J Supercomput (2010) 51: 3–19
DOI 10.1007/s11227-009-0289-2

Enabling the interactive display of large medical volume datasets by multiresolution bricking

Anupam Agrawal · Josef Kohout · Gordon J. Clapworthy · Nigel J.B. McFarlane · Feng Dong · Marco Viceconti · Fulvia Taddei · Debora Testi

Published online: 19 April 2009

Abstract In this paper, we present an approach to interactive out-of-core volume data exploration that has been developed to augment the existing capabilities of the LhpBuilder software, a core component of the European project LHDL (http://www.biomedtown.org/biomed_town/lhdl). The requirements relate to importing, accessing, visualizing and extracting a part of a very large volume dataset by interactive visual exploration. Such datasets contain billions of voxels and, therefore, several gigabytes are required just to store them, which quickly surpass the virtual address limit of current 32-bit PC platforms. We have implemented a hierarchical, bricked, partition-based, out-of-core strategy to balance the usage of main and external memories. A new indexing scheme is introduced, which permits the use of a multiresolution bricked volume layout with minimum overhead and also supports fast data compression. Using the hierarchy constructed in a pre-processing step, we generate a coarse approximation that provides a preview using direct volume visualization for large-scale datasets. A user can interactively explore the dataset by specifying a region of interest (ROI), which further generates a much more accurate data representation inside the ROI. If even more precise accuracy is needed inside the ROI, nested ROIs are used. The software has been constructed using the Multimod Application Framework, a VTK-based system; however, the approach can be adopted for the other systems in a straightforward way. Experimental results show that the user can interactively explore large volume datasets such as the Visible Human Male/Female (with

A. Agrawal · J. Kohout · G.J. Clapworthy (✉) · N.J.B. McFarlane · F. Dong
Department of Computer Science & Technology, University of Bedfordshire, Luton, UK
e-mail: gordon.clapworthy@beds.ac.uk

M. Viceconti · F. Taddei
Laboratorio di Tecnologia Medica, Istituto Ortopedico Rizzoli, Bologna, Italy

D. Testi
BioComputing Competence Centre, SCS, Casalecchio di Reno, Italy

file sizes of 3.15/12.03 GB, respectively) on a commodity graphics platform, with ease.

Keywords Medical visualization · Large volume data sets · Out-of-core processing · Multiresolution bricking · VTK

1 Introduction

Sophisticated data-acquisition techniques and complex simulations are producing volume datasets of continually increasing size. At the highest resolution, they cannot be visualized interactively in one piece on a single commodity computer. In the past, various solutions for visualizing large volume datasets have been proposed—a common feature is that they require an excellent cooperation between the rendering algorithm and the data-retrieval algorithm.

Multiresolution techniques for rendering large volume data [5, 7, 10] construct a hierarchical data structure (e.g., octree [10], multilevel pyramid [15]) during a preprocessing stage). The lowest level in this structure (i.e., leaves) retains the original data, while higher levels (inner nodes) contain increasingly lower-resolution approximations of the data. The lower levels are stored on hard disk and transferred to main memory, on demand, [21] at runtime.

These methods typically display the volume in a region of interest at a high resolution and the volume away from that region at progressively lower resolutions. This makes interactive visualization particularly challenging because the generation of the next image may have to wait for a significant amount of time while the data required is read from a disk.

Parallel visualization is also very often used to address the issues of large data processing [2, 4, 15, 22]. The overwhelming majority of such cases employ a divide-and-conquer strategy that involves the subdivision of the large volume dataset into parts small enough to be processed on a single processor, successive, or simultaneous processing (e.g., visualization) of these parts, and the merging of outcomes to produce the final result. The approaches differ in how they subdivide and merge the data (which may be quite complex, especially for transparent objects [14]). These approaches are often combined with multiresolution techniques.

There are also approaches that use wavelet compression to reduce the size of the volume data, and thus make the retention of the entire dataset in memory possible or speed up the transmission of data over the network in parallel visualization [9, 22].

Although many of the existing approaches are able to visualize, or at least to explore, volume datasets of tens of GB in real-time on a cluster of workstations (e.g., [4]), their integration into already existing visualization systems might be complex, or even impossible, for the following reasons. First, these approaches are usually problem specific, e.g., they are suitable for visualizing medical volume datasets using only an iso-surface volume rendering technique [15], while visualization systems typically aim to be able to visualize volume datasets of any kind using a variety of volume rendering techniques [7, 17], and most systems also provide the user with volume smoothing, segmentation operations, color mapping functions, etc. Further,

these systems were very often designed to run only on a single computer, rather than a cluster.

Examples of these systems are as follows. MayaVi Data Visualizer [18] and MVE-2 [8] are general-purpose scientific-visualization environments organized around the data-flow paradigm. 3D Doctor [1], and Mimics [13] are 3D visualization applications that excel in image processing activities such as segmentation. 3D Slicer [16] is an application mostly focused on neurosurgery problems. Other systems, such as OsiriX [19] and the Multimod Application Framework (MAF) [24] are specialized in the processing of medical images.

Researchers using these systems have three options: not to process large volume datasets at all; to redesign their system (which involves modifying many volume operations and the system core); or to exploit an alternative, less-efficient, approach that is, however, easier to adopt.

This paper proposes an efficient out-of-core method for the exploration of large volume data that is suitable for data-flow-oriented visualization systems (especially for those based on VTK [20]). It improves the performance of out-of-core exploration of large volume data based on the following two key concepts. Firstly, we use knowledge of the storage order of the data on disk and of the access pattern required by the visualization algorithm to optimize disk access. Secondly, we create discrete multiresolution representations of the data—we use a low-resolution version for representing the entire dataset, and a higher-resolution version only for visualizing smaller regions of interest.

