Real-time Decompression And Visualization Of Animated Volume Data

Stefan Guthe   Wolfgang Straßer*

WSI/GRIS University of Tübingen

Abstract

Interactive exploration of animated volume data is required by many application, but the huge amount of computational time and storage space needed for rendering does not allow the visualization of animated volumes by now. In this paper we introduce an algorithm running at interactive frame rates using 3d wavelet transforms that allows for any wavelet, motion compensation techniques and various encoding schemes of the resulting wavelet coefficients to be used. We analyze different families and orders of wavelets for compression ratio and the introduced error. We use a quantization that has been optimized for the visual impression of the reconstructed volume independent of the viewing. This enables us to achieve very high compression ratios while still being able to reconstruct the volume with as few visual artifacts as possible. A further improvement of the compression ratio has been achieved by applying a motion compensation scheme to exploit temporal coherency. Using these scheme we are capable of decompressing each volume of our animation at interactive frame rates, while visualizing these decompressed volumes on a single PC. We also present a number of improved visualization algorithms for high quality display using OpenGL hardware running at interactive frame rates on a standard PC.

Keywords: Time critical Visualization, Compression for Visualization, Volume Rendering

CR Categories: E.4 [Coding and Information Theory]: Data compaction and compression; I.0.3 [Computer Graphics]: General; I.3.3 [Computer Graphics]: Picture and Image Generation—Viewing algorithms;

1 Introduction

The large datasets generated by today's applications need to be compressed. This is especially the case if the dataset changes over time. The visualization of animated volume data is very interesting for applications like geologic simulations, or captured medical volume data that change over time.

To compress these datasets as high as possible, we chose a lossy compression scheme. This scheme is split into several steps: the coding of an individual volume dataset using wavelet transforms, the quantization and compression of the resulting wavelet coefficients and finally the coding of a whole sequence of volume datasets. To maximize the compression ratio while minimizing the

---

reconstruction error and the decompression time, we investigated a couple of different approaches for each of these steps.

Our algorithm is the first one capable of decompressing and visualizing animated volume data at interactive frame rates, utilizing state-of-the-art video compression algorithms such as wavelet transforms and motion compensation that have been adapted to 3d volume data. We also implemented two high quality visualization algorithms. The first algorithm is based on a shear-warp factorization using OpenGL hardware. It has been modified using register combiners [24] to fix some serious drawbacks of previous approaches with texture hardware. The second algorithm uses 3d textures or register combiners to simulate them.

1.1 Related Work

Wavelet based image compression: A wavelet based image compression has first been proposed and investigated by DeVore et. al. [11] and by Antonini et. al. [3] by simply extending the one dimensional wavelet transforms for higher dimensions using tensor product wavelet transforms. An overview of the field of wavelet based image compression can be found in Vetterli et. al. [26] or in Villasenor et. al. [27], while a more general overview is given by Stollnitz et. al. [25].

Wavelet based volume compression: The wavelet transformation and compression of non-animated volume datasets has been thoroughly discussed during the last few years, resulting in on the fly decompression using the Haar wavelet and a single 3d wavelet transform [13, 16] or multiple 2d wavelet transforms [22]. Although these compressions allow for fast reconstruction of any single sample point and therefore random access within the dataset, they yield fairly good compression ratios but tend to produce blocky images at compression ratios close to or beyond 100:1. In contrast our approach can do compression ratios of up to 200:1.

Wavelet coefficient encoding: Recently developed volume compression algorithms use run-length encoding, zerotrees introduced by Shapiro [23] or schemes similar to zerotrees that use significance maps to encode wavelet coefficients. Since we do not need to access each single sample point, but rather the whole volume dataset at once, we are not constrained to pure zerotrees, but also some combinations using run-length encoding and other, more general, encoders. For comparison between different encoders, we combined the zerotrees with a final step of arithmetic encoding [19], run-length encoding with arithmetic coding and run-length encoding with LZH compression [29].

Compression of animated data: For compressing animated volume datasets we can apply some kind of motion prediction and compensation similar to MPEG [14, 15]. Although the MPEG motion compensation works very well if applied to the blocks used for the discrete cosine transformation it leads to some severe problems if applied unmodified to wavelet transformed images as discussed by Watanabe and Singhal [28]. Their modified motion compensation, the windowed motion compensation, will be used because of its significant lower reconstruction error the resulting compression ratio and its small overhead. These previous 2d algorithms were adapted for the 3d case for this paper.

