






QuadStack: An Efficient Representation and Direct Rendering of Layered Datasets

Alejandro Graciano , Antonio J. Rueda , Adam Pospíšil , Jiří Bittner ,
and Bedrich Benes , *Senior Member, IEEE*

Abstract—We introduce *QuadStack*, a novel algorithm for volumetric data compression and direct rendering. Our algorithm exploits the data redundancy often found in layered datasets which are common in science and engineering fields such as geology, biology, mechanical engineering, medicine, etc. *QuadStack* first compresses the volumetric data into vertical stacks which are then compressed into a *quadtree* that identifies and represents the layered structures at the internal nodes. The associated data (color, material, density, etc.) and shape of these layer structures are decoupled and encoded independently, leading to high compression rates ($4\times$ to $54\times$ of the original voxel model memory footprint in our experiments). We also introduce an algorithm for value retrieving from the *QuadStack* representation and we show that the access has logarithmic complexity. Because of the fast access, *QuadStack* is suitable for efficient data representation and direct rendering. We show that our GPU implementation performs comparably in speed with the state-of-the-art algorithms (18-79 MRays/s in our implementation), while maintaining a significantly smaller memory footprint.

Index Terms—Computer graphics, object hierarchies, graphics data structures and data types

1 INTRODUCTION

GEOMETRIC data often contain redundancies that can be represented in a compact way to save space. A compact representation usually requires a certain amount of work to convert the data to the original representation, but algorithms often exist that can access the original values, in an efficient way, directly. Various applications have different needs and these give rise to a wide spectrum of data representations. The focus of this paper is on discrete volumetric data that is usually represented in an uncompressed form as a 3D grid of volumetric elements (voxels).

The key observation of our work is that many research and engineering fields produce *layered volumetric data* which have strong directional anisotropy and high coherency in a prevailing direction. An example is geology (Fig. 1 left), where geological strata are made up of layers of continuous material. Many biological materials such as skin or leaves are also layered, but on a much smaller scale. Even though certain volumetric materials are not composed of clearly visible layers, they include organized stacks of uniform material; an example is particles in materials such as stones, microstructures, or even atmospheric data with layers of air at different humidity, temperature, and velocity. Although existing algorithms can be applied to layered data and

provide good compression, representation, and fast access, we argue that by exploiting their layered structure, we can achieve better results in both data storage and retrieval.

We introduce *QuadStack*, a novel algorithm for volumetric data representation of layered datasets. *QuadStack* uses a *quadtree* for data representation while efficiently encoding the layers in the tree. The layers are converted to stacks, and the algorithm decouples the voxel values from their height values. We also introduce an algorithm for *value retrieval* from the *QuadStack* representation and we show that the access requires $\mathcal{O}(\log(n) + m)$ time, where $n = w \times h$ for a voxel space with dimensions $w \times h \times d$ and m is the maximum stack size. In practice, m is small compared to n in a layered model, so the method can be assumed to run in logarithmic time. Because of its fast access, *QuadStack* is suitable for efficient rendering of layered data and can be implemented on the GPU as we show in a raycasting implementation.

We apply our algorithm to real datasets from different domains. In particular, we show its performance on geological datasets [1], industrial models [2], microstructural data [3], and with a snapshot of a magnetic reconnection simulation [4]. We render these datasets by using *QuadStack* which has comparable performance to other state-of-art techniques, but has 66 percent to 99 percent less memory requirements compared to the uncompressed data. An example in Fig. 1 shows three layered data from various applications compressed and rendered by using *QuadStack*.

We claim the following contributions: 1) *QuadStack*, a novel data structure that arranges volumetric layered datasets into a set of heightfields, isolating them from the attribute values, and compressing them into a quadtree, 2) a fast method for the *QuadStack* construction based on string matching, and 3) a rendering algorithm that displays the compressed data directly.

- A. Graciano and A. J. Rueda are with the Universidad de Jaén, 23071 Jaén, Spain. E-mail: {graciano, ajrueda}@ujaen.es.
- A. Pospíšil and J. Bittner are with the Czech Technical University in Prague, 166 36 Prague 6, Czech Republic. E-mail: {pospiad2, bittner}@fel.cvut.cz.
- B. Benes is with Purdue University, West Lafayette, IN 47907 USA. E-mail: bbenes@purdue.edu.

Manuscript received 13 Sept. 2019; revised 12 Feb. 2020; accepted 13 Mar. 2020. Date of publication 18 Mar. 2020; date of current version 29 July 2021.
(Corresponding author: Bedrich Benes.)

Recommended for acceptance by E. Eisemann.

Digital Object Identifier no. 10.1109/TVCG.2020.2981565

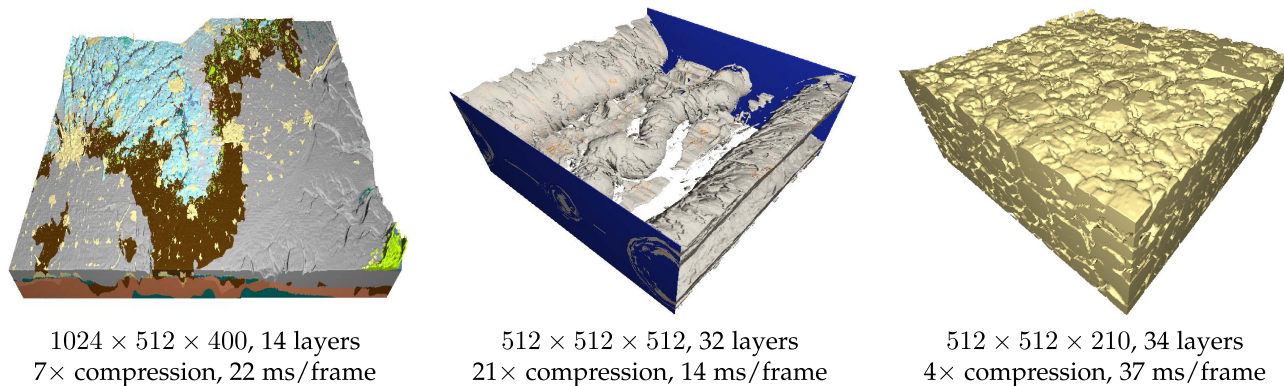


Fig. 1. *QuadStack* efficiently compresses and directly renders layered data like large terrain (left), outputs of a magnetic reconstruction simulation (middle), or microstructures (right). We report the compression ratios and rendering performance on an NVIDIA 970 GTX.

2 PREVIOUS WORK

Here we review related methods for representing and visualizing general volumetric data followed by an overview of methods specific for heightfields and layered data.

The most common *volumetric data representation* is the use of regular grids [5], [6] that allow quick random access and modifications needed by many data processing algorithms. However, regular grids are often highly redundant and do not provide scalable data representation. The grids are compressed into Octrees [7], hierarchical grids [8], or collections of tetrahedra [9]. In general, these methods are able to focus on the details of representation into the corresponding data, but modifications can require recalculation of the compressed representations.

A specific class of volumetric representation is layered representation, which has been studied in the context of geological structures such as terrains and landscapes. An inspiration for our work is the layered data structure for terrain representation introduced by Benes and Forsbach [10], extended to enable interactive modeling of terrains, including simulations of natural processes such as erosion [11], and the evolution of snow covered mountains [12]. Peytavie *et al.* [13] used this representation for modeling and visualizing complex terrain, including features like arches or overhanged cliffs. Later, Löffler *et al.* [14] achieved realtime rendering using a LoD hierarchy. Both methods convert the layered data to a triangle mesh prior to the visualization, and only the surface is rendered. Also, based on this data structure, the recent work of Graciano *et al.* [15] introduced the Stack-Based Representation (SBR) that is a compact representation for a layered volumetric datasets. This work also introduced a GPU-based method for direct rendering of layered geological structures by using SBR. *QuadStack*, the method being introduced in the present work, goes further by proposing compression of stacks using a *quadtree* without compromising realtime visualization.

Volume data visualization is often conducted by using direct volume ray casting that provides a flexible approach to handle varying density of the data, implementing transfer functions, or focusing the visualization by clipping [16]. Amanatides and Woo [5] introduced a fast algorithm for traversing volumetric data encoded in a regular grid using a 3D-DDA algorithm. Levoy [8] extended the traversal to hierarchical representation. Danskin and Hanrahan [17] introduced several adaptive acceleration methods for volume ray

tracing using homogeneity and opacity accumulation. Cohen and Sheffer [18] proposed proximity clouds to accelerate traversal of empty regions. Revelles *et al.* [7] developed an optimized Octree ray tracing using a fast recursive algorithm. Efficient skipping techniques for Octree traversal were proposed by Grimm *et al.* [19] and Lim and Shin [20]. Crassin *et al.* [21] used node and brick pools to optimize the ray traversal and data filtering using sparse voxel Octrees [22]. A detailed discussion of direct volume rendering techniques can be found in surveys [16], [23].