This approach allows the user to roam through multi-gigabyte volume datasets in real time, with low memory requirements. The proposed indexing scheme permits the use of a multiresolution bricked volume layout with minimum overhead. Additionally, it applies fast compression to reorganize the volume data by skipping empty voxels, which usually occupy a large proportion of the space in medical datasets, such as the National Library of Medicine's Visible Human (VH) dataset [23].

The paper is organized as follows. The background of the research is given in Sect. 2, and the proposed hierarchical, bricked, partition-based, out-of-core strategy is explained in Sect. 2. Results achieved with the proposed technique are presented in Sect. 4. Finally, in Sect. 5, concluding remarks and future work are described.

2 Background

The Visualization ToolKit (VTK) [20] is an open source, freely available software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers around the world. VTK consists of a C++ class library, and several interpreted interface layers including Tcl/Tk, Java, and Python. VTK supports a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods; and advanced modeling techniques such as implicit modeling, polygon reduction, mesh smoothing, cutting, and contouring.

In a VTK application, the process of visualization can be described in terms of data flow through the so-called visualization pipeline. The visualization pipeline consists of process objects that operate on input data and transform it into output data. These

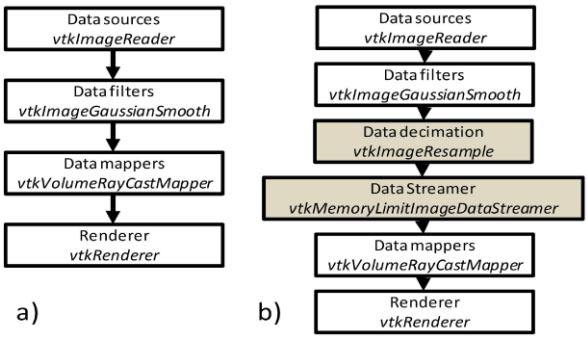

Fig. 1 VTK visualization pipeline: (**a**) without, and (**b**) with, streaming support; an example for visualizing volume is given in *italics*

objects can be divided into three main categories: sources, filters and mappers. The source objects have no input and have at least one output. The filters have at least one input and one output, and the mappers have at least one input but no output. An example of a typical VTK pipeline is depicted in Fig. 1a.

The majority of process objects hold the data in main memory, which makes the processing of large datasets using a common approach impossible. VTK, therefore, provides streaming support to process larger datasets; see Fig. 1b. Using the streaming technique, a dataset is processed piece by piece: a subset (piece) is loaded into memory, processed through the classic VTK visual pipeline and the results obtained are decimated (e.g., subsampled) and successively combined together so they can be visualized. If, however, we consider roaming through a large volume dataset, this solution is somewhat inefficient, as it needs to process the entire dataset whenever the region of interest (ROI) changes.

Despite its limitations, VTK is a basis for many visualization systems, e.g., MayaVi [18], 3D Slicer [16], and OsiriX [19], while in others, it can be used optionally, e.g., MVE-2 [8].

The Multimod Application Framework (MAF) [24] is another visualization system based on VTK (and other specialized libraries). This framework is designed to support the rapid development of biomedical software. It allows the development of multimodal visualization applications, which supports the fusion of data from multiple sources and in which different views of the same data are synchronized, so that when the position of an object changes in one view, it is updated in all the other views.

There are four main components of MAF: *Virtual Medical Entities* (VME), which are the data objects; *Views*, which provide interactive visualization of the VMEs; *Operations*, which create new VMEs or modify existing ones; and the *Interface Elements*, GUI components that define the user interface of the application. Special operations are importers, which import and convert almost any biomedical dataset (e.g., RAW, DICOM, STL, etc.) into VME form, and exporters, which convert the VME into files formatted according to the most common standards.

One method of creating a VME is by importing digital datasets. However, the importers for volume datasets and operations on the imported volume VME inside MAF are limited by the data size, which is typically a few hundred megabytes—this limitation is primarily due to the constraints imposed by VTK on handling large data. The relationship between MAF components and the VTK visualization pipeline is shown in Fig. 2.

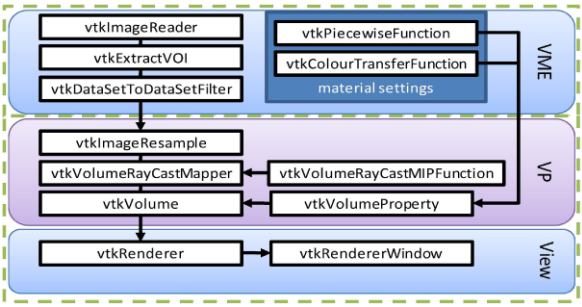

Fig. 2 An example of the VTK pipeline and its encapsulation in MAF. Note that from the user's point of view, VP (which is the visual pipe within MAF) is a part of View

MAF is exploited in LhpBuilder, which is software being developed within the Living Human Digital Library [11] project (LHDL). This three-year project, which concluded in January 2009, aimed to create the technical infrastructure for another ongoing initiative: the Living Human Project (LHP). The LHP is a long-term project that aims to create an in-silico model of the human musculo-skeletal apparatus which can predict how mechanical forces are exchanged internally and externally at any dimensional scale from the whole body down to the protein level. This model has been designed as an infrastructure that can be updated and extended whenever new data and algorithms become available.

The main objective of the research reported in this paper was to add a new capability to LhpBuilder for importing, accessing, visualizing, and extracting part of a large volume dataset by interactive visual exploration. For the visualization of limited-size volume data, the existing volume visualization functions of LhpBuilder were used, which are based on DRR (Digital Reconstructed Radiograph), MIP (Maximum Intensity Projection), iso-surface algorithms [17, 20], and others.

3 Approach

As mentioned earlier, an efficient algorithm for exploring large volume data sets requires excellent cooperation between the rendering and data-retrieval algorithms. Within the VTK scheme, however, rendering and data retrieval are two independent processes; this introduces some limitations.

