EG UK Theory and Practice of Computer Graphics (2008)
Ik Soo Lim, Wen Tang (Editors)

Interactive Out-of-Core Exploration of Large Volume Datasets in VTK-Based Visualisation Systems

A. Agrawal[1], J. Kohout[1], G.J. Clapworthy[1], N.J.B. McFarlane[1], F. Dong[1], M. Viceconti[2], F. Taddei[2], D. Testi[3]

[1]Department of Computing & Information Systems, University of Bedfordshire, Luton, UK
[2]Laboratorio di Tecnologia Medica, Istituti Ortopedici Rizzoli, Bologna, Italy
[3]BioComputing Competence Centre, SCS, Casalecchio di Reno, Italy

Abstract

*The Visualisation ToolKit (VTK) has become a very popular tool for scientific data visualisation and it is used as a base in many existing visualisation systems. Scientific datasets produced nowadays by complex scientific simulations or by modern data acquisition techniques (e.g., airborne laser scanning) are often too large to be processed in one piece on commodity hardware, as simply storing it requires several giga-bytes. Although VTK provides a means for processing such datasets, their straightforward use is rarely efficient. This paper describes an efficient interactive exploration of large volume datasets under the Multimod Application Framework (MAF) [VZT*07], a VTK-based system; however, the proposed approach can be adopted for other systems with ease. It exploits various techniques such as multi-resolution layout and a volume bricking scheme to access data at an interactive rate. A user can explore the dataset by specifying a region of interest (ROI), which leads to the generation of a more accurate data representation inside the ROI. If even more precise accuracy is needed inside the ROI, nested ROIs are used. Experimental results show that the user can interactively explore large volume datasets such as the Visible Human male (1760x1024x1878, with a file size of 3.15 GB) on a commodity platform.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Graphics data structures and data types

1. Introduction

Sophisticated data-acquisition techniques and complex simulations are producing volume datasets of continually increasing size. These datasets often contain billions of voxels and, therefore, several gigabytes are required just to store them, which quickly surpasses the virtual address limit of current 32-bit PC platforms. At the highest resolution, they cannot be visualised in one piece on a single commodity computer. In the past, various solutions for visualisation of large volume datasets were proposed - a common feature is that they require an excellent cooperation of rendering algorithm and data retrieval algorithm.

Multiresolution techniques for rendering large volume data [LHJ99, DKC00, EHK*06] construct a hierarchical data structure (e.g., octree [LHJ99], multilevel pyramid [PPL*99]) in the preprocessing. The lowest level in this structure (i.e., leaves) keeps the original data, whilst higher levels (inner nodes) contain low-resolution approximations of the data. The lower levels are stored on hard disk and transferred to main memory on demand [SCESL02] at run-time. These methods typically display the volume in a region of interest at a high resolution and the volume away from that region at a progressively lower resolutions. This makes interactive visualisation 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 disk.

Parallel visualisation is also very often used to address the issues of large data processing [CMCL06, SMW*05, ABM*01, PPL*99]. In the overwhelming majority, it exploits 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., visualisation) of these parts, and the merging of outcomes to produce the final result. The existing approaches differ in how they subdivide and merge the data (which may be quite complex, especially for transparent objects [MCEF94]). 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 visualisation [GWGS02, SMW*05].

Although many of the existing approaches are able to visualise, or at least to explore, volume datasets of tens of GB in real-time on a cluster of workstations (e.g., [CMCL06]), their integration into already existing visualisation systems might be complex, or even impossible, for the following reasons. First, these approaches are usually problem specific, e.g., they are suitable for visualising medical volume datasets using only an iso-surface volume rendering technique [PPL*99], while visualisation systems typically aim to be able to visualise volume datasets of any kind using a variety of volume rendering techniques [EHK*06, PB07], and most systems also provide the user with volume smoothing and segmentation operations, colour mapping functions, etc. Further, these systems were very often designed to run only on a single computer.

