp. 1315–1324, 2012.
- [18] C. Y. Ip, A. Varshney, and J. JaJa, "Hierarchical exploration of volumes using multilevel segmentation of the intensity-gradient histograms," *TVCG*, vol. 18, no. 12, pp. 2355–2363, 2012.