This problem is even more serious in MAF applications, e.g., in LhpBuilder, (though a similar problem may be expected in other systems), because the VME, which incorporates an appropriate data-retrieval algorithm, is not aware of the View that will be used to visualize it, and the View (which contains the rendering algorithm) does not know which VME it will visualize. Under these circumstances, all we can do when visualizing large volume datasets is to prepare a subsampled snapshot of the requested region of interest (ROI) that has memory requirements below the predefined memory limit, and let the system visualize this snapshot.

The straightforward approach would be to have a VME that reads the original volume dataset (typically stored in a RAW file) and subsamples it whenever the requested ROI or memory limit changes. However, this would be time consuming and

would not support interactive visual exploration. We propose, therefore, a more efficient way of data retrieval based on a discrete multi-resolution model, combined with a bricking technique during preprocessing.

3.1 Preprocessing

When a user wants to import a volume that is larger than some given threshold, the volume is not loaded into the memory using the regular approach (*vtkImageReader*) but is preprocessed to produce subsampled versions (resolution levels), which are stored on disk in an optimized form.

Starting with a sample rate (*SR*) equal to one, and incrementing this by one at every step, the algorithm successively samples the input dataset until the size of the lowest resolution level drops below some given threshold T. We decided to use $T = 1$ MB as we found experimentally that data at this resolution can be visualized in almost real-time on commodity hardware even by the slowest rendering technique (supported by LhpBuilder). If we consider cubic volumes with N voxels in one dimension, the number of levels k to be constructed can be computed as: $k^3 \geq \frac{N^3}{T=1\text{MB}} \Rightarrow k \approx \frac{N}{100}$. Assuming that there are 8 bits per voxel, the total size S of all constructed levels is then: $S = \sum_{i=1}^{k} (\frac{N}{i})^3 = N^3 \sum_{i=1}^{k} \frac{1}{i^3}$ bytes. If we recall the Riemann zeta function [6] $\zeta(s)$ of a complex variable s that is defined, for s with real part > 1, by the following infinite series: $\zeta(s) = \sum_{i=1}^{\infty} \frac{1}{i^s}$, we can write that $S \leq N^3 \zeta(3)$. The value $\zeta(3)$ is known as Apéry's constant and is approximately 1.202. Thus, we can conclude that the total size of the constructed levels is less than $1.21 \cdot N^3$ bytes. The dependency of the number of levels to the volume data size is shown in Table 1. We note that for arbitrary volume datasets, the number of levels may vary slightly from these.

The processing of the input dataset has three main steps: sampling, bricking, and fast compression; see Fig. 3. As the input dataset is too large to be fully loaded into memory, it must be processed in smaller parts, so these steps must be repeated for each part. In the first step, the loaded part is subsampled (using the sample rate *SR*). Next, the voxels are reorganized into blocks (bricks) and finally they are stored (except for bricks containing voxels that all have the same value) in the output file. We shall now describe these steps in detail.

Given the sample rate *SR* and the number of levels to be constructed k, the algorithm computes from these values the brick size *BS*, which is the number of voxels per one side of brick (its meaning will be explained later in this section), using the following formula: $BS = \max(4, \min(16, \lfloor k - (SR - 1) \rfloor_4))$ where $\lfloor x \rfloor_r$ is a function that rounds x down to the integer value divisible by the constant r, e.g., $\lfloor 14 \rfloor_4 = 12$.

Table 1 Relationship between the number of resolution levels and the volume data size

Dimensions	Size [GB]	Levels (k)
$1024 \times 1024 \times 1024$	1	11
$1280 \times 1280 \times 1280$	2	13
$1625 \times 1625 \times 1625$	4	16
$2048 \times 2048 \times 2048$	8	21
$2600 \times 2600 \times 2600$	16	26

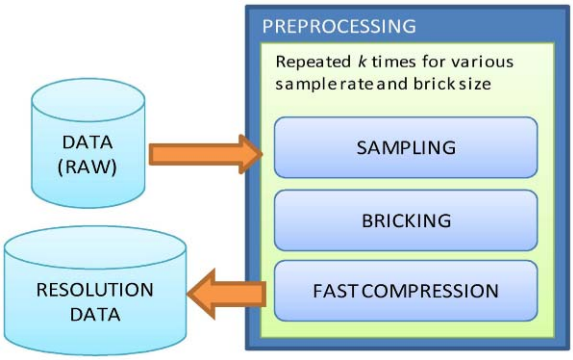

Fig. 3 The preprocessing scheme

0	0	0	0	0	2	2	2	2	0	0	0	0
0	0	0	0	**0**	2	2	2	**2**	2	0	0	**0**
0	0	0	2	2	5	5	2	5	5	2	0	0
0	0	2	2	2	5	5	2	5	5	2	2	0
0	0	2	2	2	2	2	2	2	2	2	2	0
0	0	2	2	**2**	4	2	2	**2**	4	2	2	**0**
0	0	2	2	2	4	4	4	4	4	2	2	0
0	0	0	2	2	2	4	4	4	2	2	0	0
0	0	0	0	2	2	2	2	2	2	0	0	0
0	0	0	0	**0**	2	2	2	**2**	0	0	0	**0**

Fig. 4 Sampling one slice with a sample rate of $SR = 4$; only the light lines are loaded from the file and only the underlined voxels are retained

Actually, this means that the BS constant takes one of values: 4, 8, 12, and 16. The construction of this formula was motivated by several facts. First, we found experimentally (see Sect. 4) that minimal data retrieval times were achieved with the value $BS = 16$. This value also proved to be optimal for the involved fast compression of the highest resolution that brings additional speed-up. However, when the data is sampled, e.g., with $SR = 4$, a $16 \times 16 \times 16$ brick would cover a big region ($64 \times 64 \times 64$ for $SR = 4$), thus reducing the efficiency of the compression algorithm. While smaller values are acceptable, it appears that bricks with $BS = 2$ are too small and produce a large overhead.