Examples of these systems are as follows. MayaVi Data Visualizer [Ram01] and MVE-2 [FVS06] are general-purpose scientific-visualisation environments organised around the data-flow paradigm. 3D Doctor [DD], and Mimics [Mim] are 3D visualisation applications that excel in image processing activities such as segmentation. 3D Slicer [PLSK06] is an application mostly focused on neurosurgery problems. Other systems, such as OsiriX [Ros04] and the Multimod Application Framework (MAF) [VZT*07] are specialised 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 visualisation systems (especially, for those based on VTK [SML04]). It exploits multiresolution representations of the data (we use a low-resolution version for representing the entire dataset, and a higher-resolution version only for visualising smaller regions of interest) combined with a fast compression by skipping "empty" voxels, which usually occupy a large proportion of the space in medical datasets, such as National Library of Medicine's Visible Human dataset [VH]. We also perform a reorganisation of voxels in the preprocessing stage to optimise disk access at runtime. The approach allows the user to roam through multi-gigabyte volume datasets in almost real time, with low memory requirements.

The paper is organised as follows. The background of the research is given in Section 2, and the proposed hierarchical bricked partition-based out-of-core strategy is explained in Section 3. Results achieved with the proposed technique are presented in Section 4. Finally, in Section 5, concluding remarks and future work are described.

2. Background

The Visualisation ToolKit (VTK) [SML04] is an open source, freely available software system for 3D computer graphics, image processing, and visualisation 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 visualisation algorithms including scalar, vector, tensor, texture, and volumetric methods; and advanced modelling techniques such as implicit modelling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation.

In a VTK application, the process of visualisation can be described in terms of data flow through the so-called visualisation pipeline. The visualisation pipeline consists of process objects that operate on input data and transform it into output data. These 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 typical VTK pipeline is depicted in Figure 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 a streaming support to process larger datasets - see Figure 1b. Using the streaming technique, a dataset is processed piece by piece: a subset (piece) is loaded into memory, processed through classic VTK visual pipeline and the obtained results are decimated (e.g., subsampled) and successively combined together so they can be visualised. If we consider roaming through a large volume dataset, this solution suffers, however, from inefficiency as it needs to process the entire dataset whenever the region of interest (ROI) changes.

Despite its limitations, VTK is a base for many visualisation systems, e.g., MayaVi [Ram01], 3D Slicer [PLSK06] and OsiriX [Ros04], while in others, it can be used optionally, e.g., MVE-2 [FVS06].

The Multimod Application Framework (MAF) [VZT*07] is another visualisation system based on VTK (and other specialised libraries). This framework is designed to support the rapid development of biomedical software. It allows the development of a multimodal visualisation application in which different views of the same data are synchronised,

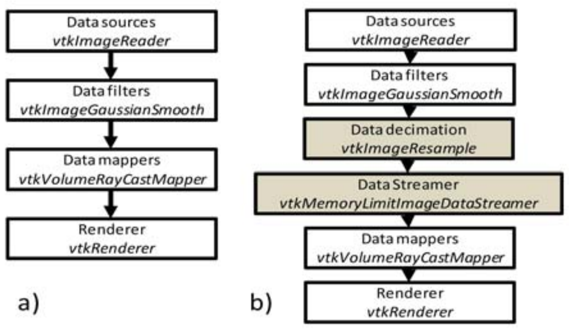


Figure 1: *VTK visualisation pipeline without (a) and with (b) streaming support. An example for visualising volume data is given in italics.*

and 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: *VirtualMedicalEntities* (VME), which are the data objects; *Views*, which provide interactive visualisation of the VMEs; *Operations*, which create new VMEs or modify existing ones; and the *InterfaceElements*, GUI components that define the user interface of the application. Special operations are importers, which import and convert into VME almost any biomedical dataset (e.g., RAW, DICOM, STL, etc.), 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 visualisation pipeline in shown in Figure 2.

