An improved quadtree-based algorithm for lossless compression of volumetric datasets

GREGOR KLAJNŠEK, BOJAN RUPNIK, DENIS ŠPELIČ
Laboratory for Geometric Modeling and Multimedia Algorithms,
Institute of Computer Science
Faculty of Electrical Engineering and Computer Science, University of Maribor
Smetanova ul. 17, SI-2000 Maribor
SLOVENIA
gregor.klajnsek@uni-mb.si http://gemma.uni-mb.si

Abstract: In this paper a novel algorithm for lossless compression of volumetric data is presented. This algorithm is based on our previously presented algorithm for lossless compression of volumetric data, which uses quadtree encoding of slices of data for discovering the coherence and similarities between consecutive slices. By exploiting these properties of the data, the algorithm can efficiently compress volumetric datasets. In this paper we upgrade the basic algorithm by introducing several new routines for determination of coherence and similarities between slices, as well as some new entropy encoding techniques. With this approach, we managed to additionally improve the compression ratio of the algorithm. Presented algorithm has two significant properties. Firstly, it is designed for lossless compression of volumetric data, which is not the case with most of existing algorithms for compression of voxel data, but this is a very important feature in some fields, i.e. medicine. Secondly, the algorithm supports progressive reconstruction of volumetric data and is therefore appropriate for visualization of compressed volumetric datasets over the internet.

Key-Words: Volumetric Data, Voxels, Compression, Lossless Compression, Quadtree Encoding

1 Introduction

Voxel data have become an asserted mean when representation of interiors of geometric objects is required. Their elemental parts are called voxels. Voxels are smallest discrete parts that can store information about the substance that covers that part of space [1]. Medical applications represent a field where representation of objects in form of voxel data is most common. In medicine voxel datasets are usually obtained with computerized tomography (CT) or magnetic resonance imaging (MRI) [2, 3]. The output of the CT and MRI scanners is usually a set of two dimensional raster images called slices. If those slices are arranged in space one after another along the depth axis, a complete description of the observed object is obtained. Besides medicine voxel representation of data is also often used in computational fluid dynamics (CFD), geology and even in terrain visualization (i.e. representation of water). In the past, majority of research in the field of voxel graphics was related to visualization. Recently, due to significant progress in graphics hardware, real-time or near real-time volume visualization has become possible [4, 5, 6].

However, two considerable problems still exist: the storing of voxel data and the transfer of voxel data through the Internet. This is due to the extreme size of voxel data. For example, a typical medical voxel dataset can occupy up to few hundred megabytes of data. Compression algorithms are generally used for reducing the size of the data. General-purpose algorithms, such as ZIP occur as the first solution. However, general purpose algorithms cannot achieve maximal efficiency as they do not exploit special properties, like homogeneity and coherence, of voxel data. A much more efficient compression can be achieved with the special-purpose algorithms for the compression of voxel data.

We have already performed some significant research in the field of compression of volumetric data. Our initial algorithm for compression of volumetric data, which uses the determination of coherence and similarities of consecutive slices for efficient compression, has already been presented in a paper authored by Klajnšek and Žalik in 2005 [7]. In this paper we introduce new techniques for determination of similarities between consecutive slices and entropy encoding, which significantly improve the compression ratio of the basic algorithm.

The paper consists of four sections. After a short introduction, we present the previous work that has been carried out in the field of compression of voxel data. In the third section we present our algorithm for compression of volumetric data. In subsection 3.1 a basic idea behind the algorithm is presented. Subsection 3.2 introduces our novel techniques that improve the efficiency of the basic algorithm and therefore represents

the core of the paper. In subsection 3.3 we present practical results of the usage of algorithm. In the last section the work presented in the paper is evaluated.

2 Previous work

Several algorithms for compression of volumetric data have already been presented, however neither one of them has become popular enough to become a standard. Most of the presented algorithms are oriented towards lossy compression. The reason for this is that a while ago it was not possible to load a complete voxel dataset into memory from where it could be visualized. In the rest of the section we present some of the more important algorithms for compression of volumetric data. Besides those algorithms several additional algorithms exist, but majority of them are based on the presented ideas and only introduce some upgrades as for example new approach to entropy encoding.