Volume visualization: There have also been various approaches to the visualization of volume datasets needed in the final step of our algorithm. The visualization using a shear-warp factorization by Lacroute and Levoy [17] has been adapted to modern graphics
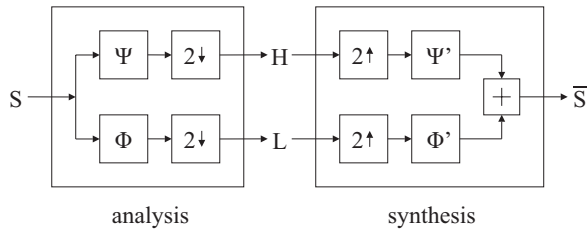
Figure 1: Wavelet transform of a 1d signal S.

Figure 2: Single step of three dimensional translation (L=low pass filtered, h=high pass filtered) and complete recursive decomposition.

hardware using 2d textures by Brady et. al. [4] to achieve interactive frame rates. A different approach using 3d textures by Akley [2] has until recently only been available on graphics workstations, but is starting to show up on standard PC graphics hardware, e.g. the ATI Radeon graphics adapter. Rezk-Salama et. al. [21] used standard PC graphics hardware. In this case the NVidia GeForce graphics adapter is used to implement the tri-linear interpolation that is needed to simulate 3d textures. We will later use this tri-linear interpolation to implement the shell rendering proposed for the ATI Radeon [1] on a GeForce graphics adapter using its register combiners [24].

1.2 Paper Overview

The algorithm is split into three steps to encode or decode a sequence of volumes and an additional step for visualizing each decoded volume. In section 2.1 we take a closer look at the wavelet transforms used, the computational time and the reconstruction error after applying some quantization to the wavelet coefficients that will be discussed in section 2.2. Afterwards we will investigate some compression schemes for these quantized wavelet coefficients in section 2.3. Section 3 will be discussing several ways to encode a sequence of volumes and compare their compression ratio, decompression time and reconstruction error. In section 4 we will discuss improved visualization methods using standard PC hardware.

2 Single Volumes

The compression of single volumes consists of three steps: the wavelet transform, the quantization of the wavelet coefficients and their compressed encoding. A three dimensional wavelet transform of the whole volume at once, instead of a block-wise wavelet transform [16, 13], was chosen to maximize the compression ratio for any given quality and avoid blocking artifacts for higher order wavelets. The quantization uses different accuracies for each kind of wavelet coefficients to minimize the information to be compressed.

2.1 Wavelets

The wavelet transform of a 1d signal can be regarded as the filtering of this signal with both the wavelet Ψ and the scaling function Φ and the downsampling of the resulting signals by a factor of 2 (see figure 1). The wavelet is a high pass filter and the scaling function a low pass filter. Note that the total amount of information is not modified. It is just a change of the basis of the function space. The part of the wavelet transform up to now is called *analysis* of the signal, while the following part is called *synthesis*. The values of H and L are called wavelet coefficients in the discrete case. The reconstruction upsamples the transformed signals again, filters them using Ψ' for the high sub band, the wavelet coefficients representing high frequencies and Φ' for the low sub band, the wavelet coefficients representing low frequencies. The two resulting signals are added together and reproduce the original signal without any loss.
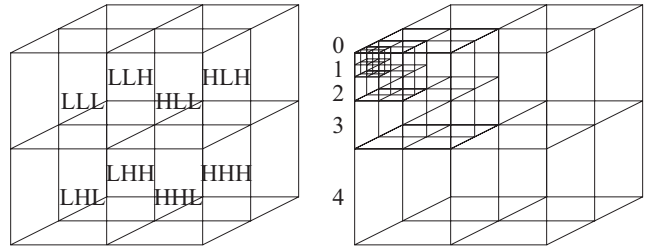
To transform signals of higher dimension, we apply the 1d wavelet transform in all three dimensions separately, resulting in a 3d tensor product wavelet transform [25] as seen in the left part of figure 2. After the first wavelet transform of the whole volume, we apply the 3d wavelet transform to the low sub band recursively. We repeat this step four times to get the sub bands seen in the right part of figure 2. Any further wavelet transforms does usually not reduce the size of the compressed volume, but rather introduce more visual artifacts.