Kämpe *et al.* [24] encoded the geometry of high resolution volumetric models obtained using DAGs. These models are typically generated from high-resolution rasterization of surface representation into binary voxels representing either full or occupied model parts. Later, Dado *et al.* [25] and Dolonius *et al.* [26] proposed techniques for attaching attributes to geometry compressed using DAGs. Our method also decouples geometry, and attributes and encodes them separately. A notable difference in our method is that our representation primarily structures the data according to attributes (layers). This allows us to optimize direct rendering of the compressed data for transfer functions that cull many layers that are often used to study the layered datasets.

Guthe and Goesele [27] proposed a method using block-wise compression of general volumetric data and block-wise decompression optimized for direct volume rendering in CUDA. Our method also compresses geometric information, but it also stores semantic information about the layers that can be used, for example, to render individual layers differently. We compare to this method in Section 7.

Volumetric data can be also visualized by converting to *boundary representation* and applying efficient ray tracing methods for B-reps such as kD-Trees [28] or BVHs [29], [30]. There are several powerful implementations for both CPU [31], [32] and GPU ray tracing [33]. Limited bandwidth and data access latency are two of the main limitations for GPU rendering. Cache efficient layouts [34] and compressed data representation [35], [36] mitigate both of these issues.

We focus on layered data that can be thought of as a general form of *heightfields* that are commonly used to represent terrains with a single layer of material [37], [38], [39], [40].

Early heightfield rendering used a 2D grid traversal with a DDA [41]. Later techniques often used precomputed hierarchical representations [42], [43]. Henning and Stephenson [44] focused on accelerating ray tracing and local reconstruction

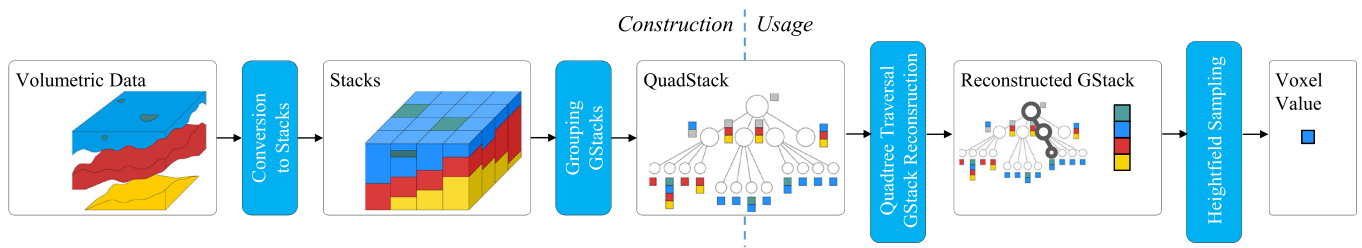


Fig. 2. Overview: The input volumetric data is converted to stacks then similar stacks are put into groups of stacks (gstacks) that are organized in a *quadtree*. When using the *QuadStack*, the *quadtree* is traversed first, and the topology of the given part of the volume is reconstructed. Then the corresponding layer boundaries encoded as heightfields are sampled to determine the result.

for the ray at the intersection. An efficient GPU implementation of terrain ray casting was proposed by Dick *et al.* [45], and a hybrid rendering technique for terrains combining rasterization with ray casting was proposed by Ammann *et al.* [46]. Lux and Fröhlich [47] focused on out-of-core large terrains rendering. Acceleration of terrain rendering by skipping empty regions of space was addressed by Baboud *et al.* [48] and more recently by Lee *et al.* [49].

Scandolo *et al.* [50] used compressed hierarchical representation to encode high resolution shadow maps, which are similar to heightfield compression. Their method maintains accurate shadows by encoding depths with values within limits provided by two consecutive depth layers. For other applications, such as representing and rendering general layered models, the method only provides a lossy compression of individual layers and is optimized for lookups using point queries instead of ray queries needed for direct visualization.

A number of other efficient techniques for multi-resolution heightfield representations and rendering have been surveyed by Pajarola and Gobbetti [51].

Our method builds on previous work by combining the layered representation of general volumetric data with hierarchical representation using *quadtree*s and a collection of heightfields. We hierarchically encode the volume into regions that have constant layer topology for which simple compressed heightfield representation can be used. This representation achieves high compression rates while still allowing efficient data retrieval. Thus, the method reduces memory usage as well as the bandwidth and latency when directly rendering the compressed representation.

3 METHOD OVERVIEW

Datasets are often composed of horizontal layers of identical values of a certain material or physical property that we refer to as *attributes*. The approach of our method is in representing the layers as run-length encoded vertical stacks [10] that are encoded further into a horizontal *quadtree*. In this way, a *QuadStack* is an efficient data structure for layered volumetric data that decouples the attribute data (layer attribute values) from the geometric data (layer heights).

Without a loss of generality, we assume that the direction of the layers is known, and that the data is oriented so layers are parallel to the (horizontal) direction \mathbf{xy} . Although the height of each layer may vary by location, the vertical sequence (the order) of the layer attributes is spatially coherent as can be seen in Fig. 2 on the left. The height of the layers may vary between two vertical columns, but their order,

often, will not change. A common change in real-world data sets is that one layer disappears or a new one is introduced.

The input volumetric data V (Fig. 2) with dimensions $w \times h \times d$ is first converted to a set of stacks S , where the stack $S_{x,y} \in S$ represent the columns of voxels $V_{x,y}$ encoded as a sequence of *intervals* i_1, i_2, \dots, i_n . Each interval consists of voxels with the same attribute value. The conversion to stacks is depicted as the first step of the construction in Fig. 2. A detailed representation of a layered volumetric dataset as stacks is shown in Fig. 3 and explained in Section 4.

In the second step, a region *quadtree* organizes S into quadrants of stacks with the same sequence of attribute values, referred to as *groups of stacks*, or simply as *gstacks*. A *gstack* G encodes the stacks in the quadrant $[x_{\min}y_{\min}, x_{\max}y_{\max}]$ in a compact manner, and is defined as a sequence of n intervals i_1, i_2, \dots, i_n . The attributes a_1, a_2, \dots, a_n of the intervals of G are common to all the stacks in the quadrant: $S_{x,y} \in S$ where $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$. A *quadtree* of *gstacks* is denoted as *QuadStack* (see Fig. 2 and Section 5).

The *QuadStack* provides a lossless compression of the input data. It can be easily converted back to a voxel-based representation by point sampling (see Fig. 2 right). A similar sampling procedure can be used to directly render data represented in *QuadStack* by using a modified ray traversal algorithm for *quadtree* (Section 6).

4 STACK-BASED REPRESENTATION

Given a voxel grid V with resolution $w \times h \times d$, each voxel $v_{x,y,z} \in V$ stores one or more *attributes*, such as color, material, or density, that depend on the application. Layers are the maximal sets of connected voxels with a constant value for a given attribute.

The Stack-Based Representation (SBR) of V (see Fig. 3) is its decomposition into a set S of vertical stacks, where each stack $S_{x,y} \in S$ comprises the space defined by the column of voxels at position \mathbf{xy} :

$$S_{x,y} \cong V_{x,y} = \bigcup_{i=1}^d v_{x,y,i}.$$

A stack is compacted as a run-length encoding of voxels with the same value for the attribute. Therefore, the stack $S_{x,y}$ is a sequence of intervals i_1, i_2, \dots, i_n along the z axis where i_k is a tuple $i_k = \langle a_k, h_k \rangle$ that represents the space comprised by a range of voxels of the column $V_{x,y}$ with identical attribute value a_k :

$$i_k \cong \bigcup_{i=1+h_{k-1}}^{h_k} v_{x,y,i}.$$

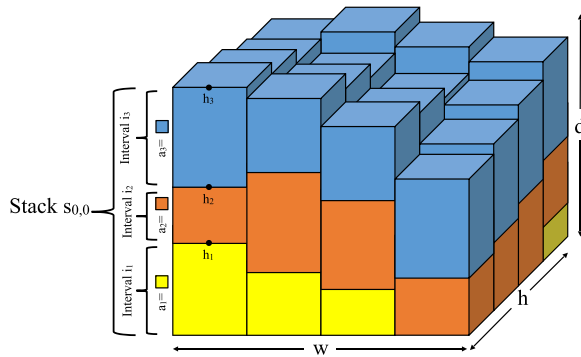


Fig. 3. Stack-Based Representation: A voxel dataset of resolution $w \times h \times d$ composed of layers is organized into vertical stacks consisting of intervals. Each interval $i_k = \langle a_k, h_k \rangle$ with the attribute value a_k and height h_k .

If $k = 1$ then h_{k-1} is assumed to be 0. The intervals are sorted by height in ascending order: $h_k < h_{k+1}$ for any given k such that $1 \leq k \leq n$.

The complexity of the SBR construction is $\mathcal{O}(n)$, where $n = w \times h \times d$ is the number of voxels, because each voxel needs to be processed exactly once. The SBR construction is embarrassingly parallel, because the stack construction does not require information about neighboring stacks.