The input file is then read line-by-line, slice-by-slice,[1] skipping lines, and slices that do not correlate with the given sample rate. When a line is loaded, it is sampled and the samples are stored in a buffer capable of holding BS sampled slices. Hence, if we have a data set of $N \times N \times N$ voxels, $N \cdot \frac{N}{SR} \cdot \frac{N}{SR}$ voxels are progressively loaded and sampled.

For example, let us suppose we have volume data of $13 \times 10 \times 10$ voxels. For $SR = 4$, only lines 0, 4, 8, in slices 0, 4, 8 are loaded and sampled, giving four voxels per line to be stored; see Fig. 4 (in which line 0 is at the bottom). Hence, if BS is 2, the buffer must be capable of storing 24 samples.

[1] A slice is formed from lines and a line is formed from voxels.

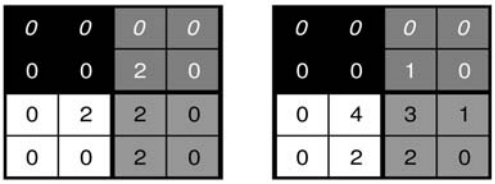

Fig. 5 Samples of two slices organized into $2 \times 2 \times 2$ bricks (the first corresponds to Fig. 4). Missing samples are given in *italics*. Bricks are denoted by *grey* scale

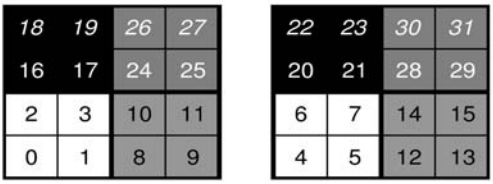

Fig. 6 The storing order of samples

As the process is repeated for every *SR*, the data is read several times, which negatively influences the overall time needed for the preprocessing stage. It can be avoided, if loaded lines were sampled by different sample rates but it would make the algorithm more difficult and it would also consume more memory. Anyway, this extra reading is not a serious problem since the preprocessing has to be performed only once for any set of data. The overall number of voxels to be loaded in all iterations is: $\sum_{i=1}^{k} N \cdot (\frac{N}{i})^2 = N^3 \sum_{i=1}^{k} \frac{1}{i^2} \leq N^3 \sum_{i=1}^{\infty} \frac{1}{i^2} = N^3 \zeta(2)$. As $\zeta(2) = \frac{\pi^2}{6} \approx 1.644934$ (see [6]), the preprocessing algorithm must read about $1.65 \cdot N^3$ voxels.

When the buffer is full, its samples can be immediately stored in the output file. However, this straightforward strategy may not be the best because it would store two voxels that are adjacent in the z-direction at two totally different places in the file. This may lead to inefficient use of disk cache because the effectiveness of current disk I/O algorithms is based on the assumption that the file is likely to be read (either entirely, or in part) in the order in which the data items were originally stored in the file. This assumption is very often false when applied to multidimensional scientific data such as volume data, which represents data in a 3D spatial domain [12].

To circumvent this problem, samples are grouped into bricks of $BS \times BS \times BS$ samples. If there are not enough samples to form a brick, the missing samples are considered to be zero; see Fig. 5. After that, bricks can be stored, one brick after another, into the output file. The order in which the samples are stored is illustrated in Fig. 6. This strategy ensures that when the user explores a small portion of volume data, the voxels to be retrieved will lie close to each other in the storage.

It may happen that some bricks contain samples that all have the same value, i.e., these bricks are uniform (like the top left brick in Fig. 5). This case is quite common, especially for medical data which frequently contains a lot of empty space; see Fig. 7. It is not necessary to store such a brick in the normal way—we simply need to store one sample (the uniform value). This not only reduces the size of the resolution level but may also speed up the data retrieval as less data has to be transferred from the disk into the memory.

The algorithm computes an average value for every brick and stores this value into the "brick map" constructed in memory. This map is stored into the output file at the

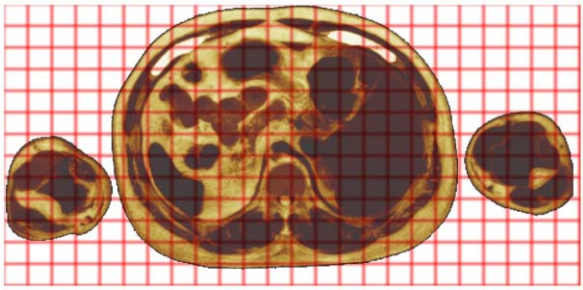

Fig. 7 An example of medical data with highlighted brick boundaries

end of process, optimally at the beginning of the file. An advantage of this map is that the retrieval algorithm can very easily find the value for a skipped uniform brick. This map can also be used for a fast preview of the data, which may be very useful, particularly if the data is stored remotely.

There are many ways to store the positions of the uniform bricks. A straightforward approach is to store, for every brick, a single bit denoting whether the brick is uniform or not, followed by several bits identifying the address in the output file. It can be shown that the largest number of bricks is obtained when $SR = 1$.

The required memory (in bytes) can be, therefore, computed as: $\frac{N^3 \cdot c}{16^3 \cdot 8}$ where c is the number of bits consumed per brick. When dealing with volumes up to 32 GB, at least 24 bits per brick are required assuming that the position is stored in brick units ($16^3 = 4096$). For VH Male volume data [23] (see also Sect. 4) of $1760 \times 1024 \times 1878$ voxels that is bricked into $110 \times 64 \times 118$ bricks, 2.38 MB (24 bits per brick) are required.