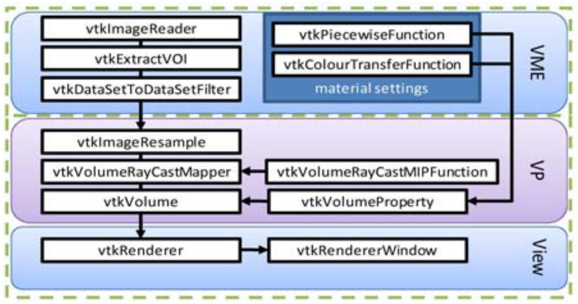


Figure 2: *An example of the VTK pipeline and its encapsulation in MAF. Note that from the user's point of view, VP (which is the visual pipe within MAF) is a part of View.*

MAF is exploited in LhpBuilder, which is software being developed within The Living Human Digital Library [LHD] project (LHDL). This project, which began in February 2006 and will last for three years, aims to create the technical infrastructure for another ongoing project: The Living Human Project (LHP). 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 is to add a new capability to LhpBuilder for importing, accessing, visualising and extracting part of a large volume dataset by interactive visual exploration. For the visualisation of limited-size volume data, the existing volume visualisation functions of LhpBuilder have been used, which are based on DRR (Digital Reconstructed Radiograph), MIP (Maximum Intensity Projection) and iso-surface algorithms [PB07, KSC*04].

3. Our Approach

As mentioned above, an efficient algorithm for exploring large volume data sets requires an excellent cooperation of the rendering algorithm and the data retrieval algorithm. Within the VTK scheme, however, rendering and data retrieval are two somewhat independent processes, which introduces some limitations.

This is even more serious in MAF applications, e.g., in LhpBuilder, (though similar problem may be expected in other systems), where the *VME*, which includes the data retrieval algorithm, is not aware of the *View* that will be used to visualise it, and the *View* (which contains the rendering algorithm) does not know which *VME* it will visualise. Under these circumstances, all we can do when visualising 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 visualise 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 the construction of a discrete multi-resolution model 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 optimised form.

Starting with a sample rate (*SR*) of 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 (we use 1 MB).

If we consider cubic volumes with N voxels in one dimension, the number of levels k to be constructed can be computed as: $k^3 \geq \frac{N^3}{1MB} \Rightarrow k \approx \frac{N}{100}$. 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.

The Riemann zeta function $\zeta(s)$ of a complex variable s 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 now 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.

Figure 3 shows the dependency of the number of levels to the volume data size. We note that for arbitrary volume datasets, the number of levels may vary slightly from these.

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

Figure 3: *Relationship between the number of resolution levels and the volume data size.*

The processing of the input dataset has three main steps: sampling, bricking and fast compression - see Figure 4. As the input dataset is too large to be fully loaded into memory, it must be processed in 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 reorganised 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.

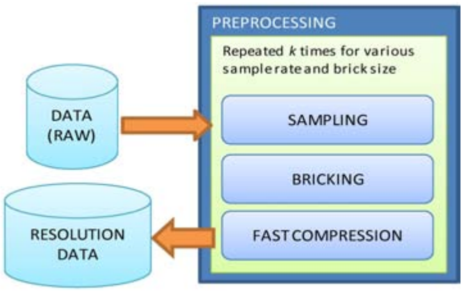
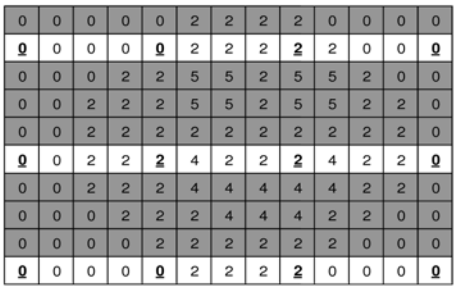


Figure 4: *The preprocessing scheme.*

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, 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., $\lfloor 14 \rfloor_4 = 12$. Actually, this means that the *BS* constant takes one of values: 4, 8, 12 and 16. The meaning of this constant will be explained later.

The input file is then read line-by-line, slice-by-slice, 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 memory buffer capable of holding *BS* 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 that 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 Figure 5 (in which line 0 is at the bottom). If *BS* is 2, the buffer must be capable of storing 24 samples.



