CEIG'08, Barcelona, Sept. 3–5 (2008)
L. Matey and J. C. Torres (Editors)

S3Dc: A 3Dc-based Volume Compression Algorithm

H. Yela & I. Navazo & P. Vazquez

[1]MOVING, UPC, Barcelona

Abstract

Volumes acquired for medical purposes are continuously increasing in size, faster than graphic cards memory capacity. Large volumetric datasets do not fit into GPU memory and therefore direct rendering is not possible. Even large volumes that still fit into GPU memory make frame rates decay. In order to reduce the size of large volumetric models, we present a new compression scheme.
In this paper we present S3Dc, a lossy volume compression algorithm suitable for scalar values. It is inspired in hardware-accelerated 3Dc normal compression technique. S3Dc allows us to compress the volume in CPU up to a 4:1 or 8:1 ratio, while still yielding good quality results. We provide details on the compression scheme and show how to render directly from a S3Dc compressed texture. Furthermore, we analyze the image quality theoretical error and the average error with several images in order to assess the results.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Graphics data structures and data types

1. Introduction

With the improvement of capture devices and techniques, volume models have grown continuously during the last years. The amount of memory modern GPUs ship with is also growing. Unfortunately, the sizes of volumetric datasets easily surpass the memory capacities of most modern GPUs. Although techniques like bricking, multiresolution, or compression, have been proposed, the efficient rendering of large volumes is still challenging. On the other side, modern GPUs also support some formats of texture which are compressed and decompressed by hardware. However, none of the provided formats is adequate for the scalar data contained in volumetric models.

In this paper we propose a block volume compression technique, dubbed S3Dc (from Scalar 3Dc), inspired by 3Dc method, a lossy compression algorithm which is used to compress 2D normal maps. A given scalar volume, represented as a voxel model, is compressed in CPU by the algorithm presented in Section 3. The result is a set of two volumes of lower size. In order to load them into GPU, this compressed volumes are stored in two 3D textures. In Section 4, a GPU-based S3Dc decompression algorithm is proposed. We will see that both the error due to the lossy compression and the compression ratio depend on the configuration parameters. In Section 5 we will show that a good election of those yields good quality results.

Our main contributions are:

- A compression scheme, dubbed S3Dc (from Scalar 3Dc), suitable for 3D scalar volumetric datasets, that allows for random access.
- A GPU-friendly storage method that allows us to compactly store the compressed information to be passed to the GPU.
- A GPU-based decompression algorithm suitable for directly rendering S3Dc-compressed 3D textures.

Furthermore, we analyze the theoretical, the average, and the visual error of our lossy scheme, and show that the results are satisfactory in terms of compression ratio and decompression speed for a compression of up to 8:1.

The remainder of this paper is organized as follows. In Section 2 we review related work. The overview of our compression scheme is the subject of Section 3. In Section 4 we describe required algorithm modifications to adapt the decompression process to GPU. In Section 5 we show further results of our approach applied to real data sets. We conclude the paper with a detailed discussion and future work.

2. Related Work

There are three kinds of methods taylored to reduce the amount of information sent to the GPU for volume rendering of large volumes: bricking, multiresolution, and compression.

Bricking [Gri05, Bru04, KBKG07] techniques subdivide the large volume into several blocks, named bricks, in such a way each block fits into GPU. Bricks are stored in main memory, then they are sorted either front-to-back or back-to-front order respect to the camera position, depending on the rendering algorithm. In order to avoid discontinuities on brick boundaries when using trilinear interpolation, for computing new values inside the volume, bricks must overlap by one voxel (two if we want to compute gradients on-the-fly). The main drawback of bricking strategies are the high amount of texture transfers required as each brick is sent once per frame. A second problem is the difficulty in determining the adequate size of blocks for obtaining an optimal frame rate, which may vary substantially from one graphics card to another.

The second approach is *multiresolution* model [LHJ99, ZWE*00, BNS00, RV06], originally presented by LaMar *et al.* [LHJ99]. The idea is to render at high resolution only the region of interest and using progressively low resolution when moving away from that region. Their algorithm is based on an octree hierarchy where the leaves of the tree represent the original data and the internal nodes define lower-resolution versions. This technique allows memory savings for empty or uniform portions of the volume data by omitting sub-trees of the hierarchy. Furthermore, rendering performance may increase due to lower sampling rates for certain blocks or omitting of empty blocks. Boada et al. [BNS00] propose a new texture memory representation and a management policy that substitute the classical one-texel per voxel approach for a hierarchical approach. Unfortunately, multiresolution data structures have been built for CPU purposes and its translation to GPU's is not straightforward due to the high number of texture accesses that would require.

The last method is *data compression*. Compression techniques consist on decreasing the size of volumetric data applying an specific encoding algorithm. This has been faced using wavelets [Gla95, KS99], or compressed texture formats [Bro00, Cra04, ATI05]. Wavelets transforms offer considerable compression ratios in homogeneous regions of an image while they conserve the detail in the non uniform zones. However, wavelets approaches usually require a costly compression preprocess. Moreover, current implementations are CPU-based, thus, interactive frame rates are difficult to obtain. A GPU-decompression of a wavelet-transformed data is not efficient due to the number of texture fetches required. Recently, modern GPU's have been equipped with hardware that natively supports rendering compressed textures, such as DXT [Bro00] for 2D textures or VTC [Cra04] for 3D textures. The advantage of hardware-based techniques is that they decode and render much faster than software versions. Although the compression they provide is at maximum 8:1, they do not support scalar volume data but only RGB(A) color format. As a consequence, if we try to code scalar values in a RGB(A) format, we will observe block artifacts for non-smooth data.