5 THE QUADSTACK DATA STRUCTURE

A simple approach to compressing SBR data would be to use a hierarchical data structure such as a *quadtree* that would efficiently encode the neighboring stacks with the same sequence of intervals, i.e., stacks that have identical attribute values and heights. However, variations in the interval heights are common and would lead to low compression rates due to the high number of tree subdivisions.

Our observation is that while stacks differ significantly in their height values, their attribute values do not change very often between neighboring stacks. This change happens only when a layer disappears, or when a new layer appears, as can be seen in the left image in Fig. 2. Therefore our approach is to pack groups of neighbouring stacks with

an identical sequences of attribute values into a single structure denoted as a group of stacks or *gstack*.

5.1 Group of Stacks - Gstacks

Given the decomposition of volume V into stacks S , a *gstack* G represents a rectangular region of S with the same number of intervals n and identical sequence of attribute values a_1, a_2, \dots, a_n . More specifically, G represents the stacks $S_{x,y}$ of a rectangle $[x_{min}y_{min}, x_{max}y_{max}]$ of S , where $1 \leq x_{min} \leq x \leq x_{max} \leq w$ and $1 \leq y_{min} \leq y \leq y_{max} \leq h$. Since the attribute information of the stacks $S_{x,y}$ is identical, G can be encoded in a compact way as a sequence of intervals i_1, i_2, \dots, i_n . Each interval $i_k = \langle a_k, H_k \rangle$ contains the attribute value a_k , common to all the intervals i_k of $S_{x,y}$, and a heightfield H_k with dimensions $(x_{max} - x_{min} + 1) \times (y_{max} - y_{min} + 1)$ that stores the heights of these intervals. More specifically, the height h_k of $S_{x,y}$ is mapped to the height $h_{x-x_{min}, y-y_{min}}$ of H_k . The intervals and the attribute values are consistently ordered for all stacks $S_{x,y}$ therefore the heightfields H_k never intersect i.e., given $h_{i,j,k} \in H_k$ and $h_{i,j,k+1} \in H_{k+1}$, where $1 \leq k \leq n$, the condition $h_{i,j,k} < h_{i,j,k+1}$ is always met.

A *gstack* is simple and space-efficient encoding for a group of stacks with identical attribute information. Although only attribute information is compressed, the geometry information is stored as a set of non-intersecting heightfields that can also be compressed by using any existing heightfield encoding method.

Gstacks are built during the construction of a *QuadStack* that combines the spatial decomposition of a *quadtree* with the compact representation for groups of similar stacks given by *gstacks*.

5.2 Group of Stacks Hierarchies

A *QuadStack* represents the stacks as a hierarchy of *gstacks*. It divides the volume in the direction of xy recursively until a quadrant can be represented by a *gstack*. These quadrants are not guaranteed to be squared, or a power of two, since there are no restrictions on the dimensions of the volume.

A *QuadStack* stores information, not only in leaf nodes, but also in internal nodes, which further improves the

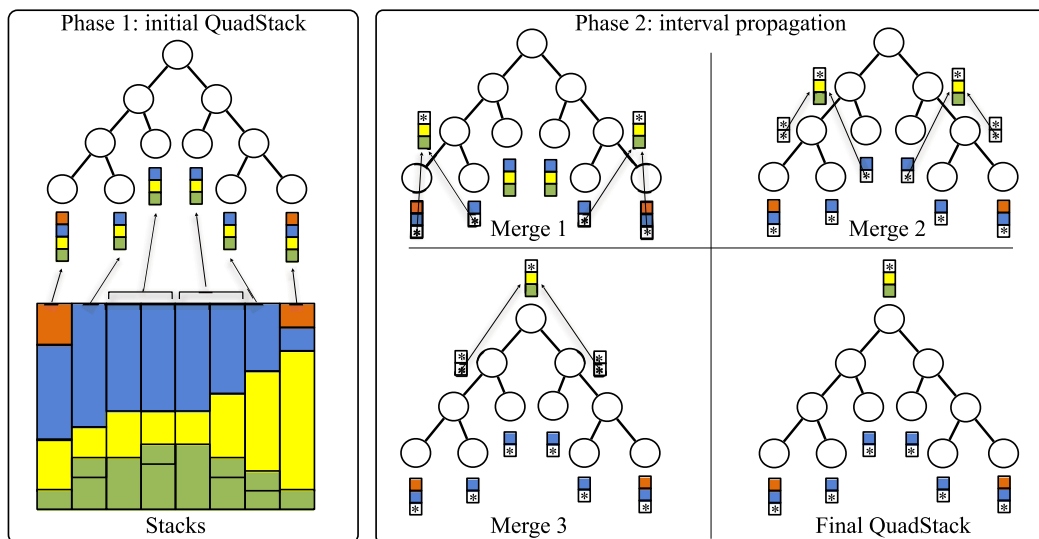


Fig. 4. The *QuadStack* construction: The initial *QuadStack* is a *quadtree* that is optimized by merging intervals of equal or similar attributes.

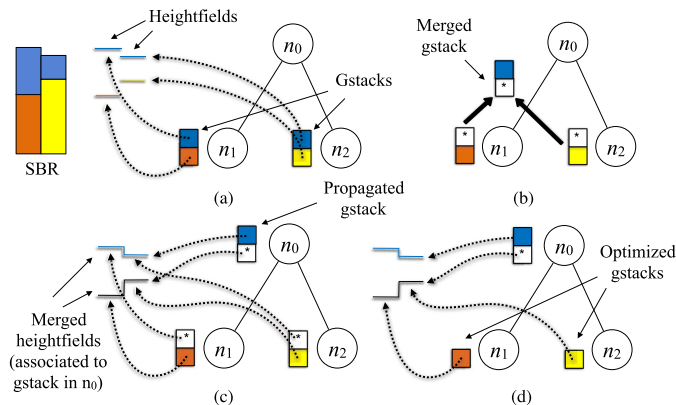


Fig. 5. Details of the *QuadStack* construction: Initial construction as a *quadtree* encoding groups of stacks with the same sequence of attributes (a), merging *gstacks* in nodes n_1 and n_2 into a *gstack* with common intervals and $*$ -intervals (b), propagation of the new *gstack* to the parent node n_0 , restructuring heightfields (c) and optimization, deleting $*$ -intervals in nodes n_1 and n_2 associated to the intervals propagated to the parent node (d).

compression. An internal node can contain a *gstack* grouping intervals common to all its descendants. Since these intervals are not necessarily consecutive, the gaps between them, corresponding to one or more intervals that are stored elsewhere (i.e., in a descendant or ancestor node), are represented with a new type of interval called *wildcard interval* or $*$ -interval. This enables a flexible form of a *gstack* that combines intervals (hereinafter referred to as *terminal intervals*) with areas lacking information on this level of the *QuadStack*. These intervals, in many cases, correspond to intervals with different attribute information that could not be directly represented by the *gstack*.

Fig. 5a shows a 2D depiction of a SBR and its corresponding *QuadStack* represented as a binary tree. The *gstacks* in nodes n_1 and n_2 encode the stacks as intervals of different attributes with their associated heightfields. Fig. 5c shows an equivalent *QuadStack* where the blue intervals, common to nodes n_1 and n_2 , have been stored in the *gstack* at n_0 , together with a $*$ -interval that represents the rest of intervals of the block of stacks (intervals orange and yellow). The blue interval in *gstacks* of the leaf nodes is replaced by $*$ -intervals, since it is already stored in an ancestor.

Just like a terminal interval, a $*$ -interval has an associated heightfield. Let G be a *gstack* stored in node n and let $i_w \in G$ be a $*$ -interval, defined as $i_w = \langle *, H_w \rangle$. If i_w represents the intervals $i_k, i_{k+1}, \dots, i_{k+j}$ of *gstack* G_d in a descendant n_d of node n , then the heightfield H_{k+j} of the top interval i_{k+j} corresponds to a quadrant of H_w . In this way H_w combines the heightfields of the top intervals of all the sets of intervals existing in descendant nodes which are grouped by i_w . This is depicted in Fig. 5c as the heightfield associated to the $*$ -interval in n_0 . If the $*$ -interval in G refers to a group of intervals in *gstack* G_a from an ancestor n_a of n , H_w corresponds to a quadrant of the heightfield H_{k+j} , since G_a covers a larger part of the xy plane than G . This case corresponds to the heightfields of the $*$ -intervals in nodes n_1 and n_2 .

5.3 QuadStack Construction

The *QuadStack* is constructed in two steps: a top-down *sub-division*, and a bottom-up *merging*. The subdivision step is a

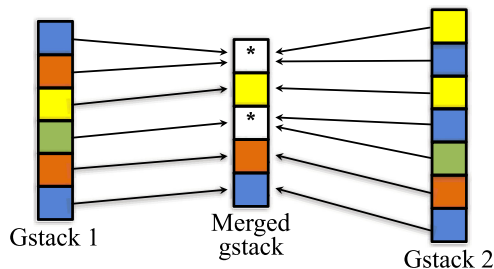


Fig. 6. Finding a common mapping for two *gstacks* that maximizes the number of terminal intervals.

standard *quadtree* construction of the stacks by using the criterion of same attribute sequence. This criterion ensures that the blocks of stacks at each leaf node can be encoded as a *gstack*, and the resulting data structure is already a *QuadStack*. Fig. 4 illustrates the resulting *QuadStack*.