Considering medical data (see Fig. 7), this seems to be unnecessarily space consuming, as there is typically only one block of nonuniform bricks in any line of bricks. We therefore propose another indexing strategy. For every line of bricks, we store the number of bricks skipped in previous lines and a list of pairs of indices identifying the beginning and the end of each block of nonuniform bricks. This information is retained in two separate tables. The primary table has a fixed number of entries—one entry per one brick line. It stores the number of skipped bricks (using a 32-bit integer), the length of the list (8 bits), the first pair of indices (using 16 bits per index), and the index (24 bits), which points into the secondary table in which the remaining pairs are stored in linear order. Clearly, the number of entries in the secondary table is data dependent.

Figure 8 shows an example of these tables for the medical data from Fig. 7. In this figure, S designates the number of bricks skipped so far, L is the list length, B and E are the beginning and end indices of the block of nonuniform bricks, and I is the index to the secondary table. For instance, there is only one block of nonuniform bricks in the line $L - 1$ and it starts at index 5 and ends at index 17. Hence, the second entry of the primary table contains $L = 1$, $B = 5$ and $E = 17$. As uniform bricks at indices 0–4 and 18–23 are not stored, i.e., they are skipped, the total number of bricks skipped before line $L - 2$ is incremented by the value 11, which gives the value $S = 27$ for the third entry.

The storage requirements are 12 bytes per entry in the primary table and 4 bytes per entry in the secondary table. We have discovered empirically that the average list

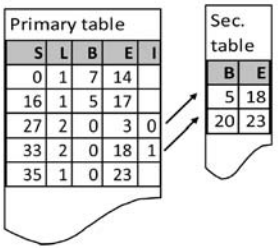

Fig. 8 An example of the data structure for the data from Fig. 7

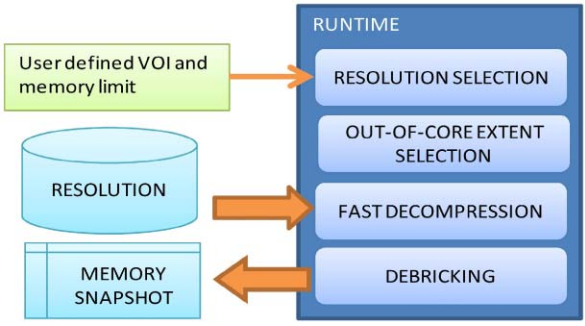

Fig. 9 The scheme of runtime operations

length is about 1.3, which means that the required memory is about $\frac{N^2}{256} \cdot (12 + 0.3 \cdot 4) \approx 0.05 \cdot N^2$. For the VH Male data set ($1760 \times 1024 \times 1878$), only about 100 KB are required (which is only 4% of size in comparison with the straightforward approach).

Due to this fast compression strategy, the total size of all 16 resolution files constructed for this dataset in the preprocessing is only 1.2 GB, which is about one third of the original size (3.15 GB).

3.2 Runtime

At runtime, instead of having to sample the original large data, the VME now simply selects the proper resolution from the subsampled versions computed during the preprocessing stage. It loads the requested ROI from the file, performing reconstruction of the skipped (not stored) bricks and linearization of the voxels on the fly. When the VTK visualization pipeline is to be constructed at runtime, in order to support the visualization of large volume datasets we need to use a custom data source object (with *vtkImageData* as output) instead of using the traditional *vtkImageReader*. This custom object handles the above processing required to support the visualization. An overview of the process is depicted in Fig. 9.

The details of the data retrieval algorithm are as follows. The user has to specify the maximum amount of memory to be made available for the data that is to be visualized. Increasing the size would allow a higher resolution for the data, but it would also mean longer retrieval and rendering times, which may reduce the smoothness of real-time manipulations such as rotation, translation, or zooming. In addition, as *vtkImageData* stores volume data in a large one-dimensional array, fragmentation of

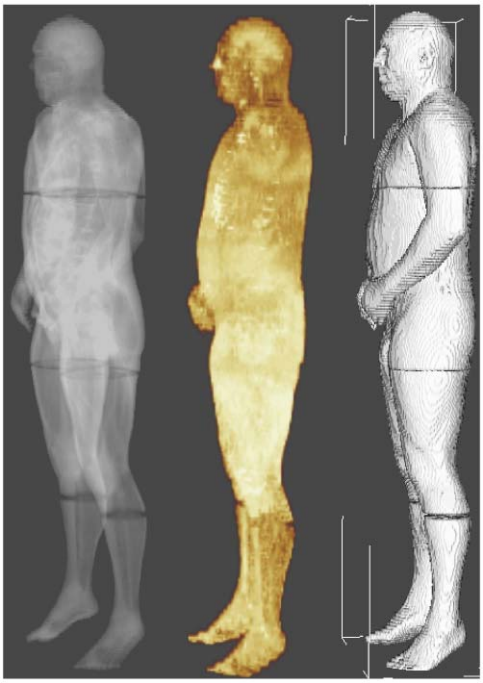

Fig. 10 Visualization of the whole VH Male dataset (3.15 GB) with the memory limit set to 8 MB using DRR, MIP and iso-surface volume rendering. The selected sample rate is 8

memory may mean that there is not a free memory block large enough to satisfy the demands. Therefore, in general, smaller limits (e.g., 16 MB) are recommended. We note that an automatic selection of this memory limit is impossible as the data retrieval algorithm is not aware of the rendering algorithm (due to the VTK pipeline design) and, therefore, it cannot estimate the time required for the rendering.

The user also specifies the area he/she wants to explore, i.e., the volume of interest (ROI). The algorithm computes the sample rate needed for sampling of the specified ROI in order to produce a snapshot that fits the given memory limit. The resolution file with this sample rate is selected. When the resolution is accessed for the first time, the brick map and both the primary and the secondary tables are loaded into main memory.

The bricks that cover the requested ROI are identified and these are processed line-by-line, slice-by-slice. Uniform bricks are filled using the constant value stored in the brick map; nonuniform bricks are loaded from file. The position of a nonuniform brick in the file is computed from the information stored in the primary and secondary indexing tables. It is quite clear that brick samples, which are stored in a linear order, must pass through a reverse process to bricking when they are copied into the volume memory snapshot. In order to speed up the retrieval, only bricks that have not already been loaded or filled in the previous ROI request are processed.