2.1. 3Dc Compression

Another compressed texture format supported by GPU's is 3Dc [ATI05]. 3Dc provides an algorithm for 2D normal maps compression and it proceeds as follows: first, each normal vector is normalized and only x and y components are stored in two channels of the same texture (ATI2N texture format). Then, for each channel and independently, texture map is broken up into blocks containing 4×4 texels each. The maximum and the minimum values are determined for each block, and these are stored as 8-bit values. A set of six intermediate values are then calculated, equally spaced between the minimum and the maximum. Each component is assigned a 3-bit index corresponding to whichever of these values is closest to its original value. This process is depicted in Figure 1. Thus, the resulting compressed blocks consist of four 8-bit values (the minimum and the maximum for x and y channel), and thirty-two 3-bit values (16 indices for each channel), for a total of 128 bits per block. Since the original blocks consisted of sixteen 32-bit values, for a total of 512 bits, this represents a compression ratio of 4:1. This technique yields good results for 2D normal maps representations but there are not extensions to compression of scalar volume data.

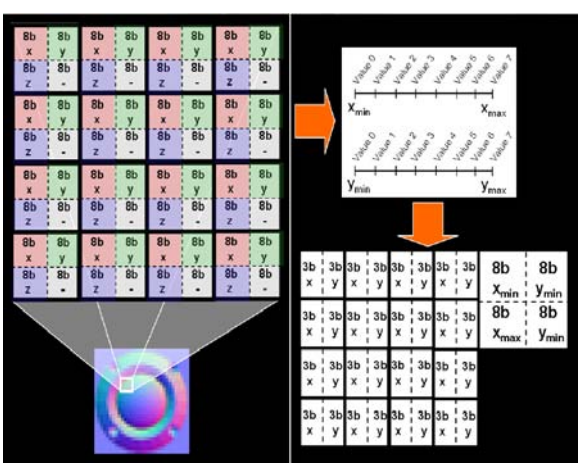

Figure 1: *3Dc compression scheme, taken from [ATI05].*

3. S3Dc Overview

S3Dc (from Scalar 3Dc) is a new GPU-based scalar volume data compression, inspired by 3Dc [ATI05] 2D normal map compression algorithm. We adapt the way 3Dc separately

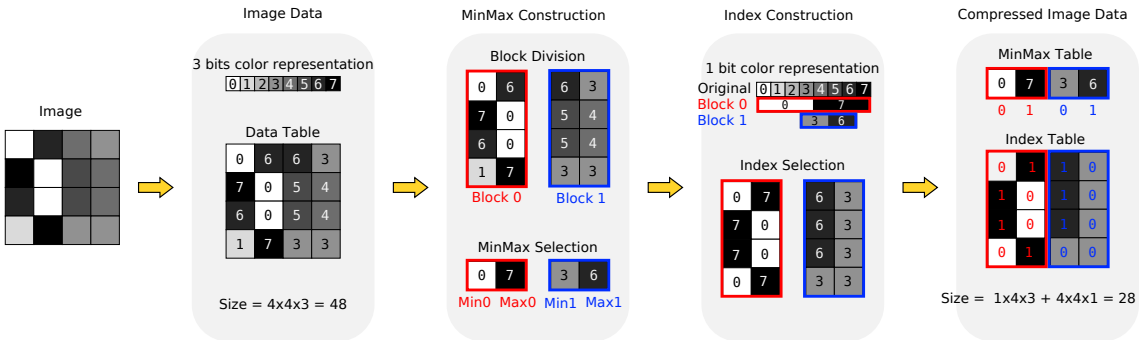

Figure 2: *Overview of our algorithm in 2D.*

compresses each component of a normal vector, using only one component (the scalar value of the volume model).

From an original volume (OV), we will represent its compressed volume (CV) with two new volumes called **MinMax Volume** (MMV), and **Index Volume** (IV). Thus, S3Dc volume compression algorithm is a two-step process:

- First, a virtual block subdivision of the volume is performed, obtaining **for each block its minimum and maximum sample values**. These values will be stored in MMV.
- Next, every sample of OV is processed and the block where the sample belongs is identified. Then the minimum and the maximum values from this block are obtained and from the position where the sample value lies and between these minimum and maximum values, an **index is computed**. These indices are stored as IV.

The compression resides in the fact these indices require less bits than the original sample values. A 2D version of S3Dc compression scheme is depicted in Figure 2.

In order to determine if any lossy compression technique is useful, we also have to analyze the theoretical maximum error, and, more importantly, the average error as obtained with real models. Those are analyzed in Sections 3.4 and 4.4, respectively.