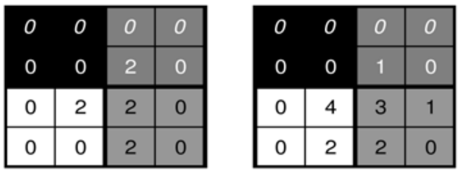
0	0	0	0	0	2	2	2	2	0	0	0	0
$\underline{0}$	0	0	0	$\underline{0}$	2	2	2	$\underline{2}$	2	0	0	$\underline{0}$
0	0	0	2	2	5	5	2	5	5	2	0	0
0	0	2	2	2	5	5	2	5	5	2	2	0
0	0	2	2	2	2	2	2	2	2	2	2	0
$\underline{0}$	0	2	2	$\underline{2}$	4	2	2	$\underline{2}$	4	2	2	$\underline{0}$
0	0	2	2	2	4	4	4	4	4	2	2	0
0	0	0	2	2	2	4	4	4	2	2	0	0
0	0	0	0	2	2	2	2	2	2	0	0	0
$\underline{0}$	0	0	0	$\underline{0}$	2	2	2	$\underline{2}$	0	0	0	$\underline{0}$

Figure 5: *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.*

We note that, as the process is repeated for every *SR*, the data is read several times. While this may negatively influence the overall time needed for the preprocessing stage, it does makes the algorithm much simpler, and the preprocessing stage only has to be performed once on 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 $\zeta(2) = \frac{\pi^2}{6} \approx 1.644934$, 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 multi-dimensional scientific data such as volume data, which represents data in a 3D spatial domain [LRBS07].

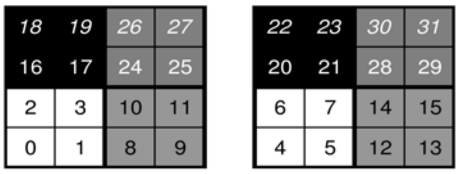
To circumvent this problem, samples are grouped into bricks of $BS \times BS \times BS$ samples. If there are not enough samples to form a brick, the missing samples are considered to be zero - see Figure 6. After that, bricks can be stored, one brick after another, into the output file. The order in which the samples are stored is illustrated in Figure 7. 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.



0	*0*	*0*	*0*
0	0	2	0
0	2	2	0
0	0	2	0

0	*0*	*0*	*0*
0	0	1	0
0	4	3	1
0	2	2	0

Figure 6: *Samples of two slices organised into 2x2x2 bricks (the first corresponds to Figure 5). The missing samples are given in italics. Bricks are denoted by grey scale.*



18	*19*	*26*	*27*
16	17	24	25
2	3	10	11
0	1	8	9

22	*23*	*30*	*31*
20	21	28	29
6	7	14	15
4	5	12	13

Figure 7: *The storing order of samples.*

It may happen that some bricks contain samples all with the same value, i.e., these bricks are uniform (like the top left brick in Figure 6). This case is quite common, especially for medical data which frequently contains a lot of empty space - see Figure 8. 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.

However, this also introduces a problem: while, previously, the position of brick samples in the file was given implicitly by the brick index, now, we have to define the position explicitly and we need not only to store these positions in the output file but also to keep them in memory during runtime.

A straightforward approach is to store, for every brick, a single bit denoting whether the brick is a uniform one 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 $SR = 1$ (although the brick size BS is in that case, save for smaller volumes, the largest one, i.e.,16).

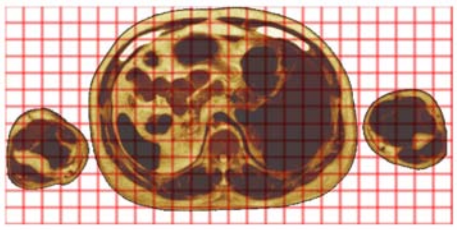
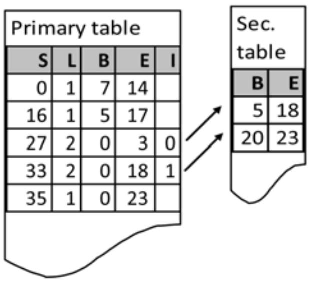


Figure 8: *An example of medical data with highlighted brick boundaries.*

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 volume data $1760 \times 1024 \times 1878$ that is bricked into $110 \times 64 \times 118$ bricks, 2.38 MB (24 bits per brick) are required.