The first voxel compression method was introduced in 1992 by Hesselink and Ning, and was called vector quantization [8, 9]. The basic idea of the algorithm is that instead of storing the actual value of the voxel, the codeword is stored. Usually, the bit-length of the codeword is shorter than the bit-length used for storing actual value and thus the compression is achieved. Obviously, this compression method performs lossy compression.

Second algorithm for compression of volumetric datasets was presented by Muraki in 1993 [10]. The idea behind this method is that the voxel space is at first divided into subspaces. These subspaces are then encoded using wavelet transformations. Method introduced by Muraki is very important as a lot of existing advanced compression methods today are based on this approach. While it is possible to perform lossless compression using wavelet encoding [11, 12] majority of existing wavelet based compression methods, including Muraki's, support only lossy compression.

In 1994 Fowler and Yagel introduced the first algorithm for lossless compression of voxel data [13]. Presented algorithm uses a combination of differential pulse code modulation (DPCM) and Huffman coding. DPCM technique belongs into a set of compression methods called predictive techniques. The value of the next voxel in the dataset is always predicted from the previous samples and the difference between the predicted and actual value is stored instead of actual value.

Yeo and Liu introduced a method for compression of voxel data based on discrete cosine transform (DCT) in 1995 [14]. This algorithm at first divides the voxel space into blocks consisting of 8×8×8 voxels. Each of blocks is then encoded with the DCT. In the third step the algorithm performs the scalar quantization on the obtained DCT coefficients. In the last step entropy encoding is performed using a combination of run-length encoding and Huffman coding. Because the method uses scalar quantization in the process it performs lossy compression.

Chiueh et al. presented an algorithm for compression of voxel data, which uses discrete Fourier transform (DFT) in 1997 [15]. Idea behind the algorithm is very similar to the previously described algorithm presented by Yeo and Liu. At first the DFT is performed over blocks of voxel data to obtain Fourier coefficients. Then the obtained coefficients are quantized and the entropy encoding is performed. Because of quantization this algorithm also performs lossy compression.

Zhu et al. presented another wavelet based algorithm for compression of voxel data in 1997 [16]. Their approach applies the wavelet transformation twice. At first the wavelet transformation is performed over the whole voxel space. After initial transformation, the algorithm determines interesting structures and homogeneous areas in the data and encodes these areas with an octree. The second wavelet transformation is then applied on the non-empty and non-homogeneous blocks of data. This method also performs lossy compression.

In 1999 Rodler introduced an algorithm for compression of volumetric data that was significantly different from all previously presented algorithms [17]. While other algorithms consider the whole voxel space as a whole entity, Rodler's algorithm considers the voxel space a set of arranged slices. By using this approach the compression of volumetric datasets becomes very similar to the compression of digital video and the introduced algorithm uses many ideas from this field. In the first step the 'temporal' prediction is used for determining the correlation between consecutive slices. In the second step 2D wavelet transformation is performed over the 'predicted' slices. Then the obtained coefficients which lie under the set threshold are removed and at the end the remaining coefficients are quantized. Due to usage of quantization and thresholding this approach performs lossy compression.

3 Problem Solution

As has been already stated, despite the existence of many algorithms for compression of voxel data none of them has yet become an official or even 'unofficial' standard. It is also hard to even pick one algorithm as the best, as every one of them has some advantages as well as disadvantages. For instance, most of the existing algorithms are strictly oriented towards lossy compression, which in unacceptable in some fields, especially medicine. Therefore it is our idea to develop an algorithm that would try to use as many advantages of

the other algorithms, while overcoming most of the disadvantages. Although the algorithm presented in this paper does not reach the desired goal yet, it represents, in our opinion, a big step in the right direction.

The proposed algorithm is based on encoding of the slices of voxel data with quadtrees and uses comparison of the quadtrees related to neighboring slices for determination of data coherence. Each slice is split into small patches called macro-blocks as first. This approach is very similar to the techniques used in algorithms for the compression of digital video. However, algorithms for the compression of digital video usually use macro-blocks of constant size, while in our algorithm the size of the macro-block is variable. Macro-blocks that are connected to the nodes of the lowest level of the related quadtree are called basic macro-blocks. The size of the basic macro-blocks is usually 4×4 or 8×8 voxels. In the following subsection we present the basic idea behind the algorithm

3.1 Basic algorithm

For clearance we will at first give description of the basic quadtree based compression algorithm that was presented in [7]. The algorithm for the lossless compression of voxel data using quadtrees consists of four steps:

- creation of the division tree,
- creation of quadtrees for all slices,
- calculation of difference trees for neighboring slice and
- entropy encoding.