Although the first step generates a more compact representation for the volumetric model than an SBR, layers of common attributes lead to duplicate intervals in many leaf nodes, as shown by the blue, yellow, and green attributes on the left part of Fig. 4. The second step extracts and merges these duplicate intervals in ancestor nodes. It proceeds from the bottom-up by propagating to a node every interval common to all its children (i.e., having the same attribute value). We map the attribute values of the intervals of the four *gstacks* in the children to a common sequence of attribute values, using $*$ to group non-matching attributes (Fig. 6).

We are interested in mapping with the highest number of terminal intervals. The search space can be very large and our problem is related to finding common motifs with gaps, with applications in text mining and the analysis of DNA sequences. Many solutions have been proposed [52], [53], [54] that assume certain restrictions (e.g., motif size, maximum gap size, etc.) and require an exact match of the motifs, or accepting a certain degree of similarity.

However, our sequences are rather short and we propose a brute-force solution for this problem (Algorithm 1). The input is two *gstacks* and the output is two *gstacks* with a matching sequence of interval attribute values. The algorithm takes every possible pair of intervals from the first and second *gstack* and tests if their attribute values match; if a matching pair $i_{1,s}, i_{2,t}$ is found, the intervals are added to their corresponding result *gstack* and the function calls itself with the remaining intervals $i_{1,s+1}, \dots, i_{1,n}$ and $i_{2,t}, \dots, i_{2,m}$. If $i_{1,s}$ and $i_{2,t}$ are not in the first positions of their *gstacks* (i.e., $s > 0$ and $t > 0$), the predecessors intervals $i_{1,1}, \dots, i_{1,s-1}$ and $i_{2,1}, \dots, i_{2,t}$ are grouped into two $*$ -intervals associated to the heightfields of the top intervals $i_{1,1}$ and $i_{2,1}$. The number of terminal intervals of the solution obtained are computed, and finally the best solution is returned. Generalizing this solution to the four children of a *QuadStack* node is straightforward. The theoretical complexity of this algorithm is $\mathcal{O}((m!)^4)$ for four stacks with m intervals, but it performs much better in practice with the heuristics described in Section 5.5.

If the derived *gstacks* have at least one terminal interval, they are merged into a single *gstack* at the parent node (Figs. 5b and 5c). The heightfields of the newly created *gstack* are generated by packing the four heightfields of the

intervals of the derived *gstacks*. Finally the derived *gstacks* are deleted, and every propagated interval is converted to an ***-interval at the children *gstacks*, grouping adjacent intervals if necessary (Fig. 5c). If a *gstack* ends up as a single ***-interval, it can be safely deleted from the node since it does not provide any information.

Algorithm 1. Algorithm *matchGS*

Input: *gstacks* $G_1^i = \{i_{1,1}, \dots, i_{1,n}\}$ and $G_2^i = \{i_{2,1}, \dots, i_{2,m}\}$.
Interval $i_{i,k} = \langle a_{i,k}, H_{i,k} \rangle$ where $a_{i,k}$ and $H_{i,k}$ are its attribute value and heightfield respectively.

Output: *gstacks* G_1^o and G_2^o with the same sequence of attribute values.

```

if  $G_1^i \neq \emptyset$   $G_2^i \neq \emptyset$ 
   $G_1^o \leftarrow \{ \langle *, H_{1,1} \rangle \}$ 
   $G_2^o \leftarrow \{ \langle *, H_{2,1} \rangle \}$ 
else
   $G_1^o \leftarrow \emptyset$ 
   $G_2^o \leftarrow \emptyset$ 
 $sc^{best} \leftarrow 0$ 
foreach interval  $i_{1,s}$  from  $G_1^i$  do
  foreach interval  $i_{2,t}$  from  $G_2^i$  do
    if  $a_{1,s} = a_{2,t}$  and  $a_{1,s} \neq *$  and
       $(s > 1 \text{ xor } t > 1)$  and
       $(s < n \text{ xor } t < m)$  then
       $G_1^r, G_2^r \leftarrow matchGS(\{i_{1,s+1}, \dots, i_{1,n}\},$ 
         $\{i_{2,t+1}, \dots, i_{2,m}\})$ 
       $sc \leftarrow numTerminalIntervals(G_1^r)$ 
      if  $sc > sc^{best}$  then
         $sc^{best} \leftarrow sc$ 
      if  $s > 0$  then
         $G_1^o \leftarrow \{ \langle *, H_{1,1} \rangle \} \cup \{i_{1,s}\} \cup G_1^r$ 
         $G_2^o \leftarrow \{ \langle *, H_{2,1} \rangle \} \cup \{i_{2,t}\} \cup G_2^r$ 
      else
         $G_1^o \leftarrow \{i_{1,s}\} \cup G_1^r$ 
         $G_2^o \leftarrow \{i_{2,t}\} \cup G_2^r$ 
    end
  end
return  $G_1^o, G_2^o$ 

```

The time complexity is $\mathcal{O}(n \log n)$ for the initial *quadtree* construction and $\mathcal{O}(n \times (m!)^4)$ for the interval propagation phase, where n is the number of stacks in the SBR ($n = w \times h$) and m the maximum number of intervals in a *gstack*.

5.4 Heightfield Compression

QuadStack implements a compact encoding of the ordered sequence of attributes, but does not deal with the compression of the interval heights. In our approach, attribute and heightfield representation are decoupled, therefore heightfields can be stored in a raw form, or compressed by any existing method such as the algorithm of [55].

A simple delta encoding provides good results, since height values usually vary progressively. However, the main problem is that the access to any data requires decompressing the whole dataset which limits the practical use in applications where efficient queries or traversals are required (e.g., realtime visualization).

We propose a method that provides a trade-off between compression and access time, inspired by the work of Andujar [56]. This method partitions the heightfield into equal-sized blocks and compresses the values in each block

independently. Consider a heightfield H partitioned into $w \times h$ blocks with the same dimensions $m \times n$. For each block $B_{i,j} \in H$ the lower height value is taken as the base value, encoding the $m \times n$ elements in the block as differences from this base value. Using blocks of a relatively small size makes these differences close to zero, allowing them to be encoded with a reduced number of fixed bits. This enables random access to a particular location with only two reads: a first one to a header that comprises the base value of the block and the number of bits required to encode each height difference, and a second one to the actual difference located at the bit field. Other predictors for a value in a block are possible: for instance it can be encoded as the difference with respect to the bilinear interpolation of the height values at the four corners of the block.

5.5 Optimizations

The *QuadStack* construction algorithm described in Section 5.3 generates a compact representation of the attributes of the model. This can be further improved if ***-intervals that do not provide useful information for sampling operations are removed. Good candidates are the ***-intervals representing information that can be found elsewhere in an ancestor node, such as the top ***-intervals in the *gstacks* of nodes n_1 and n_2 in Fig. 5c. The information represented by these intervals is included in the blue interval in the *gstack* of the node n_0 . These intervals are never reached during sampling (Section 6), and can be discarded during the bottom-up phase of the *QuadStack* construction algorithm. When an interval is propagated to an ancestor (Figs. 5b and 5c), it is completely removed from the initial *gstack*, instead of being converted to an ***-interval. Note that the result is no longer a *gstack*, since it represents only a subset of intervals instead of the full range of the stacks. We refer to this as a *partial gstack*. Under the described conditions, *gstacks* can be converted into *partial gstacks* reducing the *QuadStack* space requirements without affecting its performance in sampling or rendering operations. Fig. 5d shows the resulting *QuadStack* after this optimization.

An optimal match of the *gstacks* during the bottom-up phase is essential for a good attribute compression. Algorithm 1 finds the optimal matching, although its time complexity is high. We use two efficient heuristics to reduce its computation time.

First, the exploration of a new solution can be avoided if the minimum of the lengths of the two lists of explored intervals is less than the best score so far, since a better score cannot be found. Notice that the score of a solution is the number of terminal intervals, so at least two lists with a higher number of intervals are required to be able to find a better solution.

Second, we store the optimal matching found for four given lists of intervals, since certain combinations are explored repeatedly. For this purpose we use a simple map with a key computed from the length of the four lists of intervals. This improvement reduces the time requirements of the algorithm to $\mathcal{O}(m^5)$. To illustrate this, in the computer used for our experiments (Section 7) matching 4 stacks with 30 intervals is solved in less than 26 ms compared to 764 ms without optimizations (30× faster).