3.1. Compression

Originally we have an original volume (OV), composed of $v_x \times v_y \times v_z$ samples. First, a virtual block subdivision of the original volume is performed, obtaining for each block its minimum and maximum sample values. Then, these minimum and maximum values per block are stored in MMV. The dimensions of the virtual blocks, b_x, b_x, and b_z , determine how many samples lie in every block and they are an input parameter. This process of MMV construction is depicted in Figure 4.

Another input parameter is the number of bits used to represent the index values (i_{bits}), which will determine the possible index values. For example, taking i_{bits} equal to 2, we will have a set of indices equal to $\{0,3\}$. Once we have determined the value of i_{bits}, for each sample from OV we identify the block where it lies, and we take from MMV the minimum and the maximum values corresponding to this block. Next, the index for this sample is computed as:

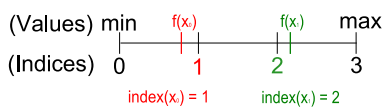

Figure 3: *Example of index computation with $i_{bits} = 2$.*

$$index = \lfloor \frac{value - min}{max - min}(2^{i_{bits}} - 1) + \frac{1}{2} \rfloor \quad (1)$$

3.2. Decompression

In order to obtain the decompressed value from a determined sample, we first obtain its correspondent index from the IV. Then, the block where the sample lies is identified, and we take from the MMV the minimum and the maximum values corresponding to this block. Finally, the decompressed value is computed as:

$$dv = min + (max - min) \cdot \frac{index}{2^{i_{bits}} - 1} \quad (2)$$

3.3. Compression Ratio

The input parameters of S3Dc determine the compression ratio achieved and the theoretical maximum error that S3Dc is doing.

In order to compute the compression ratio, first we must compute the size of OV, MMV, and IV. Each sample of MMV contains two values (see Figure 4), the minimum and

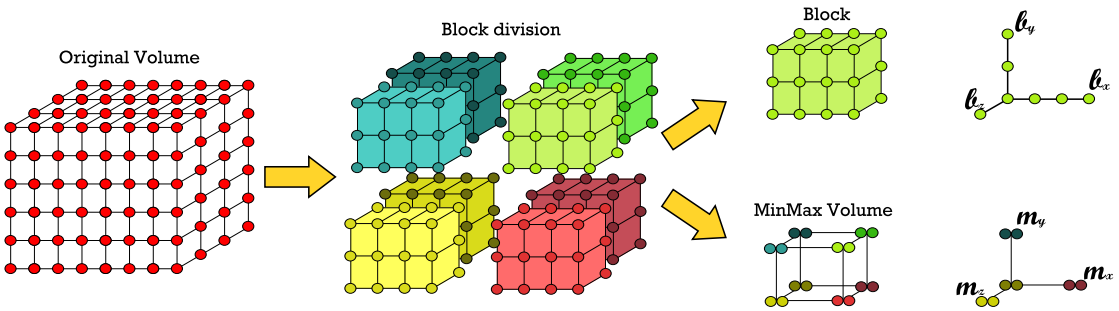

Figure 4: *Overview of MinMax Volume construction.*

the maximum values computed for the block corresponding to this MMV sample. Taking account that v_x, v_y, and v_z are the OV dimensions, that s_{bits} are the bits used to represent original values, and that b_x, b_y, and b_z are the chosen block dimensions (we assume block dimensions are divisors of original volume dimensions), the size of MMV relatively to OV is:

$$\frac{MMV_{size}}{OV_{size}} = \frac{\lceil \frac{v_x}{b_x} \rceil \cdot \lceil \frac{v_y}{b_y} \rceil \cdot \lceil \frac{v_z}{b_z} \rceil \cdot s_{bits} \cdot 2}{v_x \cdot v_y \cdot v_z \cdot s_{bits}} = \frac{2}{b_x \cdot b_y \cdot b_z} \quad (3)$$

IV has the same dimensions of OV and less bits (i_{bits}) per sample. Then, the size of IV relatively to OV is:

$$\frac{IV_{size}}{OV_{size}} = \frac{v_x \cdot v_y \cdot v_z \cdot i_{bits}}{v_x \cdot v_y \cdot v_z \cdot s_{bits}} = \frac{i_{bits}}{s_{bits}} \quad (4)$$

Thus, the **compression ratio** (CR) is the sum of the relative size of MMV and IV relatively to OV:

$$CR = \frac{OV_{size}}{MMV_{size} + IV_{size}} = (\frac{i_{bits}}{s_{bits}} + \frac{2}{b_x \cdot b_y \cdot b_z})^{-1} \quad (5)$$

The main summand of compression ratio is the ratio between the number of bits used to represent an index in IV and the bits used to represent a sample value from OV. The other term, the inverse of the block dimensions product is not so significative, if block size is larger than 4^3, then, this factor has a value of $\frac{1}{64}$. That is, the larger the block size the higher the compression ratio, although this is not a linear relationship, as can be seen in Figure 5. For instance, an OV with 512^3 samples and $s_{bits} = 8$, will require $512^3 \cdot 8 = 128$ MBytes. If we choose blocks composed by 32^3 samples and $s_{bits} = 2$, the compression ratio will be $(\frac{2}{8} + \frac{2}{32^3})^{-1} \simeq 4$.