Considering medical data (see Figure 8), this seems to be unnecessarily space consuming, as there is typically only one block of uniform bricks in any line of bricks. We therefore propose another 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 non-uniform 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.



Primary table

S	L	B	E	I
0	1	7	14	
16	1	5	17	
27	2	0	3	0
33	2	0	18	1
35	1	0	23	

Sec. table

B	E
5	18
20	23

Figure 9: *An example of the data structure for the data from Figure 8.*

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 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 Visible Human data set of $1760 \times 1024 \times 1878$ voxels, only about 100 KB are required (which is only 4% of size in comparison with the straightforward approach).

We note that 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 in comparison to the original size (3.15 GB).

3.2. Runtime

When the VTK visualisation pipeline is to be constructed at runtime, all we have to do to support the visualisation of large volume datasets is to use a custom data source object (with *vtkImageData* as output) instead of using traditional *vtkImageReader*. This custom object handles the selection of the proper resolution file (from those computed during the preprocessing stage) and the loading of the requested region of interest (ROI) from the file, which includes the reconstruction of the skipped (not stored) bricks and linearisation of the voxels on the fly.

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 visualised. 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 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.

The user also specifies the area to explore, i.e., the region of interest (ROI). The algorithm computes the sample rate for the specified ROI that will produce a snapshot that fits within the given memory limit. The resolution file with this sample rate is selected. 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 uniform value stored in the file; non-uniform bricks are loaded from file. The position of a 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.

After the data is retrieved by the *VME*, it is passed to all *Views* connected to this *VME*. In LhpBuilder, the user can visualise the data using various volume rendering techniques including DRR, MIP or iso-surface - see Figure 10. Figure 11 shows an example of the iterative process of data exploration by specifying a smaller and smaller ROI 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.

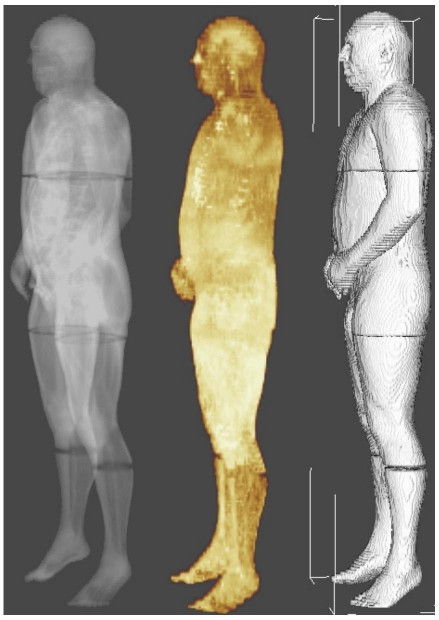


Figure 10: *Visualisation of the whole Visual Human male dataset (3.15 GB) with the memory limit set to 8 MB using DRR, MIP and iso-surface volume rendering. The selected sample rate is 8.*

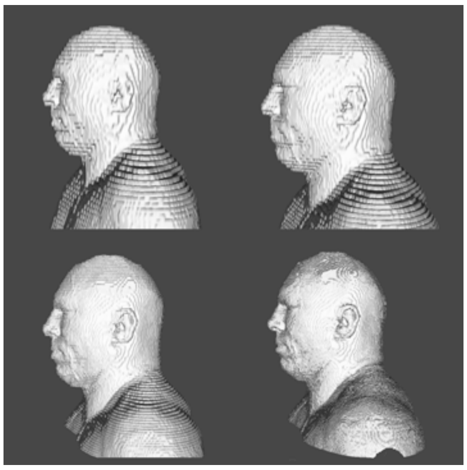


Figure 11: *Iterative exploration of Visual Human male dataset. The memory limits / selected sample rates are (from the left to right and from top to bottom): 8MB / 8, 8MB / 6, 8MB / 4, 64MB / 2.*