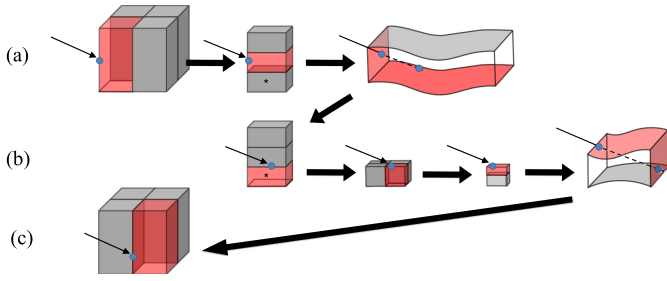


Fig. 7. *QuadStack* raycasting is first resolved at the *QuadStack* level, then at the interval level and finally, at the heightfield level (a). When an ***-interval is found, a recursive call to traverse the *gstacks* in children nodes is required (b). After processing a *gstack*, the traverse continues with the next one, until the ray exits the volumetric model.

6 QUADSTACK SAMPLING AND DIRECT RENDERING

We discuss two techniques for retrieving data from *QuadStack* representation. The first is point sampling that is crucial for compression/decompression applications, the second technique uses ray casting and enables direct rendering of *QuadStack* data.

Algorithm 2. Function *sampleQS*

Input: node n to be sampled; sampling point p .
Output: final node and attribute value sampled at p .

```

 $a_r \leftarrow null$ 
 $n_r \leftarrow null$ 
if  $p_z > 0$  then
  foreach interval  $i_k = \langle a_k, H_k \rangle$  from gstack  $G$  in  $n$  do
    if  $h_{p_x, p_y} \geq p_z$  then
       $a_r \leftarrow a_k$ 
       $n_r \leftarrow n$ 
      break
  end
  if  $r = *$  then
     $C \leftarrow getChildren(n)$ 
    if  $C \neq \emptyset$  then
      Given  $C = \{c_0, c_1, c_2, c_3, c_4\}$ 
      if insideQuadrant( $c_0, p$ ) then
         $a_r, n_r \leftarrow sampleQS(c_0, p)$ 
      else if insideQuadrant( $c_1, p$ ) then
         $a_r, n_r \leftarrow sampleQS(c_1, p)$ 
      else if insideQuadrant( $c_2, p$ ) then
         $a_r, n_r \leftarrow sampleQS(c_2, p)$ 
      elseif insideQuadrant( $c_3, p$ ) then
         $a_r, n_r \leftarrow sampleQS(c_3, p)$ 
  return  $a_r, n_r$ 

```

6.1 Point Sampling for Selective Decompression

The *QuadStack* decompression can be performed selectively by using *point sampling* of the *QuadStack* representation. Algorithm 2 shows the regular structure of a point sampling procedure in a hierarchical data structure adapted to a *QuadStack*. Querying a point p is carried out by recursive traversing of the *quadtree* data structure. First, an inclusion test between p and the bounding box of the data is computed, if the test is successful, the query can start by sampling the root node. In order to sample a node, the heightfield H_k of each interval in

the *gstack* must be sampled at the xy position of p , comparing its height with the z coordinate. The iteration stops when the lowest interval whose height is above z is found. If it is a terminal interval, regardless of whether a leaf node is reached or not, the attribute is returned. When an ***-interval is found, the traversal continues in the successive nodes.

The overall time complexity is $\mathcal{O}(\log n + m)$ since, in the worst case, it is necessary to reach a leaf of the tree to retrieve the interval, checking at most m intervals during the traversal. Point sampling can also be easily generalized to decompress an arbitrary rectangular region or box of the model into stacks, or further into voxel data.

6.2 Ray Casting for Direct Volume Rendering

GPU-accelerated ray casting is currently the most common approach for the visualization of volumetric data providing a good trade-off between simplicity, quality, and speed. Models represented by *QuadStack* can be rendered by ray casting without using an intermediate representation (e.g., a SBR or a 3D grid of voxels). Similar to other hierarchical data structures *QuadStack* allows an efficient implementation of ray casting, which is solved at the *gstack* level first, then at the interval level, and finally at the heightfield level, as depicted in Fig. 7.

The rendering procedure starts by computing the intersection between the ray and the *gstack* at the root node. This can be computed efficiently, considering that a *gstack* defines a cuboid that spans the entire z dimension of the volumetric space. Then, the first intersection with an interval i_k of the *gstack* is calculated. This involves computing the intersection of the ray with its four lateral faces and two bounding heightfields: H_k (top) and H_{k-1} (bottom). If the interval i_k is terminal, its contribution to the accumulated color and opacity is computed as the integral of the transfer function for the attribute a_k between the entry and exit points of the ray, together with an opacity correction due to adaptive sampling [57]. The ray processing stops if the opacity of the color is close to one. If i_k is an ***-interval, a recursive call is made to compute the contribution by the ray traversal of the *gstacks* in the four descendant nodes. After i_k has been processed, the traversal continues with a new interval until the ray exits the *gstack*. If the *gstack* is at the root node the *QuadStack* sampling is completed. In general, it implies the return of a recursive call and further processing of the *gstack* at the parent node.

The most time-consuming step in the *QuadStack* raycasting is the ray-heightfield intersection computation. In order to accelerate this step, each heightfield is mipmapped storing the min-max instead of averaging values [58]. Each mipmap level defines a bounding geometry for the heightfield with the shape of a set of cuboids with the same dimensions in the xy plane. The highest mipmap level represents the coarsest approximation (i.e., a bounding box) and the level zero represents the finest (i.e., the heightfield itself). To test if a ray intersects the heightfield associated with a given interval of a *gstack*, first the intersection with the bounding cuboid defined by the highest mipmap level is computed. If the cuboid is hit, the intersection computation continues with the four contained cuboids in the preceding mipmap level, until the ray passes by or hits the heightfield at level zero. The extra memory required (66 percent for each heightfield)

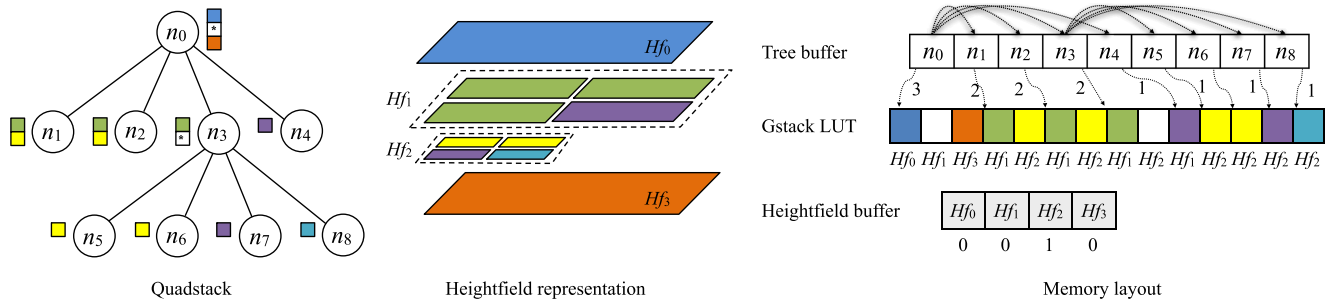


Fig. 8. a) Rendering procedure overview and GPU memory structure for the direct rendering of a *QuadStack* represented. b) Its heightfield arrangement. c) Indices in the heightfield buffer indicate the *QuadStack* level to which the heightfield belongs.

can be reduced by using heightfield compression explained in Section 5.4, resulting in a good trade-off between rendering time and memory footprint.

7 IMPLEMENTATION AND RESULTS

We have implemented our algorithm in C++ with support of OpenGL and GLSL. Results were generated on a desktop computer with an Intel i7-4790 quad-core processor running at 3.6 GHz, 16 GB of RAM, and a NVIDIA GTX 970 GPU. Below, we first discuss details of the GPU implementation and then present results and comparisons.

7.1 *QuadStack* Encoding in the GPU Memory

The key for an efficient raycasting is a careful encoding of the model representation in the GPU memory. Our memory layout for *QuadStack* consists of three buffers: a tree buffer that encodes the *QuadStack* structure, a lookup table (LUT) for the set of *gstacks*, and a heightfield buffer that packs the heightfields associated with each *gstack* (Fig. 8).

The structure of the tree buffer is inspired by [60], where each tree node keeps either the data itself (leaf), or an index to its descendants (otherwise). Contrary to the previous work, an inner node in our structure also contains the corresponding *gstack*. Therefore, a node in the tree buffer holds indices of its children and an extra pointer to the *gstack* LUT indicating the beginning of the sequence of intervals and its size.

The *gstack* LUT comprises every *gstack* in a consecutive manner. Each element of this buffer defines a *gstack* interval formed by its attribute and a pointer to the beginning of its corresponding heightfield in the heightfield buffer.

Heightfields are compressed by using the approach described in Section 5.4. The detailed structure of the heightfield buffer is shown in Fig. 9. A header contains a field with the number of blocks into which the heightfield is divided, followed by a sequence of block descriptors that comprises the base value, the number of bits required for encoding the height differences, and the address of the height data. Next, the encoded height data for each block is stored. Morton encoding layout both for block metadata and height differences provides the required spatial coherence when accessing data. As shown in Fig. 8, an index indicating the level of the *QuadStack* to which the heightfield is associated (base level) has been added. When a *gstack* in a descendant node references a specific quadrant of this heightfield, the use of this index avoids adding extra information at the LUT buffer: the actual quadrant can be quickly determined from the base level of the heightfield and the level queried.