3.4. Error Analysis

In order to determine if a certain lossy compression technique is useful, we also have to analyze the theoretical maximum error, and, more importantly, the average error as obtained with real models.

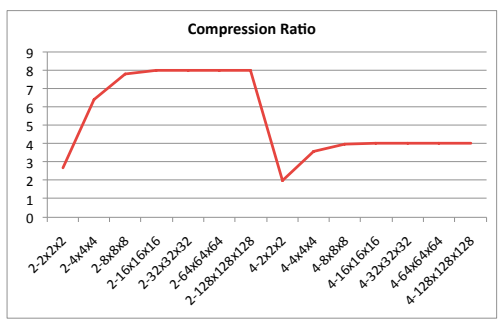

Figure 5: *Graphic showing the compression ratio for different configurations. Configurations are determined by i_bits and block sizes.*

The **theoretical maximum error** is the maximum distance between the original sample value and the corresponding uncompressed value computed from IV and MMV (Equation 2).

For each sample from MMV we may build a segment comprised between *min* and *max*. We subdivide each segment in $2^{i_{bits}} - 1$ intervals in order to assign values to each index. The interval borders represent each index (Figure 6). The worst case will be when $min = 0$ and $max = 2^{s_{bits}} - 1$. Then, when an original sample value is in the middle of this interval, the distance with the uncompressed value will be maximum. Thus, theoretical maximum error (e_{max}) is computed as:

$$e_{max} = \lceil \frac{1}{2} \cdot \Delta Interval \rceil = \lceil \frac{1}{2} \cdot \frac{2^{s_{bits}} - 1}{2^{i_{bits}} - 1} \rceil \quad (6)$$

With the example configuration, we have a theoretical maximum error of $\lceil \frac{1}{2} \cdot \frac{255}{3} \rceil = \lceil 42.5 \rceil = 43$, where 255 is the maximum possible value. Then theoretical maximum error is around 20%.

Usual sample value bits are 8 and 16, and we can represent indices with 1 to 8 bits. For example, if we have 16-bit

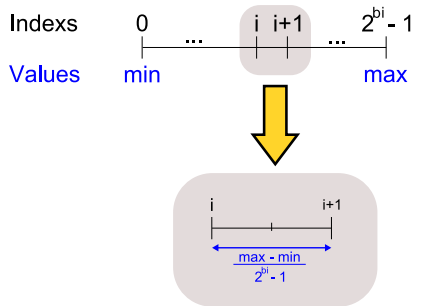

Figure 6: *Theoretical maximum error computation.*

sample values, and 2-bit index, this means 4 possible indices between 0 and 3, the compression ratio will be some value around 8 if the block size is big enough. From the maximum error we can conclude main factor is the number of bits used to represent index values. Obviously, in most cases we will not have the worst scenario, as this depends on the volume data distribution.

In the following sections we propose a couple of configurations that we believe are a good balance between compression ratio, quality, and rendering speed.

4. GPU-Based Decompression and Rendering

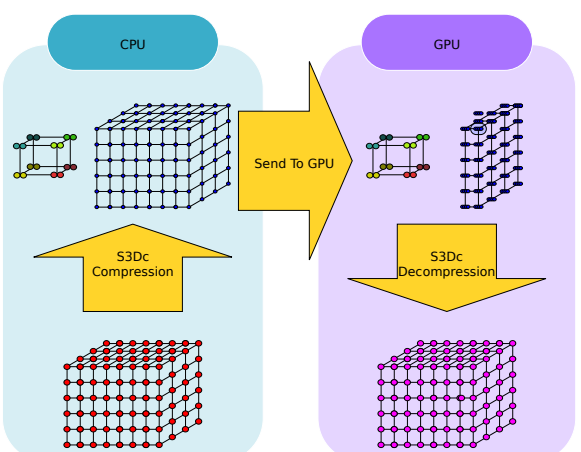

Figure 7: *Step process overview.*

Our purpose is to reduce the amount of GPU memory required for volumetric models. Therefore, we will apply our compression scheme and render the data directly from the compressed volume. This means that we will need to obtain the decompressed values for each sample in the GPU. These accesses to the compressed volume will be performed in the fragment shader.

With this aim, we compress OV in CPU with our algorithm. MMV and IV are generated and stored in 3D textures. Transferring a less amount of data to the GPU we are reducing the CPU-GPU transfer data bottleneck, as shown as Figure 7.

4.1. Algorithm Equivalence CPU-GPU

We want to notice that in terms of compression ratio and theoretical maximum error, CPU algorithm and GPU algorithm are equivalent. We are not increasing theoretical maximum error because decompressing in GPU and not in CPU. We are representing the exactly same values in MMV, and in IV we are not losing precision with the merge/split process when we work with index values.

4.2. GPU Codification

As texture formats are limited, we have to adapt our compression scheme in order to fit into any of the supported formats.

GPU MMV texture will consist in a 3D-texture with internal format Luminance16Alpha16. We assume that sample values are represented with 16 bits. Then, we store in each element two values, luminance channel contains the minimum value and alpha channel contains the maximum value. Using this 3D texture we guarantee we are not losing precision with MMV values because we are using the same number of bits to represent each integer value in CPU than in GPU.