Previous approaches of volume compression used the simple Haar wavelets. Higher order wavelets on the other hand have some properties that make them very suitable for implementation of a compression algorithm, such as more than one vanishing moment, i.e. polynomials that map to a single wavelet coefficient only. The Daubechies wavelets [7] are the wavelets with maximum vanishing moments at a given support. But there is also a major drawback when using higher order wavelets due to their longer support. If we want to reconstruct a signal using a wavelet of support greater than 2, we need a periodic extension of the original signal and therefore for each of the sub bands as this is the only way to maintain the same number of wavelet coefficients for any family of wavelets regardless of their support. Although this might seem to solve all our problems with higher order wavelets, we have to keep in mind that our volume is not of periodic nature and may therefore have very high contrast between its opposite surfaces.

A better way of extending our signal for higher order wavelets would be a symmetric extension, i.e. the signal is mirrored at it's borders. Although this removes our border problem, we now need symmetric wavelets [9]. Yet constructing a symmetric wavelet is not possible, as we require the wavelet to be orthonormal to guarantee a change of the basis of the functional space. Although it is possible to construct wavelets that are as symmetric as possible, i.e. the Coiflets proposed by Coifman and later constructed by Daubechies [10], they can still not be used with symmetric extension. However, if we use a different filter for reconstruction, we can construct symmetric, so called bi-orthogonal wavelets. The CDF wavelets introduced by Cohen, Daubechies and Feauveau [6] are the most widespread bi-orthonormal wavelets used for image encoding and will also be used in addition to Haar wavelets, Daubechies Wavelets and Coiflets.

2.2 Quantization

After the wavelet transform, the complete volume is represented by wavelet coefficients in floating point values, enabling us to reconstruct the original volume correctly. To reduce the data to be stored while maintaining a minimum error in L^2 norm, we first have to choose the dead zone [26], i.e. the range that will be mapped to zero, for each different region of wavelet coefficients seen in figure 2 to cut down the number of non-zero wavelet coefficients. The remaining coefficients then have to be scaled and quantized appropriately for the further compression. To take the sensitivity of the human visual system for different frequencies into account, we use

level	LLH	LHH	HHH	dead zone
0	0.80	0.60	0.50	0.33
1	0.80	0.60	0.50	0.60
2	0.60	0.40	0.30	0.60
3	0.30	0.20	0.15	0.60
4	0.13	0.08	0.06	1.00

Table 1: Factors f for quantization ($f * maxQuant$) and dead zone ($1 + f * maxDead$) of different levels and regions of the wavelet coefficients, resulting in $2f * maxQuant + 1$ possible quantized values per wavelet coefficient.

individual dead zones and scaling factors with different regions of the transformed volume after all coefficients have been normalized to a range of $[-1, 1]$.

There is only one region of wavelet coefficients LLL_0 that has been transformed using the low pass filter only. At each of the five recursion levels there are three regions that have been filtered through the high pass filter once $LLH_i/LHL_i/HLL_i$, three regions that have been high pass filtered twice $LHH_i/HLH_i/HHL_i$ and only one region that has been high pass filtered three times HHH_i. Since we treat all three directions equally only the number of high or low pass filters applied to each region are of interest. This delivers a total of 16 different regions of wavelet coefficients.

With a user defined global maximum $maxQuant$ for the number of coefficients and a global maximum dead zone $maxDead$ the dead zones and quantization steps are defined as follows. For LLL_0 the interval $[0, 1]$ is split into $maxQuant$ steps while the dead zone is defined as $1 + \frac{4}{15} maxDead$ as large as the difference between two quantized values. The maximum number of coefficients and the relative size of the dead zone for the remaining regions of wavelet coefficients within the range of $[-1, 1]$ can be seen in table 1. The dead zone and the quantization steps have been adapted to represent similar amounts of visual contrast depending on the underlying frequency, according to the threshold contrast sensitivity function [12] that represents the sensitivity of the human visual system for contrast at different frequencies.

The user defines $maxQuant = 127 + quality * 128$ and $maxDead = 20 * (255 - quality)/255$ by specifying the $quality$ parameter in the range $[0, 255]$ to easily define the compression ratio while trying to minimizing the visual loss and drop in PSNR.