A detailed description of each step is given in the continuation of the paper.

3.1.1 Creation of the division tree

Creation of the division tree represents the first step of the compression algorithm. Division tree is a fully-grown quadtree. Nodes of the division tree carry information about the macro-blocks of the mask slice. Mask slice is an empty slice that is split into macro-blocks, which are arranged accordingly to the following rule (called a row order rule):

At level i the number of macro-blocks is 4i-1 and the number of macro-blocks in one row is 2i-1. If the index of the starting macro-block of the row is n, then the following macro-blocks of the same row are indexed n+1, n+2, …, n+2i-1.

Each node of the division tree carries the information of starting coordinates and the size of the macro block which it represents. Division tree is useful because all the slices are represented by the same macro-block structure. By using the division tree we have to store the information about the position and the size of the macro blocks only once.

3.1.2 Creation of slice quadtrees

In the second step of the algorithm each slice of voxel data is encoded with a quadtree. These quadtrees are called slice quadtrees. Each slice is divided into small patches, called basic macro-blocks, at first. For each basic macro-block the average value of the voxels in the macro-block is calculated.

For each basic macro-block we must also determine whether is it homogeneous (all voxels in the macro-block are the same) or non-homogeneous (voxels are not the same). This information is stored into the appropriate node of the slice quadtree. The nodes of slice quadtrees carry the following information:

- *index* – defines mapping between the macro-block and the adequate node in the division tree,
- *node type* – depends on the position of the node in the quadtree and the values of voxels in the related macro-block,
- *average value* – average value of the voxels in the macro-block,
- *pointer to voxel array* – a pointer to an external array of voxel values. When we come across a non-homogeneous basic macro-block the values of the voxels have to be stored in an external array.

Encoding of basic macro-blocks produces the nodes of the lowest level of the slice quadtree. The nodes of higher levels are obtained with pruning. The pruning process follows this simple rule:

If all sons of the current node represent homogeneous macro-blocks with the same average value, then they are joined into one larger homogenous macro-block.

When such a case occurs, the average value of sons is forwarded into the father node and all four sons are removed from the quadtree. If the pruning can not be performed, the average value of the father node is calculated from the average values of all four sons. In this case the father node is an inner node (branch) in the quadtree. Regarding the position of the node and the properties of the related macro-block we can now define three different node types:

- *Type 1* – the node is an inner node – branch of the quadtree. Such node represents a non-homogeneous macro-block, which consist of smaller macro-blocks.
- *Type 2* – the node represent homogeneous macro-block. The nodes of this type represent the leaves of the quadtree. However, this leaves are not always on the lowest level of the quadtree. Since we used

pruning in the process the slice quadtrees are not fully-grown quadtrees.

- *Type 3* – the node represents a non-homogeneous basic macro-block. Such node can only appear on the lowest level of the quadtree. The voxels of the related basic macro-block must be stored into an external array. The address of the array is stored into the pointer to voxel array.

The pruning process is repeated until the nodes of the highest level are created. In this way the slice quadtree for one slice is created. This process must be repeated for all slices. As a final result a set of slice quadtrees is obtained.

3.1.3 Calculation of slice quadtrees

When the slice quadtrees of all slices are created we can start determining the similarities between neighboring slices. For this we use special trees, called difference trees. A difference tree is a fully-grown quadtree used for determining equal nodes in two quadtrees. Nodes of the difference tree have just one parameter:

- *IsEqual* – which marks whether two adequate nodes that belong to two slice quadtrees of neighboring slices are equal or not.

Four tests have to be performed for comparison of two nodes:

- comparison of node types,
- comparison of average values,
- comparison of sons, if both compared nodes are inner nodes of the quadtree,
- comparison of voxels, if both compared nodes represent non-homogeneous basic macro-blocks.