One of the main problems we face when compressing for GPU rendering is that we must keep the number of texture accesses low, otherwise, we will not obtain interactive frame rates. The way we have chosen to solve this problem is to merge the index values by taking a set of correlative index in *x-direction* and manipulate them to finally store them in one element of 3D texture GPU IV. The internal format used is Alpha16. The *merge* and *split* processes are depicted in Figure 8. We will split index values in the decompression process in GPU.

There are more choices in order to solve the problem of limited internal formats, for example we can use RGB(A) format or merge the values in other dimensions at the same time, not only in x. We have chosen to merge only in x direction because it is the easiest to implement.

We do not want to waste texture space, thus, the possible index bits values are 1, 2, 4 or 8, because they are factors of 16, the internal format chosen. In Figure 7 we can see how each sample of GPU IV is obtained by merging index values from CPU IV.

Older GPU's do not support non-power-of-two textures. We may adapt our algorithm to these by adding blank values to our 3D textures until their dimensions will be power

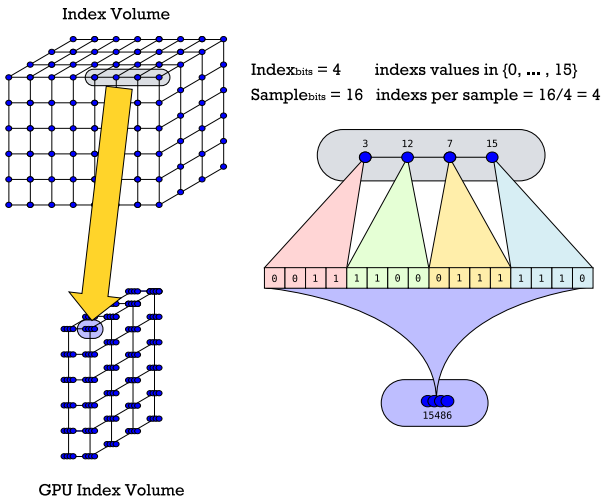

Figure 8: *Overview of merge process.*

of two. As a consequence, the compression ratio obviously reduces, but we still get an improvement over the non-compressed version.

Notice that in terms of compression ratio and theoretical maximum error, CPU and GPU algorithms are equivalent. We are not increasing theoretical maximum error. We are representing the exactly same values in MMV, and in IV we are not losing precision with the merge/split process (Figure 8) when we work with index values.

4.3. Volume Rendering from S3Dc Compressed Textures

In order to visualize compressed volumes, we have two possibilities, texture slicing, or ray-casting. Although our compression scheme is suitable for both, we present here the approach for texture slicing. In the standard approach, a 3D texture is created from the volume data, then, it is sent to GPU and it is visualized via rendering a stack of view aligned proxy quads. When each quad is rendered, GPU access to each rasterized fragment and in each of these fragments the sample value lying in it is computed.

From a given point (x,y,z) of the space, the corresponding texture coordinates in the original volume are computed automatically by the standard visualization algorithm. The texture coordinates of MMV and IV are computed by applying some linear transformations to the texture coordinates of OV. Next, from these texture coordinates, the need values to compute the decompressed value (block's minimum and maximum values, and index value) are obtained by texture fetching.

Furthermore, for volume rendering, we may also use interpolation between the sample values since not all the points inside the volume are represented in it. Two kinds of interpolations are supported by GPU's, *Nearest* and *Linear*. **Nearest access** assigns to non represented points the closest sample value and it requires **2 texture fetches**, and **Linear access** as it computes an eight-neighborhood interpolated value, requires **16 texture fetches**, Linear interpolation yields better results, but is computationally more expensive. In case we want to compute **gradient on the fly**, the number of **texture fetches will be increased by a factor of six** because we need to access to the six face-neighbors. We are able to accelerate linear calculation by reusing texture fetches needed for interpolating the neighbor samples.

4.4. Image quality metrics

Although we already analyzed the maximum theoretic error generated by our lossy scheme, it is more important to compare the results with real-world examples. In this sense, we will use different image comparison metrics as well as a perception-based one, the Structural Similarity. The quality metrics analyzed are:

- **Mean square error**: We measure, for each pixel of the resulting images and every channel on this pixel, red, green, blue, the squared difference between them. We accumulate these values and then the resultant value is normalized by the number of total pixels.
- **Peak signal-to-noise ratio**: It is the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. It is computed as

$$PSNR = 10 * \log_{10}(\frac{MAXi^2}{MSE})$$

  where MAXi is the maximum possible pixel value of the image. Typical values for the PSNR in lossy image and video compression are between 30 and 50 dB, where higher is better.
- **Structural Similarity**: The Structural SIMilarity [WBSS04] (SSIM) index is a method for measuring the similarity between two images. SSIM is designed to improve on traditional methods like PSNR and MSE, which have proved to be inconsistent with human eye perception. The SSIM index is a real value between 0 and 1. A value of 0 would mean zero correlation with the original image, and 1 means the exact same image. 0.95 SSIM, for example, would imply half as much variation from the original image as 0.90 SSIM.

5. Results

We have implemented the proposed method in a 2.6MHz Quad Core PC equipped with 8GB of RAM memory and a GeForce 8800 GTX graphics card with 768MB of memory.