2.3 Encoding

After the quantization step we have to encode the wavelet coefficients to store them using as few memory and decompression time as possible. To choose the right type of compression, we first have to take a close look at the data to be compressed. The goal of the quantization was to cut as many coefficients as possible down to zero, so the most simple and fastest compression can be achieved by run-length encoding all zeros. To produce as few overhead as possible, we just compress a run of n zeros by storing 0 followed by $n - 1$. Therefore a single 0 becomes a sequence of 0 0 and is therefore the only way that the compressed data will expand. If n exceeds the number of possible quantization steps, the run is split into several compressed sequences. We also have to specify a traversal order through the regions of the wavelet coefficients. The fastest way to encode and decode the wavelet coefficients is to store each line within a region of wavelet coefficients individually (see figure 3a). While this already produces long runs of zeros, there is a more sophisticated methods for achieving even longer runs. Similar to the zero trees [23], we use a depth first traversal through an octree that holds all our wavelet coefficients within each region of wavelet coefficients separately (see figure 3b). As regions of zero coefficients are more likely to be stored in a single run, this delivers even longer runs of zeros. However the non-linear memory access
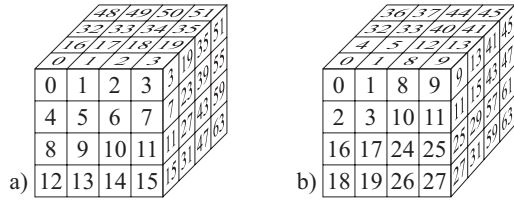

Figure 3: Order of traversal within a region of wavelet coefficients.

results in a severe loss of decoding speed.

This already results in fairly good compression ratios while needing very little time for decompression, but there is still a very simple and effective way to further improve the compression. The LZH algorithm [29] compresses a repeating sequence of different values by storing the reference to the last appearance of the same sub string. While the compression, due to the searching for matching strings within a given range of previously compressed coefficients, takes quite long, the additional time needed for decompression is nearly not noticeable. Reading the compressed data may even be slightly faster than reading uncompressed data due to the reduced amount of disc access.

The compression ratios up to now are quite good, but for storing an animated volume dataset, we have to achieve compression ratios beyond 100:1 for single volumes. The aim of arithmetic coding is basically to store a sequence of intervals using as few bits as possible by storing the shortest bit string within this interval. The main part of every compression using arithmetic coding is the model that translates the incoming symbols into intervals and vice versa. As these models can be very powerful, they can be used to implement any compression algorithm.

The most simple model presented by Moffat, Neal and Witten [19] that can be applied for encoding our wavelet coefficients is the adaptive model. At the beginning of each region of wavelet coefficients, the intervals of each quantized coefficient are of equal size and the counter of their references is set to one. After encoding a coefficient, the corresponding counter is increased by one and all intervals are resized accordingly. As the counters are organized in a binary tree, the intervals are only to be computed if the corresponding coefficient is encoded. The time needed for the computation of these intervals and therefore the compression or decompression time needed for a single coefficient is $O(log(n))$ with n being the number of possible coefficients. Therefore the decompression gets faster as the number of possible wavelet coefficients decreases.

There is also a way to implement an optimized kind of run-length encoding into the model. After a zero is encoded, the model switches to *zero mode*. While in *zero mode*, the arithmetic coder always receives the interval between 0 and 1, i.e. it does not encode anything, as long as the coefficients that are to be compressed remain zero. After receiving a non-zero coefficient, the arithmetic coder receives a combined interval representing the number of zeros received and the new wavelet coefficient. The number of zeros is stored similar to the run-length encoding. Up to n zeros are stored using only a single interval, while values beyond n are stored by an interval that notifies $n+$ zeros. Using a value of 127 for n showed up to be very good for nearly all datasets and quantizations, resulting in 128 intervals for run-length encoding that are updated using the same adaptive scheme as used for the wavelet coefficients. Therefore the length of each run is compressed in a more optimal way than by a run-length encoding prior to the arithmetic coding. An additional optimization is to store a bit-pattern, that is also encoded arithmetically using only two symbols, to mark unused wavelet coefficients. This results in the compression of a region consisting of zeros to this previously stored bit-pattern only.

The last method of compressing the wavelet coefficients is the zerotree coding of Shapiro [23] combined with an arithmetic en-
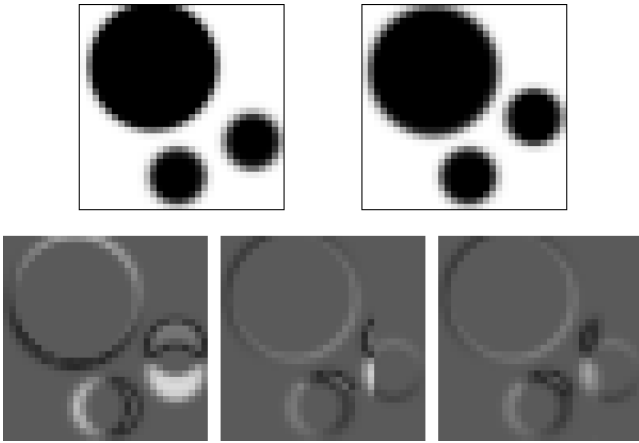
Figure 4: Previous image and current image (upper row). Differential image, standard motion compensation and windowed motion compensation (lower row).
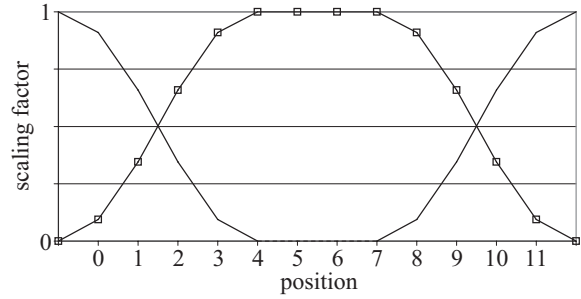