7.2 Volumetric Data Compression

We evaluated the *QuadStack* to perform lossless compression of the input volumetric data. We used five datasets for the tests that exhibit strong to medium layered structure: two terrain models with several layers of different geological content (Terrain1, Terrain2) from [1], microstructure of a Li-Pol battery (Battery) [3], a part of an industrial model of a wing of a plane (Wing) [2], and a magnetic reconnection simulation (Magnetic). We wanted to cover a wide spectrum of applications and a wide variety of layers and structures.

The measured results are shown in Table 1. Interestingly, models with comparable maximum number of layers (e.g., Terrain1 and Terrain2) lead to different compression ratios, regardless of the method used. This is caused by their structural differences leading to variability of the average number of layers per stack (that can be far from the maximum), and the sequences of layers of neighboring stacks. We leave the study of these factors and eventually the development of a measure that could show the compression potential of a layered model for future work (Section 8).

The memory for the input volume data, using 8 bits to encode each voxel, ranges from 53 MB to 163 MB. The voxel representation uses 8 bits per voxel [bpv]. *QuadStack* requires from 3 MB to 19 MB of memory for our datasets, demanding less than 2 bpv for all the scenarios. The achieved compression ratio is between 4× and 54×. It also obtained a more compact representation of the volumetric data than the SBR, except for the Terrain 1 dataset in which SBR achieved the highest compression ratio.

Table 1 also shows the construction time and memory consumption of a standard octree-based volume representation. Further, we report results of the publicly available implementation of the GigaVoxels algorithm [59] that uses an octree enhanced by a brick pool for optimized rendering

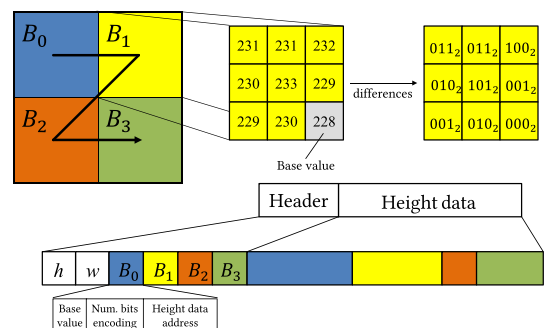
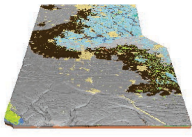
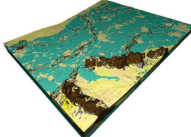
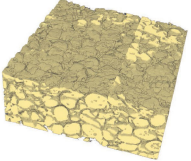
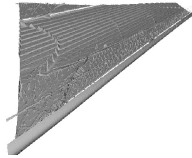
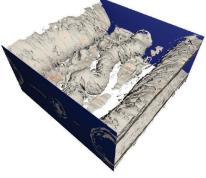


Fig. 9. Heightfield compression scheme.

TABLE 1
Results Measured From Five Test Datasets

					
	Terrain 1	Terrain 2	Battery	Wing	Magnetic
Resolution	1024 × 512 × 250	512 × 512 × 400	512 × 512 × 210	1024 × 1024 × 163	512 × 512 × 512
Max Layers	14	12	34	19	32
Construction Time [sec]					
Voxel Grid	—	—	—	—	—
SBR	4.5	2.2	2.7	8.4	2.1
Octree	1.2	0.6	1.3	1.0	2.3
GigaVoxels (Octree) [59]	1080	180	180	1080	180
Guthe & Goesele [27]	3.2	2.5	2.3	5.0	2.8
<i>QuadStack</i>	7.3	3.5	8.6	2.2	4.3
Memory Size [MB / bpv (ratio)]					
Voxel Grid	125 / 8 (100%)	100 / 8 (100%)	53 / 8 (100%)	163 / 8 (100%)	128 / 8 (100%)
SBR	15 / 0.96 (12%)	6 / 0.48 (6%)	21 / 3.2 (40%)	11 / 0.54 (6.75%)	8 / 0.5 (6.25%)
Octree	35 / 2.24 (28%)	17 / 1.36 (17%)	46 / 7 (87.62%)	7 / 0.34 (4.29%)	26 / 1.63 (20.31%)
GigaVoxels (Octree) [59]	459 / 29.4 (367%)	459 / 36.7 (459%)	459 / 69.3 (866%)	459 / 22.5 (282%)	459 / 28.7 (359%)
Guthe & Goesele [27]	10 / 0.64 (8%)	6 / 0.48 (6%)	10 / 1.52 (19.05%)	9 / 0.44 (5.52%)	6 / 0.37 (4.69%)
<i>QuadStack</i>	19 / 0.60 (15.2%)	5 / 0.40 (5%)	13 / 1.98 (24.5%)	3 / 0.15 (1.84%)	6 / 0.37 (4.69%)
Rendering Performance [ms / frame (ratio)]					
VTK (Voxel Grid)	4.45 (100%)	6.02 (100%)	4.08 (100%)	15.36 (100%)	5.27 (100%)
OSPRay (Voxel Grid)	51.20 (1150%)	40.07 (665%)	51.20 (1255%)	153.60 (998%)	32.91 (624%)
SBR	24.91 (560%)	20.03 (333%)	8.86 (217%)	57.60 (375%)	92.16 (1748%)
GigaVoxels (Octree) [59]	34.48 (775%)	7.74 (192%)	7.46 (183%)	24.63 (160%)	8.50 (161%)
Guthe & Goesele [27]	48.51 (1090%)	92.16 (1530%)	48.50 (1189%)	115.20 (750%)	54.21 (1028%)
<i>QuadStack</i>	12.80 (288%)	17.72 (294%)	38.40 (941%)	51.20 (333%)	11.67 (221%)

The table shows the characteristics of the datasets, construction times, memory requirements, and rendering performance for *QuadStack* and several alternative representations. The results for the method with the lowest memory consumption and best rendering performance are highlighted in bold.

of very large volumes (we used the default settings with brick size 8^3). *QuadStack* provided more compact storage than both Octree and GigaVoxels. The storage requirements and construction times for GigaVoxels are higher due to preallocated fixed size buffers for nodes and bricks and the associated brick pool construction overhead [59]. However, the brick pool used by the GigaVoxels enables efficient filtering and out-of-core rendering that are currently not supported by our implementation of *QuadStack*.

This study is completed with the recent volume compression method of Guthe and Goesele (GG) [27], which is inspired by 2D texture compression techniques. Volume is structured into $4 \times 4 \times 4$ blocks that are independently compressed using the approach that best suits the data in the particular block (constant, difference to the maximum/minimum, gradient or Haar wavelet). Overall, *QuadStack* and GG are comparable in terms of compression performance for the dataset in our experiments. They provided the same compression ratio for Magnetic (5 percent), GG achieved a better compression for Terrain 1 (8 versus 15 percent) and Battery (19 versus 25 percent), and finally, *QuadStack* outperformed GG with Terrain 2 (5 versus 6 percent) and the Wing model (2 versus 6 percent). Beyond these results, GG compresses data without providing any particular insight into it. In contrast, our method holds topological

information of the existing layers, enabling operations such as analysis of the structure of the model, fast generation of a triangle mesh from a given layer, direct extraction or modification of the transparency of a layer during rendering, etc.

A breakdown of the memory budget for *QuadStack* is shown in Table 2. Most of the memory is used by the quadtree and attribute information. The min-max mipmaps require slightly less memory and the compressed heightfields require the least amount of memory. In addition, the amount of memory required to encode raw heightfields is included. The memory required to store the min-max mipmaps is optional; min-max mipmaps act only as rendering acceleration data structure and they are not required for a compression only application.

7.3 Direct Volume Visualization

The second aspect that we have examined is the performance of the GPU-based direct visualization of *QuadStack* by using the five datasets from Section 7.2. As reference, we used two visualization backends implemented in the ParaView software: the GPU accelerated rendering using VTK, and CPU based rendering using OSPRay. For all scenes we used five representative views for which we measured the rendering times that were converted to performance numbers expressed in milliseconds required to generate a frame

TABLE 2
Breakdown of the *QuadStack* Memory Requirements

Dataset	Attributes [MB (%)]	Heightfields [MB raw / compressed (%)]	Mipmaps [MB (%)]	Total [MB (%)]
Terrain 1	11.4 (59%)	5.9/3.3 (17%)	4.7 (24%)	19.4 (100%)
Terrain 2	2.9 (62%)	2.3/1.0 (21%)	0.8 (17%)	4.7 (100%)
Battery	6.4 (49%)	8.3/4.6 (36%)	1.9 (15%)	13.1 (100%)
Wing	1.3 (52%)	3.1/0.8 (32%)	0.4 (16%)	2.5 (100%)
Magnetic	3.4 (58%)	4.6/1.5 (25%)	1.0 (17%)	5.9 (100%)

The columns represent memory needed for representing the quadtree and attributes, raw and compressed heightfields, and min-max mipmaps.