In order to compare the quality of the compressed volume rendering, we tested several models. For the mumerical results appearing later, we used a volume model of

$512 \times 512 \times 486$ samples and 16 s_{bits}, rendered in a standard 3D textures way and with different configurations of our algorithm. We have compared the original volume rendering using linear interpolation with the compressed volume rendering with nearest texture access, linear interpolation and optimized linear interpolation. We are mainly concerned with the qualitative results, that is, the relationship between image quality and compression ratio, presented in Section 4.4 , but we also measured the efficiency of the proposed approach (Sect. 5.1).

5.1. Algorithm analysis

The quality parameters we have computed are the average frame rate obtained, see Figure 9, the average PSNR of the compared image and the average SSIM of this same compared image.

In Table 1 the results obtained with our method are shown. We can observe S3Dc Nearest is the best configuration in terms of frame rate. That is because standard algorithm frame rates are taken from linear access also. We obtain interactive frame rates for S3Dc Nearest access and good image quality results. With S3Dc Linear the image quality results are better, mainly in terms of SSIM, and we have frame rates enough to let interactivity. We can observe that block size has a more important influence in image quality when i_{bits} value is low. Furthermore PSNR and SSIM metrics have a similar response to the variation of algorithm configuration. They yield better results when the i_{bits} value is the largest one and the block size is the lowest possible, but this configuration is also the one with worse compression ratio. As a result, taking account frame rates, image quality metrics, and compression ratio, the algorithm configuration with $i_{bits} = 4$ and block size = 16^3 is the one with better overall results.

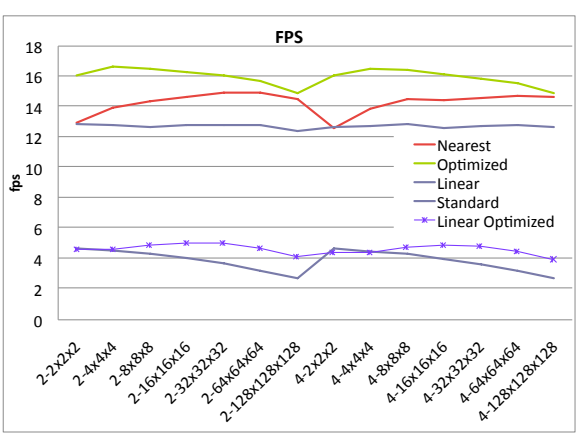

Figure 9: *Frame rate comparison.*

In Figure 12 we can visually compare different algorithm configurations with respect to the image generated with the

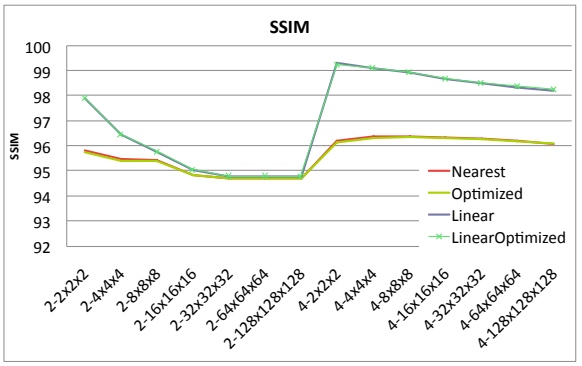

Figure 10: *SSIM comparison.*

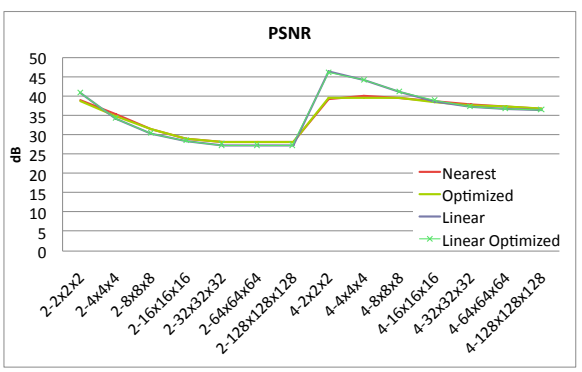

Figure 11: *PSNR comparison.*

standard volume rendering. Top left image shows a rendering without volume compression. Top right shows the image obtained with our empirically determined best configuration: block size = 16^3 and 4-bits index. The middle row contains images resulting from $i_{bits} = 2$ configurations, with block sizes of 2^3 (left) and 64^3 (right), respectively. Although 2 bits is supposed to be the configuration that yields the poorer quality, middle right image shows that results may still be acceptable when the block size is small enough. On the left image we can see an artifact on skull mouth caused because the block is too big to let high range variation, and we only have 4 indices. Finally, bottom row shows two images that encode the difference from this last configuration for nearest texture access, on the left, and for linear interpolation, on the right. We can observe the compared image between standard volume rendering and our nearest/linear interpolation compressed method. The blue pixels represent an error lower of 5%, the green ones and error between 5-10%, and if they are yellow or red the error is bigger than 15%. We can infer from these results that the error usually concentrates in blocks containing the highest gradients in terms of magnitude. On the other hand, uniform blocks usually show low error. On the left image there are more yellow-red pixels