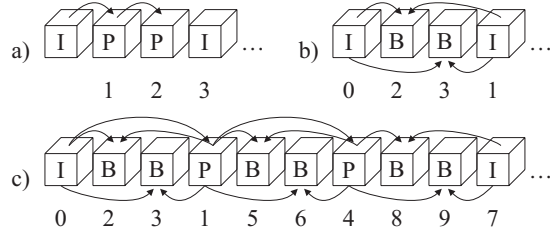Figure 5: Scaling function used for the cosine windowed motion compensation with neighboring windows.

Figure 6: Reconstruction and storage order for different types of volumes and sequences, similar to MPEG. a) motion compensated differences b) motion compensation only, c) popular MPEG order

coder using an adaptive model. Although this might seem the most natural way to exploit the huge amount of zeros and the hierarchical nature of the wavelet coefficients, this turns out not to be the best encoder most of the time and also by far the slowest one.

3 Volumetric Animation

To store animated volumes rather than single volumes effectively requires to exploit the temporal coherency between consecutive volumes. The most simple way to exploit the temporal coherency is storing the difference between the current and the previous volume rather than the current volume. Although this already reduces the compressed data significantly, it is not sufficient for storing large volumetric animations, therefore we have adapted the motion compensation used for video encoding.

3.1 Motion Compensation

The easiest way is to store differential volumes only, as seen in figure 4 (shown for 2d images for clarification). As this is not very effective for our wavelet compression scheme, we implemented a simple motion prediction and compensation. The motion prediction is done by simple block matching of 8^3 blocks between the two volumes. The motion compensation is done before the differential encoding to reduce the differential content. A block of high similarity, i.e. minimum mean square error, in the previous image is computed by searching this minimum starting from a motion vector of length 0 using 15 steps in all three directions. This is similar to finding the correct motion vector using optical flow methods. This results in 31^3 possible motion vectors that have to be stored using some kind of encoding. The search for the local minimum mean-square error guarantees that most of the resulting motion vectors will be of zero length or at least close to zero length making an arithmetic encoding with a simple adaptive model the best choice as encoder.

The usual, i.e. MPEG, method for computing a motion compensated image is to map each block of the to be constructed volume onto a block of the previous volume as seen in figure 4 for two dimensional images. However in combination with wavelet transformation, this results in severe problems if the motion vectors of two neighboring blocks are different, i.e. regions of high contrast are present. Using wavelet transformation, these high contrasts result in large wavelet coefficients in regions that correspond to high frequencies and therefore low compression ratios. There is

another drawback using this simple approach. Due to the nature of the quantization scheme applied to the differential images these wavelet coefficients will be quantized very strong and therefore result in a large error.

The solution to this problem is the windowed motion compensation introduced by Watanabe and Singhal [28]. The blocks are extended to 12^3 overlapping blocks and filtered by a function with a cosine falloff at both ends (see figure 5) in all three directions as the sum of all neighboring window scaling functions and therefore the sum of all voxel weights with the window is already one. Although the computational overhead introduced by these overlapping blocks is with a theoretical value of 2.375 very high, it shows up that this has no severe impact on the performance in practice, due to the higher number of cache hits if the volume is reconstructed voxel by voxel, rather than block by block. The volume that has to be encoded using wavelet transforms does no longer have regions of high contrast, as seen in figure 4 and therefore less wavelet coefficients differ from zero.