4. Experiments and Results

The approach described above was implemented in C++ (MS Visual Studio 2008) in the MAF visualisation environment. For our experiments, an older laptop Asus A2542D (AMD Athlon 2.8 GHz, 512 MB RAM, HDD 60GB with

4200rpm, Windows XP Pro) and the more powerful Dell Precision 470 (2x Intel Xeon 3.4 GHz, 2 GB DDR2 400 MHz RAM, 2x HDD 137 GB SCSI with 10 000rpm, Windows XP Pro) were used. The Visible Human volume data ($1760 \times 1024 \times 1878$ - 3.15 GB) was used for the main tests.

Figure 12 compares the times required to retrieve various ROIs of $400 \times 400 \times 100$ samples from the highest resolution file (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, whilst in the linear version, inefficient use of disk cache led to retrieval times varying from 231 to 10,363 ms.

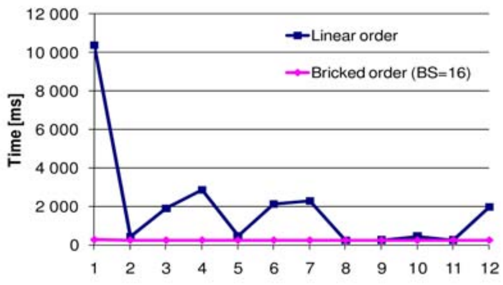


Figure 12: *The influence of bricking on the total time needed for 12 data retrievals on an Asus A2542D.*

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 resolution only) for various brick sizes is given in Figure 13. As it can be seen, whilst for smaller regions (ROIs), the data retrievals are fastest for the brick size around 32, for larger regions, the highest performance is reached when the brick size is around 16. However, 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 value also leads to a higher compression ratio when compared with larger constants.

Figure 14 compares the minimum and maximum times needed to retrieve various ROIs (of differing sizes) by the brute-force approach (Brute) that accesses the original 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 significantly outperforms the straightforward solution. It is important to point out

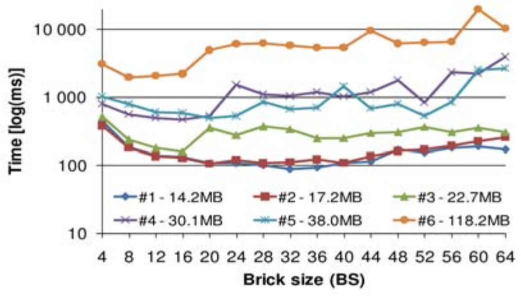


Figure 13: *Comparison of retrieval times achieved for various brick sizes on an Asus A2542D.*

	SR	*Brute*	*MR*	*MR+FC*
#1	6	934	232	233
#2	6	901	115	109
#3	5	804	96	121
#4	5	533	74	89
#5	4	888	68	82
#6	3	1,066	104	125
#7	2	448	66	68

	SR	*Brute*	*MR*	*MR+FC*
#1	6	22,799	740	238
#2	6	918	170	112
#3	5	20,801	646	289
#4	5	537	91	93
#5	4	18,143	945	316
#6	3	16,279	1,311	708
#7	2	4,242	1,644	1,386

Figure 14: *Comparison of minimum and maximum times (in ms) achieved for various ROIs.*

that although, in the very best cases, the multiresolution approach with fast compression consumes slightly more time than the version without it (due to an overhead introduced by the decompression), it is significantly faster in normal cases.

5. Conclusions and Future Scope

In this paper, we proposed an out-of-core approach that combines various techniques such as multi-resolution and volume bricking. The approach is suitable for an interactive exploration of large volumes in VTK-based visualisation systems (such as LhpBuilder). In contrast with existing approaches that also exploit the multiresolution strategy for data organisation, 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, can be integrated into existing VTK visualisation systems, with ease.

As part of future work, we shall exploit the use of multi-threading or GPU to speed up the preprocessing step. There also exists the possibility to further compress the preprocessed bricked layout data by improving on the skipping procedure for empty bricks.