[ms/frame]. To provide a more concise overview, we report average performance for each scene and method using values averaged over different views.

The lower part of Table 1 shows an overview of the rendering performance results. VTK uses a highly optimized GPU ray caster of uncompressed volume data and we can observe that it achieves the highest rendering performance for most camera positions, generally followed by the Giga-Voxels octree-based representation. The *QuadStack* rendering is between $2.21 - 9.42\times$ slower. The OSPRay renderer is CPU-based and it generally achieves the lowest performance on the tested scenes. It is $1.3 - 4\times$ slower than the direct *QuadStack* visualization. The direct SBR rendering uses an equidistant sampling to satisfy the Nyquist-Shannon sampling theorem. The direct SBR rendering is slower than the *QuadStack* (between $1.13 - 7.9\times$) for all datasets excluding the Battery scenario where the SBR is $4.33\times$ faster.

QuadStack performs better than the rest of renderers/techniques when a ray has to skip a large empty space that is encoded in high levels of the tree structure (Terrain 1-2 and the Wing). As we highlighted in the previous subsection, the lack of structure of the compressed data in [27] also penalizes the efficiency of this empty space step. However, the VTK renderer provided a better overall performance. This is primarily due to the heightfield decompression overhead implicit in each sampling step of the *QuadStack* traversal. Since *QuadStack* uses just a fraction of memory required by the VTK renderer, it provides a good tradeoff between memory usage and rendering performance.

Fig. 10 shows the time [ms] required to generate each dataset. Our rendering method performs in realtime or close to it in every scenario in resolution of $1,280 \times 720$. Terrain1 presents an outlier in one camera position. This results from rendering the dataset from the bottom and consequently no empty space-skipping was required.

8 CONCLUSION AND FUTURE WORK

We introduced *QuadStack*, a novel algorithm for layered data compression and direct rendering. The key inspiration for our work is the common output of many science and engineering applications and measurements that produce data with strong directional anisotropy in the form of layers.

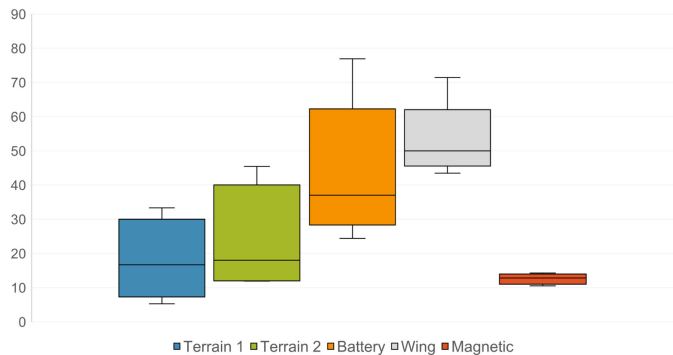


Fig. 10. Rendering results [ms] per frame. The whisker plot show the time distribution.

QuadStack compresses the layers into stacks and then compresses the stacks into a *quadtree* while considering the representing patterns among neighboring layers. We also introduce a novel algorithm for direct rendering of the compressed data and we show its GPU implementation that performs comparably to state-of-the-art algorithms for direct volume rendering, but instead of using full data it works directly with the compressed volumes.

Our method has several specific advantages that are possible for layered models. It allows for the extraction of an individual layer during rendering and its conversion to a triangle mesh on-the-fly if required. Layers can also be individually hidden/visible, which is relevant in many practical fields such as geology. We can also render layers that include water by using transparency or even refraction effects. Finally, it supports lossless and lossy compression (within the limit of numerical representation).

While the field of data compression and rendering has been active for many years, there are still many open problems that may have been enabled by our algorithm. Our algorithm could be extended to time-varying datasets that are common in fluid simulations or simulations of eroded terrains. Also, many datasets are cylindrical and it would be an interesting extension to apply *QuadStack* to a non-linear domain. We have not fully explored the internal structure of the layers and its relation to the compression factor. It would be possible to first sample and rotate the input data to detect a direction that would provide good compression factor. The construction algorithm uses rather simple matching and a possible extension would improve its efficiency for scenes with many layers. We would also like to study the possibility of using DAGs [24], [25], [26] for compressing the layer attributes as well as the layer geometry for high resolution data sets.

ACKNOWLEDGMENTS

This research was funded in part by National Science Foundation Grant #10001387, *Functional Proceduralization of 3D Geometric Models*, the Research Center for Informatics No. CZ.02.1.01/0.0/0.0/16_019/0000765, the Czech Science Foundation under Project GA18-20374S, the Ministry of Science and Innovation of Spain under Projects TIN2017-84968-R and RTI2018-099638-B-I00 and the University of Jaén. The authors would like to thank Niels Aage for providing the Wing model and Daniel and Carmen Benes-Magana as well as Colin Gray for proofreading the paper.