After the data is retrieved by the VME (it is stored in the underlying *vtkImageData* object), it is passed to all Views connected to this VME. In LhpBuilder, the user can visualize the data using various volume rendering techniques including DRR, MIP or iso-surface. Figure 10 shows the whole VH Male dataset (size 3.15 GB) with the

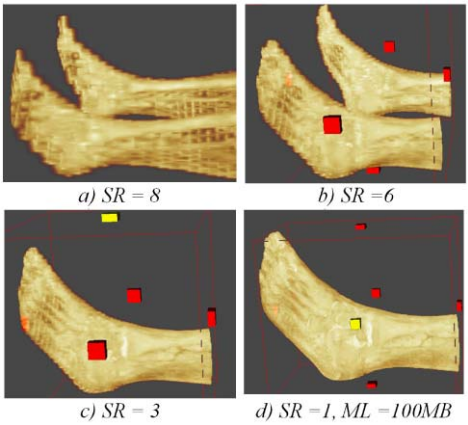

Fig. 11 Iterative exploration with the memory limit set to 8 MB; in (**d**), to display data at the highest resolution, the memory limit was set to 100 MB

memory limit set to 8 MB using three volume rendering techniques. Figure 11 shows an example of the iterative process of data exploration and extraction of the lower part of the body on the above dataset by specifying a smaller and smaller ROI (using a gizmo) at each step. It can be seen that the resolution successively increases. We note that the data retrieval process took only fractions of a second on standard hardware.

To ensure the robustness of the proposed algorithm for interactive visual exploration of very large datasets, we have tested it also on the VH Female data (12 GB). We also successfully tested it on an even larger dataset, created by replicating the VH Female data to generate a file of size approx. 24 GB.

4 Experiments and results

The approach described above was implemented in C++ (MS Visual Studio 2008) in the MAF visualization environment. For our experiments, an older Asus A2542D laptop (AMD Athlon 2.8 GHz, 512 MB RAM, HDD 60 GB with 4,200 rpm, Windows XP Pro) and more powerful Dell Precision 470 desktop computer (2x Intel Xeon 3.4 GHz, 2 GB DDR2 400 MHz RAM, 2x HDD 137 GB SCSI with 10,000 rpm, Windows XP Pro) were used. The VH volume datasets, Male (3.15 GB—1760 $\times$ 1024 $\times$ 1878) and Female (12.03 GB—2048 $\times$ 1216 $\times$ 5186), were used for the main tests on interactive visual exploration and extraction. Other datasets (e.g., [3, 25]) were also used to test the effect of noise present in the datasets on the proposed compression scheme during the optimized bricking process; see Fig. 12.

Figure 13 compares the times required to retrieve various ROIs of $400 \times 400 \times 100$ samples from the highest resolution file of VH Male data (a) when samples are stored in a linear fashion and (b) when they are stored in a bricked order ($BS = 16$). For retrievals in the bricked version, almost constant times (221–253 ms) were achieved, while in the linear version, inefficient use of disk cache led to retrieval times varying from 231 to 10,363 ms.

Table 2 shows the overall sizes of all resolution levels constructed for various datasets during the bricking process when the proposed compression scheme was

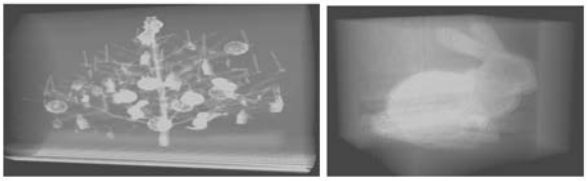

Fig. 12 X-Mas tree ($512 \times 512 \times 499$) and Bunny ($512 \times 512 \times 361$) data

Fig. 13 The influence of bricking on the total time needed for 12 data retrievals on an Asus A2542D

Table 2 Results of the proposed compression scheme and the preprocessing time for various datasets

Model	Dimensions@Bits	Orig. size [MB]	Bricked size [MB]	Compr. ratio [%]	Levels	Preproc. time [s]
Multiple_Female	$4096 \times 2432 \times 2568$@8	24396	6340	25.99	32	4079.63
VH_Female	$2048 \times 1216 \times 5186$@8	12317	3115	25.29	32	1886.61
VH_Female (part)	$2048 \times 1216 \times 3450$@8	8194	2696	32.90	32	1230.11
VH_Female (part)	$2048 \times 1216 \times 1725$@8	4097	1201	29.30	16	495.93
VH_Male	$1760 \times 1024 \times 1878$@8	3228	1205	37.32	16	393.40
VH_Female (part)	$2048 \times 1216 \times 863$@8	2050	341	16.65	16	213.64
Xmas_tree	$512 \times 512 \times 499$@16	250	261	104.56	8	27.61
Bunny	$512 \times 512 \times 361$@16	181	181	100.46	8	19.89

used. It can be observed that there is a significant gain in compression of bricked data for VH Male and Female datasets as these datasets contain large empty areas and they were also filtered in order to remove noise. However, there is no gain in compression for X-Mass and Bunny data sets because of the presence of a large amount of noise (see Fig. 12). Nevertheless, the compression ratio achieved still corresponds to the theoretical bound 1.21 (storage costs), as discussed in Sect. 3.1.

It is also evident from Table 2 that the time needed for the preprocessing (each dataset is to be preprocessed only once, first time) is almost linearly dependent on the input data size. A dataset of 24 GB is processed in 68 minutes, producing 32 resolution files. Note that the processing can be easily speeded up by constructing all levels simultaneously and/or by using multiple threads.