Acknowledgements

This work was supported by the Information Society Technologies Programme of the European Commission under the project LHDL (IST-2004-026932). The authors would like to thank the various people who contributed to the realisation of the MAF and LhpBuilder.

References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications 21*, 4 (July/Aug. 2001), 34–41.

[CMCL06] CASTANIE L., MION C., CAVIN X., LEVY B.: Distributed shared memory for roaming large volumes. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1299–1306.

[DD] 3D-DOCTOR: Vector-based 3d medical modeling and imaging software; http://www.3d-doctor.com.

[DKC00] DONG F., KROKOS M., CLAPWORTHY G.: Fast volume rendering and data classification using multiresolution in min-max octrees. *Computer Graphics Forum 19* (2000), 359–368.

[EHK*06] ENGLE K., HADWIGER M., KNISS J. M., SALAMA C., WEISKOPF D.: *Real-Time Volume Graphics*. AK Peters Ltd., 2006.

[FVS06] FRANK M., VÁŠA L., SKALA V.: MVE-2 applied in education process. In *Proceedings of .NET Technologies 2006* (2006).

[GWGS02] GUTHE S., WAND M., GONSER J., STRAER W.: Interactive rendering of large volume data sets. In *IEEE Visualization '02* (2002), pp. 53–59.

[KSC*04] KROKOS M., SAVENKO A., CLAPWORTHY G. J., LIN H., MAYORAL R., VICECONTI M., JAN S. V. S.: Real-time visualisation within the multimod application framework. In *IV '04: Proceedings of the Information Visualisation* (2004), pp. 21–26.

[LHD] LHDL: http://www.biomedtown.org/biomed_town/lhdl.

[LHJ99] LAMAR E. C., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *IEEE Visualization '99* (1999), pp. 355–362.

[LRBS07] LIPSA D., RHODES P., BERGERON R., SPARR T.: Spatial prefetching for out-of-core visualization of multidimensional data. In *IS&T/SPIE 19th Annual Symposium: Electronic Imaging Science & Technology* (2007), p. 64950G.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[Mim] MIMICS: The standard for 3d image processing and editing based on scanner data http://www.materialise.com/materialise/view/en/92458-mimics.html.

[PB07] PREIM B., BARTZ D.: *Visualization in Medicine: Theory, Algorithms and Applications*. Morgan Kaufmann, 2007.

[PLSK06] PIEPER S., LORENSEN W. E., SCHROEDER W. J., KIKINIS R.: The NA-MIC kit: ITK, VTK, pipelines, grids and 3D slicer as an open platform for the medical image computing community. In *ISBI* (2006), pp. 698–701.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999), 238–250.

[Ram01] RAMACHANDRAN P.: MayaVi: A free tool for CFD data visualization. In *4th Annual CFD Symposium, Aeronautical Society of India* (2001).

[Ros04] ROSSET A.: Osirix: An open-source software for navigating in multidimensional dicom images. *Journal of Digital Imaging 17* (2004), 205–216.

[SCESL02] SILVA C., CHIANG Y., EL-SANA J., LINDSTROM P.: Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02* (2002). Course Notes.

[SML04] SCHROEDER W., MARTIN K., LORENSEN B.: *The Visualization Toolkit, Third Edition*. Kitware Inc., 2004.

[SMW*05] STRENGERT M., MAGALLÓN M., WEISKOPF D., GUTHE S., ERTL T.: Large volume visualization of compressed time-dependent datasets on gpu clusters. *Parallel Computing 31*, 2 (2005), 205–219.

[VH] VH: http://www.nlm.nih.gov/research/visible/visible_human.html.

[VZT*07] VICECONTI M., ZANNONI C., TESTI D., PETRONE M., PERTICONI S., QUADRANI P., TADDEI F., IMBODEN S., CLAPWORTHY G.: The multimod application framework: A rapid application development tool for computer aided medicine. *Computer Methods and Programs in Biomedicine 85*, 2 (2007), 138–151.