Configuration		Compression Ratio	Frame rate		PSNR		SSIM	
index bits	block size		Nearest	Linear	Nearest	Linear	Nearest	Linear
2	2^3	2,667	16,095 fps	4,57 fps	38,76	40,9	95,75	97,88
2	4^3	6,395	16,65 fps	4,57 fps	34,95	34,23	95,44	96,44
2	8^3	7,757	16,53 fps	4,85 fps	31,52	30,27	95,42	95,76
2	16^3	7,968	16,30 fps	5,05 fps	29,18	28,30	94,86	95,04
2	32^3	7,996	16,07 fps	5,01 fps	28,32	27,49	94,77	94,82
2	64^3	7,999	15,73 fps	4,66 fps	28,20	27,31	94,79	94,83
2	128^3	8,000	14,91 fps	4,12 fps	28,21	27,33	94,72	94,79
4	2^3	2,000	16,07 fps	4,41 fps	39,48	46,19	96,13	99,25
4	4^3	3,554	16,52 fps	4,38 fps	39,61	44,08	96,34	99,11
4	8^3	3,938	16,44 fps	4,72 fps	39,48	41,04	96,36	98,92
4	16^3	3,992	16,15 fps	4,87 fps	38,62	38,77	96,33	98,68
4	32^3	3,999	15,90 fps	4,77 fps	37,77	37,32	96,30	98,51
4	64^3	4,000	15,60 fps	4,49 fps	37,30	36,67	96,21	98,36
4	128^3	4,000	15,03 fps	3,95 fps	36,93	36,35	96,10	98,22

Table 1: *Configuration comparison: In column* Configuration *shows the different configurations we tested for comparison purposes. All images were rendered on a* 768×768 *viewport. Note that the best frame rates are obtained with* nearest *interpolation method, although the ones obtained with the* linear *are still enough to provide good interactivity.* PSNR *and* SSIM *columns show the similarity measures we tested. Note that a value of 35 in PSNR column means we still have a good similarity. SSIM measures percentage of similarity, so values of up to* 95% *are usually good enough.*

than on the right image, and on the right image dark blue and green pixels predominate. This means the accumulate error is bigger on the Nearest image. All the examples in the paper were rendered in a viewport of 768×768 resolution.

In all the presented images, we have compared with a standard volume rendered view. This is our *worst* case in the sense that we have to interpolate, but for other uses of compressed textures (such as for texturing far models or simply to compress any kind of scalar data), the frame rates obtained would be the ones of the *nearest* approach.

5.2. Discussion

As we mentioned earlier in the previous work, there are some other algorithms for data compression in volumetric models. We provide here further details on the advantages and disadvantages of S3Dc with respect to the previously presented ones.

Wavelets are a very powerful compression technique. However, the decompression algorithm requires a very high number of texture accesses in order to obtain a single value. If we also require linear interpolation the cost boosts. Moreover, the nature of wavelet decomposition makes that the information required to reconstruct the signal value, must be gathered from different locations. Therefore, using them for other manipulation techniques that require the access to neighbors seem difficult to implement efficiently.

Vector quantization-based algorithms obtain a very high compression ratio, but image quality may greatly vary according to the distribution of data. We may not easily impose a maximum error, and therefore the results may change unexpectedly from model to model. In our case, the maximum error is bounded and the average error we obtain is relatively low. Like with wavelets, manipulation operations such as segmentation are not straightforward. Moreover, it requires a sensitively costly preprocess.

Concerning hardware-based methods, as already mentioned, they do not support scalar values but normals or RGB or RGBA as base units. Moreover, some parameters is also fixed in the specification and we may not vary (such as number of bits and so on) in order to adapt to our datasets.

There is another possibility with the use of multiresolution techniques, although these are not currently adapted to GPUs and present an overhead in texture access due to the hierarchical representation. However, these techniques are somehow orthogonal to ours and would combine softly.

Finally, S3Dc is at least as efficient as the previous techniques for volume compression and the image quality was assessed with different measurements. It might be combined easily with other techniques such as bricking or multiresolution and further treatment of data may be straightforward. For instance, in segmentation we require the comparison of sampled values and in most cases the minimum-maximum may be used to quickly discard regions of non interest.