We also experimented with various brick sizes in order to find the optimal one. A comparison of data retrieval times achieved for several sizes of ROI (within one

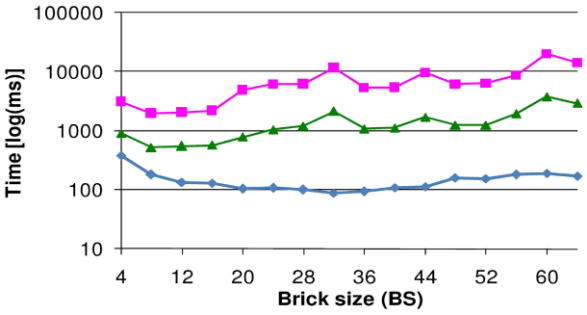

Fig. 14 Minimum, average and maximum retrieval times for various brick sizes on an Asus A2542D

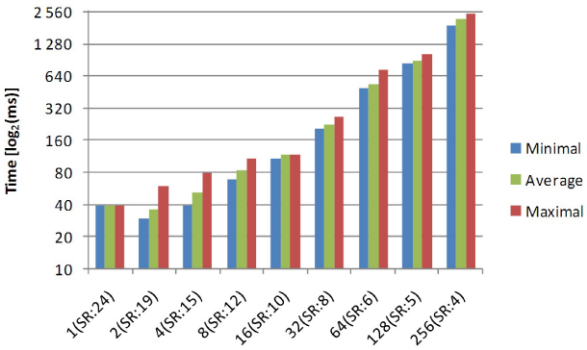

Fig. 15 Effect of memory limits on retrieval time

resolution only) for various brick sizes is given in Fig. 14. As can be seen, while for smaller regions (ROIs), the data retrievals are fastest for a brick size around 32, for larger regions, in an average case the highest performance is reached when the brick size is around 16. Whilst the differences in times achieved for brick sizes 16 and 32 are negligible for smaller regions (up to 50 ms), they are significant for larger regions (more than 1 s). Therefore, we decided to use the value 16 as a base for our bricking techniques. Further experiments (not presented here) showed that this brick size also leads to a higher compression ratio when compared with larger constants.

Figure 15 shows retrieval times for the whole VH Female data (12.03 GB) using various memory limits (which leads to the selection of different resolution levels). Times grow proportionally to the amount of data to be retrieved. In order to retrieve data interactively, the memory limit 16 MB or 32 MB is recommended (in those cases, the data is typically obtained in less than 0.25 s). Indeed, additional time is required to visualize the data, and this time can be several times larger (especially if some sophisticated rendering technique is exploited). With memory limit 256, the user needs to wait about 10 seconds to see an iso-surface (the fastest CPU-based method in LhpBuilder) of the requested whole body. A recommended strategy is to use a memory limit of 1 MB initially to quickly "zoom" into the region of interest and then to increase the memory limit and to refine the ROI.

Figure 16 shows average times needed to retrieve the requested ROI during successive exploration of the VH Female data. The memory limit was set to 16 MB. It can be seen that when a new resolution is selected, the algorithm needs a significantly

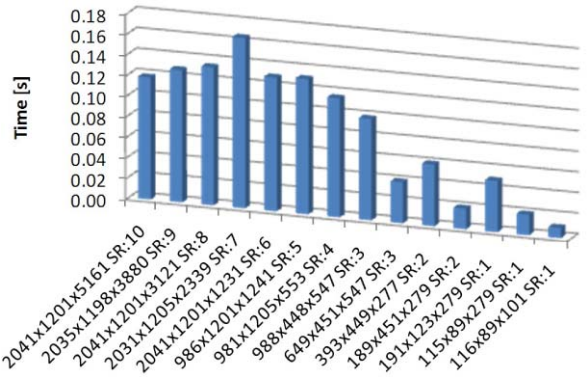

Fig. 16 Average times needed to retrieve the requested ROI during various explorations

Table 3 Comparison of minimal/maximal times (in ms) achieved for various ROIs

	SR	Brute	MR	MR+FC
#1	6	934 / 22,799	232 / 740	233 / 238
#2	6	901 / 918	115 / 170	109 / 112
#3	5	804 / 20,801	96 / 646	121 / 289
#4	5	533 / 537	74 / 91	89 / 93
#5	4	888 / 18,143	68 / 945	82 / 316
#6	3	1,066 / 16,279	104 / 1,311	125 / 708
#7	2	448 / 4,242	66 / 1,644	68 / 1,386

larger time than in the case when the same resolution level is used. This is because of hardware and software caches. Similar behavior can be seen for other memory limits.

Table 3 compares the minimum and maximum times needed to retrieve various ROIs (of differing sizes) by the brute-force approach (*Brute*), which accesses the original VH Male volume dataset and performs the sampling on the fly (this is, actually, the straightforward VTK streaming technique), with times needed by our multiresolution approach (*MR*), without and with the fast compression (*FC*) technique (i.e., in the former case, all the bricks are stored). This experiment was performed on the Dell Precision 470.

It is obvious that our approach, even on a computer with a very fast hard disk, significantly outperforms the straightforward solution. It is important also to point out that although, in the best case, the multiresolution approach with fast compression consumes slightly more time (due to an overhead introduced by the decompression) than the version without it, it is nevertheless significantly faster in the expected case (see Table 3).

5 Conclusions and future scope

In this paper, we proposed an out-of-core approach that combines various techniques such as multiresolution and volume bricking. The approach is suitable for an interactive exploration of large volumes in VTK-based visualization systems (such as

LhpBuilder). In contrast with existing approaches that also exploit the multiresolution strategy for data organization, our approach does not create a space-consuming resolution hierarchy (due to the fast compression) for typical medical data, yet, as can be seen from the results presented, it still allows almost real-time data retrievals. Moreover, the proposed solution has a simple implementation and, unlike many other approaches, it can be integrated into existing VTK visualization systems with ease. The work may be further extended to exploit the use of multi-threading or GPU to speed up the preprocessing. There also exists the possibility to further compress the preprocessed bricked layout data by improving on the skipping procedure for empty bricks.