3.1.4 Entropy encoding

In the last step the remaining information are stored. For more efficient compression we use special entropy encoding algorithms. The data are stored into two files:

- *quadtree structures file (QSF)* – stores the information about the quadtrees,
- *voxel data file (VXF)* – stores the values of voxels of non-homogeneous basic macro-blocks. We previously stored this voxels into external arrays using pointers to voxel arrays.

We will not provide the actual entropy encoding algorithm used for encoding the QSF here, because it is not necessary for understanding the basic idea behind the algorithm. However, the detailed description can be found in [7]. The entropy encoding algorithm which is used for storing values of voxels into the voxel data file is based on the idea that instead of storing the actual values of voxels, the difference to the average value of the macro-block is stored. Of course, only the minimal number of bits required for storing the biggest difference is used for representation of all difference values of the macro block.

3.2 Advanced algorithm

Advanced algorithm introduces several new techniques for determination of coherence and similarities and entropy encoding. These techniques are used for storing the values in the voxel data file. We introduce eight different types of coding by which the values of voxels of the non-homogeneous basic macro blocks can be encoded. This coding types can be separated into two large groups each consisting of four types of encoding. The first group of coding types exploits only the properties of the macro-block currently being processed. The second group of coding types performs analytical comparison to the adequate macro-block on the previous slice (called reference macro-block) and exploits the differences between macro-blocks for more efficient encoding.

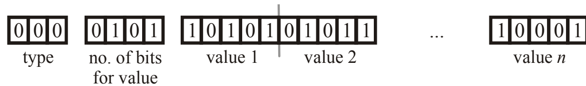
Clearly, the best possible type of encoding must be determined before the data is actually stored. To select the best encoding the typical properties of the macro-block must be determined at first (average value of the voxels in the macro-block, maximal absolute data value in the macro-block, maximal difference to the average value, number of voxels that are different between the processed macro-block and the reference macro-block). Using these properties we can mathematically determine the number of bits required for storing the data for each coding type. The coding type that requires the lowest number of bits is then used for actual storage of the data.

In the continuation we give a short description of each of the coding types. Along with the textual description of the coding types we also present the bit-mask for each coding. Although bit-masks differ from one coding type to another all coding types include two common parts. The first common part is the type number. The type number is a value between 0 and 7 and is always stored in the first three bits and is required for the decompression algorithm to choose the correct decompression routine. The second common part is number of bits used for storing the values of the macro-block. Number of bits used for storing the values can of course differ depending on which type of coding the algorithm decides to use. The position of this value in the bit-mask is not the same in all coding types. Let us remark here, that in the provided graphical representations of the bit-masks, four bits are used for storing the number of bits used for value representation. This of course means that only 1-16 bits can be used for representing values. However, in the actual implementation we could simply resize the reserved space from four to five or more bits if necessary.

3.2.1 Coding macro-blocks using analysis of the properties of the macro-block

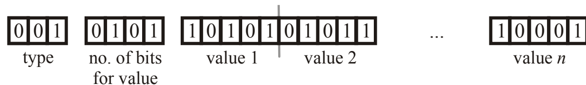
3.2.1.1 Encoding macro-blocks by storing the differences to the average value.

This coding type is used for storing the differences to the average value of the voxels in the macro-block. So, instead of the absolute value of the each voxel a difference to the average value is stored for each voxel. Bit mask of this coding type is:



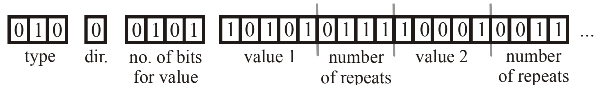
3.2.1.2 Encoding macro-blocks by storing absolute voxel values

This coding type is very similar to coding type 1. The only difference is that the actual value of the appropriate voxel is stored.