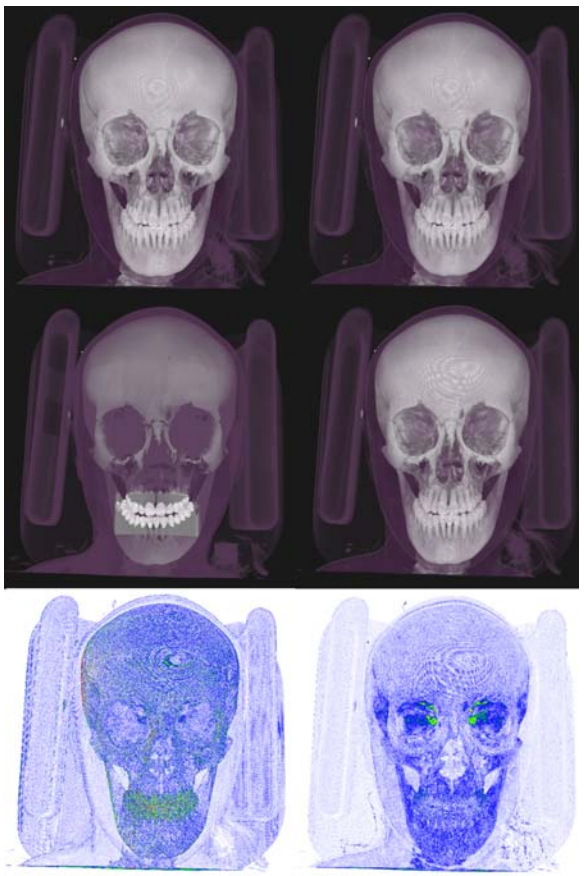

Figure 12: *Image comparison with the rendered volume (top left) and three different configurations. Top right shows the result of the best configuration in terms of image quality* vs *compression ratio. Note that it is quite difficult to distinguish the original one (top left) with the best quality one (top right) where we obtained a compression ratio of 4:1. Middle row shows the result from 2-bits index and big block size (on the left) or small block size (on the right). An artifact on skull mouth appears on the left image because the block size is too big for only 4 index representation.*

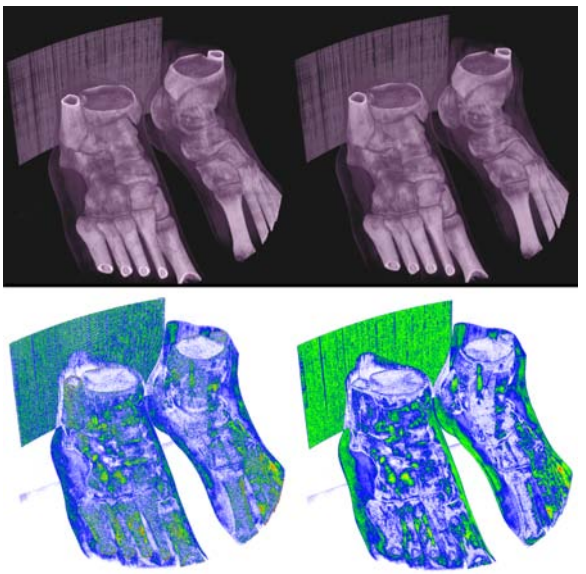

Figure 13: *Result image comparison for the best overall configuration. Top right shows Nearest access and top left shows Linear access. Bottom row shows comparison images. On Nearest comparison, bottom left, there are more yellow-red pixels. On Linear comparison image predominates dark blue-green pixels.*

6. Conclusions and Future Work

In this paper we have presented a new compression scheme based on 3Dc tailored to deal with scalar datasets. The storage strategy and the rendering algorithm has also been presented.

In future, we want to address two different issues: lower compression error, and faster decompression for interpolation.

We expect to reduce the compression errors by performing a previous analysis of the dataset and use a non linear distribution of weighs.

Concerning the efficiency, although rendering volume datasets using linear interpolation has a strong impact in the frame rate, we still have acceptable values (around 5fps) for large viewport sizes (such as 768×768). However, we strongly believe we may improve efficiency by changing the merge/split algorithm and the index internal format.

References

[ATI05] ATI: *3Dc White Paper. ATI Radeon X800.* Tech. rep., ati.amd.com/products/radeonx800-/3DcWhitePaper.pdf, 2005.

[BNS00] BOADA I., NAVAZO I., SCOPIGNO R.: A 3d texture-based octree volume visualization algorithm. In *WSCG'00* (February 2000), Plzen, pp. 1–8.

[Bro00] BROWN P.: *OpenGL S3TC texture compression extension specification*. NVIDIA Corporation, http://opengl.org/registry/, 2000.

[Bru04] BRUCKNER S.: *Efficient Volume Visualization of Large Medical Datasets*. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, May 2004.

[Cra04] CRAIGHEAD M.: *OpenGL VTC extension specification*. NVIDIA Corporation, http://www.opengl.org/registry/, 2004.

[Gla95] GLASSNER A. S.: *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995, ch. Wavelet Transforms, pp. 243–298.

[Gri05] GRIMM S.: *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria, Apr. 2005.

[KBKG07] KOHLMANN P., BRUCKNER S., KANITSAR A., GRÖLLER M. E.: Evaluation of a bricked volume layout for a medical workstation based on java. *Journal of WSCG 15*, 1-3 (Jan. 2007), 83–90.

[KS99] KIM T.-Y., SHIN Y. G.: An efficient wavelet-based compression method for volume rendering. In *PG '99: Proceedings of the 7th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 1999), IEEE Computer Society, p. 147.

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

[RV06] RUIJTERS D., VILANOVA A.: Optimizing gpu volume rendering. *Computer Graphics, Visualization and Computer Vision'2006* (2006).

[WBSS04] WANG Z., BOVIK A. C., SHEIKH H. R., SIMONCELLI. E. P.: The ssim index for image quality assessment.

[ZWE*00] ZIMMERMANN K., WESTERMANN R., ERTL T., HANSEN C., WEILER. M.: Level-of-detail volume rendering via 3d textures. *IEEE Volume Visualization and Graphics Sympsium* (2000).