REFERENCES

- [1] J. L. Gunnink, D. Maljers, S. F. Van Gessel, A. Menkovic, and H. J. Hummelman, "Digital geological model (DGM): A 3D raster model of the subsurface of the Netherlands," *Geologie en Mijnbouw/Netherlands J. Geosciences*, vol. 92, no. 1, pp. 33–46, 2013.
- [2] N. Aage, E. Andreassen, B. S. Lazarov, and O. Sigmund, "Giga-voxel computational morphogenesis for structural design," *Nature*, vol. 550, pp. 84–86, 2017.
- [3] M. Ebner, F. Geldmacher, F. Marone, M. Stampanoni, and V. Wood, "X-ray tomography of porous, transition metal oxide based lithium ion battery electrodes," *Adv. Energy Mater.*, vol. 3, no. 7, pp. 845–850, 2013.
- [4] F. Guo, H. Li, W. Daughton, and Y.-H. Liu, "Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection," *Phys. Rev. Lett.*, vol. 113, 2014, Art. no. 155005.
- [5] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Proc. European Comput. Graph. Conf. Exhibition*, 1987, pp. 3–10.
- [6] T. T. Elvins, "A survey of algorithms for volume visualization," *Comput. Graph.*, vol. 26, no. 3, pp. 194–201, 1992.
- [7] J. Revelles, C. Ureña, and M. Lastra, "An efficient parametric algorithm for octree traversal," *J. WSCG*, vol. 8, no. 1–3, pp. 212–219, 2000.
- [8] M. Levoy, "Efficient ray tracing of volume data," *ACM Trans. Graph.*, vol. 9, no. 3, pp. 245–261, 1990.
- [9] P. Muigg, M. Hadwiger, H. Doleisch, and M. E. Gröller, "Interactive volume visualization of general polyhedral grids," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2115–2124, Dec. 2011.
- [10] B. Benes and R. Forsbach, "Layered data representation for visual simulation of terrain erosion," in *Proc. 17th Spring Conf. Comput. Graph.*, 2001, pp. 80–86.
- [11] G. Cordonnier, M.-P. Cani, B. Benes, J. Braun, and E. Galin, "Sculpting mountains: Interactive terrain modeling based on subsurface geology," *IEEE Trans. Vis. Comput. Graphics*, vol. 24, no. 5, pp. 1756–1769, May 2018.
- [12] G. Cordonnier et al., "Authoring landscapes by combining ecosystem and terrain erosion simulation," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 134:1–134:12, 2017.
- [13] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou, "Arches: A framework for modeling complex terrains," *Comput. Graph. Forum*, vol. 28, pp. 457–467, 2009.
- [14] F. Löffler, A. Müller, and H. Schumann, "Real-time rendering of stack-based terrains," in *Vision, Modeling, and Visualization*, P. Eisert, J. Hornegger, and K. Polthier, Eds., Aire-la-Ville, Switzerland: The Eurographics Association, 2011.
- [15] A. Graciano, A. J. Rueda, and F. R. Feito, "Real-time visualization of 3D terrains and subsurface geological structures," *Adv. Eng. Softw.*, vol. 115, pp. 314–326, 2018.
- [16] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-art in GPU-based large-scale volume visualization," *Comput. Graph. Forum*, vol. 34, no. 8, pp. 13–37, 2015.
- [17] J. Danskin and P. Hanrahan, "Fast algorithms for volume ray tracing," in *Proc. Workshop Volume Vis.*, 1992, pp. 91–98.
- [18] D. Cohen and Z. Sheffer, "Proximity clouds — an acceleration technique for 3d grid traversal," *Vis. Comput.*, vol. 11, no. 1, pp. 27–38, 1994.
- [19] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller, "Memory efficient acceleration structures and techniques for CPU-based volume raycasting of large data," in *Proc. IEEE Symp. Volume Vis. Graph.*, 2004, pp. 1–8.
- [20] S. Lim and B.-S. Shin, "A distance template for octree traversal in CPU-based volume ray casting," *Vis. Comput.*, vol. 24, no. 4, pp. 229–237, 2008.
- [21] C. Crassin, F. Neyret, M. Sainz, and E. Eisemann, *Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels*. Natick, MA, USA: A. K. Peters, 2010, pp. 643–676.
- [22] S. Laine and T. Karras, "Efficient sparse voxel octrees," *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 8, pp. 1048–1059, Aug. 2011.
- [23] M. Balsa Rodríguez et al., "State-of-the-art in compressed GPU-based direct volume rendering," *Comput. Graph. Forum*, vol. 33, no. 6, pp. 77–100, 2014.
- [24] V. Kämpe, E. Sintorn, and U. Assarsson, "High resolution sparse voxel dags," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 101:1–101:13, 2013.
- [25] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann, "Geometry and attribute compression for voxel scenes," *Comput. Graph. Forum*, vol. 35, no. 2, pp. 397–407, May 2016.
- [26] D. Dolonius, E. Sintorn, V. Kämpe, and U. Assarsson, "Compressing color data for voxelized surface geometry," *IEEE Trans. Vis. Comput. Graphics*, vol. 25, no. 2, pp. 1270–1282, Feb. 2019. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/TVCG.2017.2741480>
- [27] S. Guthe and M. Goesele, "Variable length coding for GPU-based direct volume rendering," in *Proc. Conf. Vis. Modeling Vis.*, 2016, pp. 77–84. [Online]. Available: <https://doi.org/10.2312/vmv.20161345>
- [28] I. Wald and V. Havran, "On building fast KD-trees for ray tracing, and on doing that in $O(n \log n)$," in *Proc. IEEE Symp. Interactive Ray Tracing*, 2006, pp. 61–69. [Online]. Available: doi.ieeecomputersociety.org/10.1109/RT.2006.280216
- [29] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *Proc. IEEE Symp. Interactive Ray Tracing*, 2007, pp. 33–40.
- [30] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proc. Proc. Conf. High Performance Graph.*, 2009, pp. 7–13.
- [31] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A kernel framework for efficient cpu ray tracing," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 143:1–143:8, 2014.
- [32] I. Wald et al., "Ospray - a CPU ray tracing framework for scientific visualization," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 931–940, Jan. 2017.
- [33] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proc. Conf. High Perform. Graph.*, 2009, pp. 145–149.
- [34] S.-E. Yoon and D. Manocha, "Cache-efficient layouts of bounding volume hierarchies," *Comput. Graph. Forum*, vol. 25, no. 3, pp. 507–516, 2006.
- [35] H. Ylitie, T. Karras, and S. Laine, "Efficient incoherent ray traversal on GPUs through compressed wide BVHs," in *Proc. High Perform. Graph.*, 2017, pp. 4:1–4:13.
- [36] C. Benthin, I. Wald, S. Woop, and A. T. Áfra, "Compressed-leaf bounding volume hierarchies," in *Proc. Conf. High-Perform. Graph.*, 2018, pp. 6:1–6:4.
- [37] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner, "Real-time, continuous level of detail rendering of height fields," in *Proc. 23rd Annu. Conf. Comput. Graph. Interactive Techn.*, 1996, pp. 109–118.
- [38] H. Hoppe, "Progressive meshes," in *Proc. 23rd Annu. Conf. Comput. Graph. Interactive Techn.*, 1996, pp. 99–108.
- [39] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "Roaming terrain: real-time optimally adapting meshes," in *Proc. IEEE Vis.*, 1997, pp. 81–88.
- [40] H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *Proc. IEEE Vis.*, 1998, pp. 35–42.
- [41] F. K. Musgrave, "Grid tracing: Fast ray tracing for height fields," Dept. Comput. Sci., Yale University, New Haven, CT, Rep. no. 639, 1988.
- [42] D. Cohen-Or and A. Shaked, "Photo-realistic imaging of digital terrains," *Comput. Graph. Forum*, vol. 12, no. 3, pp. 363–373, 1993.
- [43] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar, "A real-time photo-realistic visual flythrough," *IEEE Trans. Vis. Comput. Graphics*, vol. 2, no. 3, pp. 255–264, Sep. 1996.
- [44] C. Henning and P. Stephenson, "Accelerating the ray tracing of height fields," in *Proc. Int. Conf. Comput. Graph. Interactive Techniques Australasia Southeast Asia*, 2004, pp. 254–258.
- [45] C. Dick, J. H. Krüger, and R. Westermann, "GPU ray-casting for scalable terrain rendering," in *Eurographics (Areas Papers)*. Munich, Germany: Eurographics Association, 2009, pp. 43–50.
- [46] L. Ammann, O. Gènevaux, and J.-M. Dischler, "Hybrid rendering of dynamic heightfields using ray-casting and mesh rasterization," in *Proc. Graph. Interface*, 2010, pp. 161–168.
- [47] C. Lux and B. Fröhlich, "GPU-based ray casting of stacked out-of-core height fields," in *Proc. Int. Symp. Vis. Comput.*, 2011, pp. 269–280.
- [48] L. Baboud, E. Eisemann, and H.-P. Seidel, "Precomputed safety shapes for efficient and accurate height-field rendering," *IEEE Trans. Vis. Comput. Graphics*, vol. 18, no. 11, pp. 1811–1823, Nov. 2012.
- [49] E.-S. Lee, J.-H. Lee, and B.-S. Shin, "A bimodal empty space skipping of ray casting for terrain data," *J. Supercomput.*, vol. 72, no. 7, pp. 2579–2593, 2016.
- [50] L. Scandolo, P. Bauszat, and E. Eisemann, "Compressed multiresolution hierarchies for high-quality precomputed shadows," *Comput. Graph. Forum*, vol. 35, no. 2, pp. 331–340, May 2016.

- [51] R. Pajarola and E. Gobbetti, "Survey of semi-regular multiresolution models for interactive terrain rendering," *Vis. Comput.*, vol. 23, no. 8, pp. 583–605, 2007.
- [52] C. S. Iliopoulos, J. Mchugh, P. Peterlongo, N. Pisanti, W. Rytter, and M.-F. Sagot, "A first approach to finding common motifs with gaps," *Int. J. Found. Comput. Sci.*, vol. 16, no. 06, pp. 1145–1154, 2005.
- [53] P. Antoniou, J. Holub, C. S. Iliopoulos, B. Melichar, and P. Peterlongo, "Finding common motifs with gaps using finite automata," in *Proc. Int. Conf. Implementation Appl. Automata.*, 2006, pp. 69–77.
- [54] P. Antoniou, M. Crochemore, C. S. Iliopoulos, and P. Peterlongo, "Application of suffix trees for the acquisition of common motifs with gaps in a set of strings," in *Proc. 1st Int. Conf. Lang. Autom. Theory Appl.*, 2007, pp. 57–66.
- [55] W. R. Franklin, "Compressing elevation data," in *Advances in Spatial Databases*, M. J. Egenhofer and J. R. Herring, Eds. Berlin, Germany: Springer, 1995, pp. 385–404.
- [56] C. Andújar, "Topographic map visualization from adaptively compressed textures," *Comput. Graph. Forum*, vol. 29, no. 3, pp. 1083–1092, 2010.
- [57] 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.
- [58] A. Tevs, I. Ihrke, and H.-P. Seidel, "Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering," in *Proc. Symp. Interactive 3D Graph. Games*, 2008, pp. 183–190.
- [59] C. Crassin, F. Neyret, S. Lefebvre, and É. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proc. Symp. Interactive 3D Graph. Games*, 2009, pp. 15–22.
- [60] S. Lefebvre, S. Hornus, and F. Neyret, "Octree Textures on the GPU," in *Programming Techniques for High-performance Graphics and General-Purpose Computation*, P. M. editors, Ed. Reading, MA, USA: Addison Wesley, 2005, pp. 595–613.



Alejandro Graciano received the PhD degree in computer science from the University of Jaén, Spain, in 2019. He is currently a postdoctoral researcher at University of Jaén, Spain. His research interests include topics in computer graphics such as geoscientific visualization, GPU programming and its applications as well as geographic information systems.



Antonio J. Rueda is currently an associate professor of Computer Science at the Escuela Politécnica Superior, University of Jaén, Spain. His main research interests include computer graphics, focusing on design of geometric algorithms, processing of 3D laser scanned data or GPU computing. He has presented his work in more than 40 papers and communications in journals and conferences.



Adam Pospíšil received the BS degree from the Czech Technical University, Prague. He is currently a researcher at the Czech Technical University, in Prague. His main area of research is in spatial data structures, and global illumination.



Jiří Bittner received the PhD degree from the Czech Technical University in Prague. He is currently an associate professor of Computer Science at the Czech Technical University, in Prague. His research interests include visibility computations, real-time rendering, spatial data structures, and global illumination.



Bedrich Benes (Senior Member, IEEE) is currently a George McNelly professor of Technology and professor of Computer Science at Purdue University. His area of research interests include procedural and inverse procedural modeling and simulation of natural phenomena and he has published more than 140 research papers in the field.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.