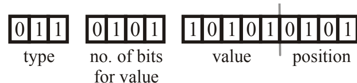
3.2.1.3 Encoding macro-blocks using run-length encoding

The third coding type represents the macro block by using run-length encoding (RLE). The RLE can be performed in either horizontal or vertical direction, thus we can represent the coding direction with just one bit, which immediately follows the type number. The bit-mask of this coding type is



3.2.1.4 Encoding single voxels

It often happens that all of the voxels in a macro block except one represent empty space. We introduce a special coding type for representing such macro blocks. This coding type only stores the value of the voxel and the position of that voxel inside the macro-block. Here is the bit-mask:



3.2.2 Coding macro-blocks using the analytical comparison to the reference macro-block in the previous slice

The second group of coding types consists of coding types which are based on the comparison of the processed macro-block with the adequate macro-block on the previous slice. Such macro-block is called a reference macro-block. However, for better compression we also compare the processed macro block with slightly offset reference macro-blocks. This is useful as some entities can slightly move, grow or shrink from one slice to another, as shown in Fig. 1. We can efficiently determine and exploit these occurrences by using offset reference macro blocks.

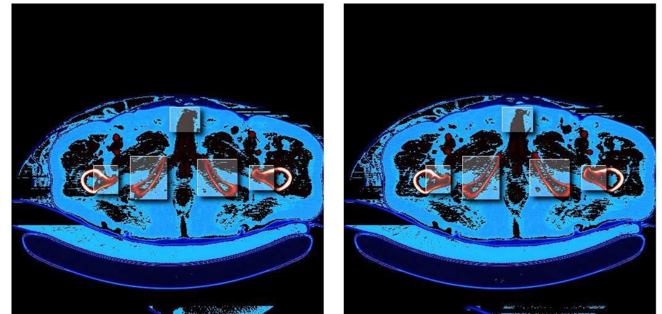
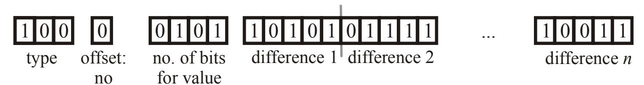


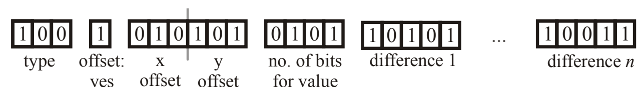
Fig. 1: Two adjacent slices taken from a real dataset with marked significant differences.

3.2.2.1 Encoding macro-blocks by storing the differences to the values of the reference macro-block

Using this coding we calculate the difference between the value of the voxel in the processed macro-block and the corresponding voxel in the reference macro-block and store those differences. Bit mask of the coding type, if the position of the reference macro-block corresponds to the position of the processed macro-block is:



Bit mask for the case where the offset reference macro block was used:

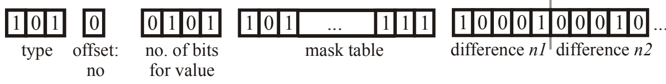


The difference between the bit-mask for using non-offset and bit-mask for using offset reference macro-blocks is the same for all coding types. Therefore, in the continuation we will present only the bit-mask for non-offset reference macro-blocks.

3.2.2.2 Encoding macro-blocks by using the masking table and storing the differences to the values of the reference macro-block

This coding type expands the coding type presented in 3.2.2.1 by introducing the mask table. The mask table tells which voxels on the processed macro-block actually differ from corresponding voxels in the reference macro-

block. If the voxels are equal we store the value 0 into the appropriate position in the mask table. However, if the voxels are different, we store the value 1 into the appropriate position in the mask table and store the difference. Bit-mask:

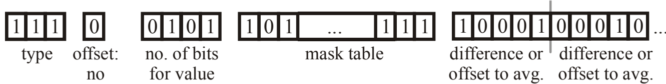


3.2.2.3 Encoding macro-blocks by using the masking table and storing the differences to the average value

This coding type is very similar to coding type 3.2.2.2. Again the mask table is used to mark which voxels differ, however, instead of storing the difference between the values of corresponding voxels, the difference between the voxel value and the average value of the macro-block is stored. Bit-mask:

3.2.2.4 Encoding macro-blocks by selecting the smaller value from the difference to the value of the reference macro-block and the difference to the average value