Acknowledgements The authors would like to thank the various people that contributed to the realization of the MAF and LhpBuilder. Thanks are also due to the National Library of Medicine (NLM), USA, for providing the high-resolution Visible Human Male and Female datasets. The work was partially supported by the European Commission in the project (LHDL: Living Human Digital Library) (FP6-2004-IST-4-026932).

References

1. 3D-DOCTOR: vector-based 3d medical modeling and imaging software. http://www.3d-doctor.com
2. Ahrens J, Brislawn K, Martin K, Geveci B, Law CC, Papka M (2001) Large-scale data visualization using parallel data streaming. IEEE Comput Graph Appl 21(4):34–41
3. Bunny data. http://www9.informatik.uni-erlangen.de/external/vollib/
4. Castanie L, Mion C, Cavin X, Levy B (2006) Distributed shared memory for roaming large volumes. IEEE Trans Vis Comput Graph 12(5):1299–1306
5. Dong F, Krokos M, Clapworthy G (2000) Fast volume rendering and data classification using multiresolution in min-max octrees. Comput Graph Forum 19:359–368
6. Edwards HM (1974) Riemann's zeta function. Academic Press, New York
7. Engle K, Hadwiger M, Kniss JM, Salama C, Weiskopf D (2006) Real-time volume graphics. AK Peters Ltd, Wellesley
8. Frank M, Váša L, Skala V (2006) MVE-2 applied in education process. In: Proceedings of NET technologies 2006, pp 39–45
9. Guthe S, Wand M, Gonser J, Straer W (2002) Interactive rendering of large volume data sets. In: IEEE visualization '02, pp 53–59
10. LaMar EC, Hamann B, Joy KI (1999) Multiresolution techniques for interactive texture-based volume visualization. In: IEEE visualization '99, pp 355–362
11. LHDL. http://www.biomedtown.org/biomed_town/lhdl
12. Lipsa D, Rhodes P, Bergeron R, Sparr T (2007) Spatial prefetching for out-of-core visualization of multidimensional data. In: IS&T/SPIE 19th annual symposium: electronic imaging science & technology, San Jose, CA, USA
13. Mimics: the standard for 3d image processing and editing based on scanner data. http://www.materialise.com/materialise/view/en/92458-mimics.html
14. Molnar S, Cox M, Ellsworth D, Fuchs H (1994) A sorting classification of parallel rendering. IEEE Comp Graph Appl 14(4):23–32
15. Parker S, Parker M, Livnat Y, Sloan PP, Hansen C, Shirley P (1999) Interactive ray tracing for volume visualization. IEEE Trans Vis Comput Graph 5(3):238–250
16. Pieper S, Lorensen WE, Schroeder WJ, Kikinis R (2006) The NA-MIC kit: ITK, VTK, pipelines, grids and 3D slicer as an open platform for the medical image computing community. In: ISBI, pp 698–701
17. Preim B, Bartz D (2007) Visualization in medicine: theory, algorithms and applications. Morgan Kaufmann, San Mateo
18. Ramachandran P (2001) MayaVi: a free tool for CFD data visualization. In: 4th annual CFD symposium, Aeronautical Society of India
19. Rosset A, Spadola L, Ratib O (2004) Osirix: an open-source software for navigating in multidimensional dicom images. J Digit Imaging 17:205–216

20. Schroeder W, Martin K, Lorensen B (2004) The visualization toolkit, 3rd edn. Kitware Inc
21. Silva C, Chiang Y, El-Sana J, Lindstrom P (2002) Out-of-core algorithms for scientific visualization and computer graphics. In: Visualization'02, course notes, pp 1–36
22. Strengert M, Magallón M, Weiskopf D, Guthe S, Ertl T (2005) Large volume visualization of compressed time-dependent datasets on gpu clusters. Parallel Comput 31(2):205–219
23. VH. http://www.nlm.nih.gov/research/visible/visible_human.html
24. Viceconti M, Zannoni C, Testi D, Petrone M, Perticoni S, Quadrani P, Taddei F, Imboden S, Clapworthy G (2007) The multimod application framework: a rapid application development tool for computer aided medicine. Comput Methods Programs Biomed 85(2):138–151
25. Xmas tree data. http://www.cg.tuwien.ac.at/xmas/

Anupam Agrawal received the M.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology Madras, Chennai and the Ph.D. degree in Information Technology from the Indian Institute of Information Technology, Allahabad. He is currently a post-doctoral research fellow in the Department of Computing and Information Systems, University of Bedfordshire, UK. His research interests are computer graphics, geometric modeling, animation, large data visualization and image processing. He is a fellow of IETE, a senior member of IEEE, and a member of ACM.

Josef Kohout received his M.Sc. and Ph.D. degrees in Computer Science from the Department of Computer Science and Engineering, University of West Bohemia, Czech Republic in 2002 and 2005. From March 2005, he works at the same department as a research assistant. In 2008, he also worked as a post-doctoral research fellow at the Department of Computing and Information Systems, University of Bedfordshire, UK. In his research, he focused on the parallelization of the Delaunay triangulation in 2D and 3D, processing of large terrain and medical data sets and later (2007) also on the representation of digital images and video by Delaunay triangulation.

Gordon J. Clapworthy received a B.Sc. (Hons.) in Mathematics and a Ph.D. in Aeronautical Engineering from the University of London, and an M.Sc., with Distinction, in Computer Science from The City University, London. He is a Professor of Computer Graphics in the Department of Computing and Information Systems and Head of the Centre for Computer Graphics and Visualization (CCGV) at the University of Bedfordshire, UK. His interests are medical visualization, computer animation, biomechanics, virtual reality, surface modeling, and fundamental computer graphics algorithms. He is a member of the ACM, ACM SIGGRAPH and Eurographics.