The last coding type introduces a combination of two previously described coding types. Again, the mask table is used, but the mask table has a slightly different meaning in this coding. At first we calculate the difference between the value of the voxel and the value of corresponding voxel in the reference macro-block (lets call this value *difference*). Then we calculate the difference between the voxel value and the average value of the macro block (lets call this value *offset to average*). If *difference* is smaller than *offset to average*, then we write 1 in the appropriate position in the mask table and store the *difference*, otherwise we enter 0 into the mask table and store the *offset to average*. The bit-mask of this coding looks like:



3.3 Results

Presented algorithm has been tested on various real voxel datasets. Tests have proven that the presented algorithm can efficiently compress various types of volumetric data without losses. The achieved compression ratio is between 2.5 and 4.5. Detailed results are presented in Table 1. We have also compared our compression algorithm to the two most popular general-purpose compression methods ZIP and RAR. In about 80% of cases our special-purpose algorithm performed better compression. However, we must also take into consideration that the quadtree based compression supports progressive reconstruction, which is impossible if the data is compressed using a general-purpose compression algorithm. An example of progressive reconstruction and visualization of a voxel dataset is shown in Fig. 2.

Dataset	Original size	Quadtree based compression		
		QSF (KB)	VXF (KB)	Together (KB)
Human	217.600	9.395	45.086	54.481
Bunny	184.832	2.595	54.233	56.828
Pelvis	89.088	975	21.578	22.553
Backpack	190.976	2.338	41.360	43.698
Colon	226.304	3.425	91.345	94.770
Aneurism I	262.144	3.541	89.442	92.983
Liver	119.808	2.036	48.011	50.047
CT head	14.464	225	5.122	5.347
MR brain	13.952	216	6.203	6.419
Engine	14.080	231	4.040	4.271
Head	20.352	310	6.536	6.846
Tooth	20.608	198	3.999	4.197
Aneurism II	16.384	75	272	347
Foot	16.384	248	4.056	4.304
Bonsai	16.384	260	2.808	3.068
Skull	16.384	474	8.923	9.397
Teapot	11.392	164	1.015	1.179

Table 1: Results of the usage of the quadtree-based algorithm for compression of volumetric data on various real datasets.

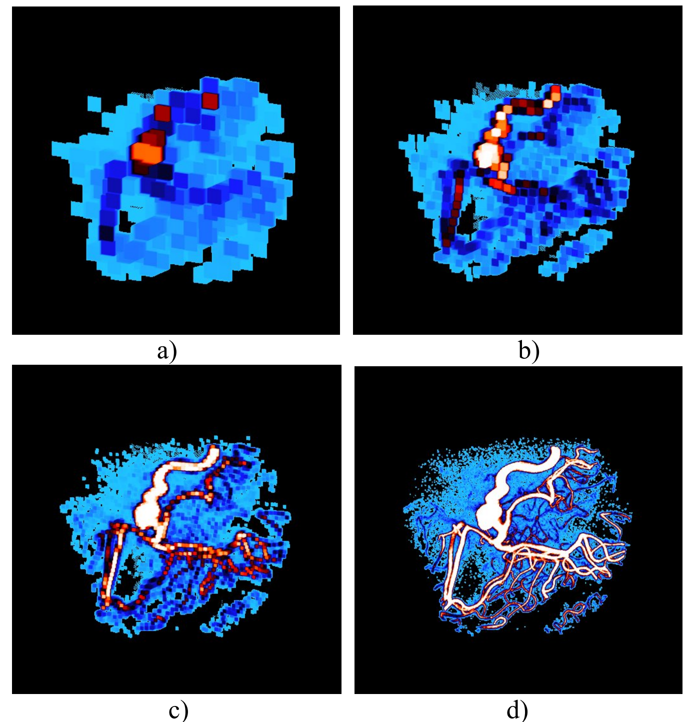


Fig. 2: Progressive reconstruction and visualization shown on the Aneurism dataset.

4 Conclusion

In the paper we have presented a novel algorithm for lossless compression of volumetric data. While the basic idea of the algorithm is the same as in our previously published work, we upgraded the algorithm by introducing several new techniques for determination of similarities between slices of voxel data and entropy encoding. These new techniques significantly improve the efficiency of the algorithm, what has been proven by testing the algorithm on various datasets.

In the future we will try to additionally improve the efficiency of the algorithm by introducing some techniques that have until now only been used in lossy compression algorithms. We would also like to expand our algorithm into 4D space, so that it could also be used for compression of time-varying volumetric datasets.

Acknowledgments

We are grateful to the Slovenian Research Agency and to the Ministry of Defence of Republic of Slovenia for supporting this research under the project M2-0141 - Development of a system for visualization of realistic terrain required for training battle crews.

References:

[1] Kaufman, A. E., Voxels as a Computational Representation of Geometry, *SIGGRAPH Course Notes*, 1996.

[2] Srihari, S. N., Representation of Three-Dimensional Digital Images, *ACM Computing Surveys*, Vol. 13, No. 4, 1981, pp. 399-424.

[3] Stytz, M. R., Frieder, G., Frieder, O., Three-Dimensional Medical Imaging: Algorithms and Computer Systems, *ACM Computing Surveys*, Vol. 23, No. 4, 1991, pp. 421-499.

[4] Engel, K., Kraus, M., Ertl, T., High-Quality Pre-Integrated Volume Rendering Using Hardware-accelerated Pixel Shading, In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHCIS workshop on Graphics hardware*, Los Angeles, California, 2001, pp. 9-16.

[5] Ma, K.-L., Lum, E. B., Muraki, S., Recent Advances in Hardware-Accelerated Volume Rendering, *Computers & Graphics*, Vol. 27, No. 5, 2003, pp. 725-734.

[6] Hadwiger, M., Kniss, J.M., Rezk-Salama, C., Weiskopf, D., Engel, K., *Real-Time Volume Graphics*, A K Peters, 2006.

[7] Klajnšek, G., Žalik, B., Progressive lossless compression of volumetric data using small memory load, *Computerized medical imaging and graphics*, Vol. 29, No. 4, 2005, pp. 305-312.

[8] Ning, P., Hesselink, L., Fast Volume Rendering of Compressed Data. In: Bergeron, D., Nielson, G. (eds.) *Proceedings of the 4th conference on Visualization '93*, San Jose, California, 1993, pp. 11-18.

[9] Hesselink, L., Ning, P., Vector Quantization for Volume Rendering, In: *Proceedings of the 1992 Workshop on Volume visualization*, Boston, Massachusetts, pp. 69-74.

[10] Muraki, S., Volume Data and Wavelet Transforms. *IEEE Computer Graphics and Applications, Vol.* 13, No. 4, 1993, pp. 50-56.

[11] Ginesu, G., Giusto, D.D., Pearlman, W.A., Lossy to lossless SPIHT-based volumetric image compression, *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '04)*, Montreal, Quebec, Canada 2004, pp. 693-696.

[12] Hashimoto, M., Matsuo, K., Koike, A., High efficiency loss-less coding method with 3-dimensional wavelet transform for volumetric data, *IEEE Nuclear Science Symposium Conference Record*, Portland, Oregon, USA, 2003, pp. 2780-2784

[13] Fowler, J.E., Yagel, R., Lossless Compression of Volume Data. In: Kaufman, A., Krueger, W. (eds.) *Proceedings of the 1994 symposium on Volume Visualization*, Tysons Corner, Virginia, USA, 1994, pp. 43-50.

[14] Yeo, B., Liu, B., Volume Rendering of DCT-Based Compressed 3D Scalar Data. *IEEE Transactions on Visualization and Computer Graphics,* Vol. 1, No. 1, 1995, pp. 29-43.

[15] Chiueh, T., Yang, C., He, T., Pfister, H., Kaufman, A., Integrated Volume Compression and Visualization. In: Yagel, R., Hagen, H. (eds.), *Proceedings of the 8th conference on Visualization '97*, Phoenix, Arizona, 1997, p. 329.

[16] Zhu, Z., Machiraju, R., Fry, B., Moorhead, R., Wavelet-Based Multiresolutional Representation of Computational Field Simulation Datasets. In: *Proceedings of the 8th conference on Visualization '97*, Phoenix, Arizona, 1997, p.151.

[17] Rodler, F.F., Wavelet Based 3D Compression for Very Large Volume Data Supporting Fast Random Access. In: *Proceedings of the 7th Pacific Conference on Computer Graphics and Applications*, Seoul, Korea, 1999, pp. 108-117.