Similar to the naming conventions of the MPEG compression [14, 15], a wavelet compressed volume is called I-volume (see figure 6). We use two different ways to reconstruct an individual volume using motion compensation. The P-volume is reconstructed by using a motion compensated version of the previous I- or P-volume and wavelet compression of the difference between this predicted and the actual volume. The B-volume is reconstructed by using a weighted average between a motion compensated version of the last and the next I- or P-volume similar to the compression scheme used in MPEG compression as seen in figure 6. There is no fixed sequence of I-, P- and B-volumes but any sequence of P- or B-volumes between two I-volumes can be defined. Note that the volumes are not stored in their original order, but in the order of their first usage, i.e. the B-volumes are stored after the next P- or I-volume. In our experiments, we analyzed various sequences for compression ratio and visual impression. It turned out that using the popular MPEG sequence (figure 6c) delivers the best results.
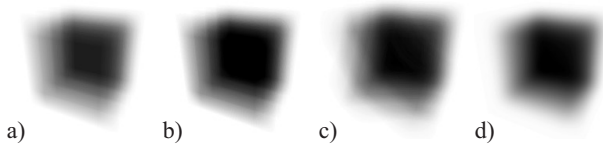
a) b) c) d)

Figure 7: Visualization using stack of textures with uniform transparency (a), register combiner (b), 3d texture (c) and raycasting (d).

4 Playback

The single volumes of the animation are decompressed in their storage order rather than in the order of playback. To compensate the different times needed for the decompression of an individual volume, we have to take a closer look at the decompressed volumes for each new time step. For this we choose the third sequence in figure 6. In a setup step, the first I-volume and the first P-volume are decoded to make sure that all the volumes for interpolation of the next volume (a B-volume) are present. The second and the third volume are decoded without any special treatment. Reaching the fourth volume, the already decoded P-volume, we decode the next P-volume as this one is needed for the next B-volume. After decoding the next two B-volumes, we start decoding the next sequence, starting with it's I-volume. Before displaying the first volume of this new sequence, we decode the next P-volume and reach the same state as at the beginning of our decoding. Thus we only have to decode one volume at a time, resulting in smoother playback of our volumetric animation.

To achieve an on the fly decompression in real time, we have to further optimize the decompression. Using LZH coding the bottleneck of the algorithm is writing the decoded wavelet coefficients into their correct position, therefore we restrict our self to storing them line by line rather than in the octree depth first order. Most of the time this reduced the compression ratio slightly but makes better usage of the write cache and thus speeds up the decompression significantly. Using the arithmetic coding, we have an integer multiplication and an integer division as part of the main interval coder and therefore no need to optimize the writing of the wavelet coefficients in terms of cache hits, as this does not result in any noticeable speedup.

4.1 Visualization

We still need a high quality visualization in real-time to display our decoded volume data. The fastest way of visualizing a volume using standard pc hardware is the usage of a texture stack along the main viewing direction as proposed by Brady et. al. [4]. This stack of textures is combined using planes along each texture and alpha blending hardware. As we need a stack of textures for each of the main viewing directions, this results in storing the volume three times in texture memory. However, during playback of a volumetric animation, the viewing direction only changes between different volumes and therefore a single stack of textures is sufficient. The usage of textures also enables us to easily define a transfer function between the values stored in the volume and the ones stored in the texture stack. Although this gives a good first impression of the dataset (see figure 7a), modern pc hardware allows for more sophisticated algorithms, that do not produce as many artifacts as this approach.

Up to now we ignored the effects seen in figure 8 that lead to some severe problems if we change from one texture stack to another, or if we rotate a dataset, as the opacity of the volume seams to change. The register combiner of the GeForce GPU from NVidia allows us to modify the opacity of the volume data depending on the angle between the surface normal and the vector pointing from the viewing location to a point on the surface and therefore allows
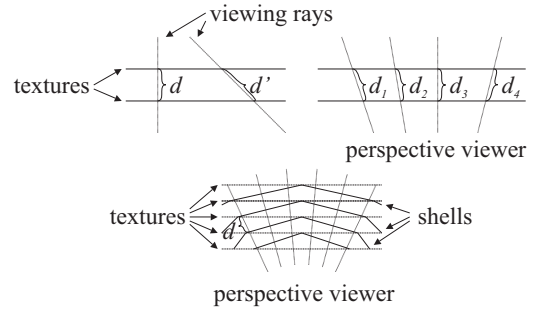

Figure 8: Different distances d between textures introduced by different viewing points and perspective distortion and correction of these differences using 3d textures.

for better visualization algorithms. Due to the exponential behavior of the opacity and the lack of this kind of operation within the combiner, we have to simulate this exponential falloff. The main idea for the approximation is to interpolate between the opacity and the squared opacity at each pixel of the display. In order to do so, we build a texture to look up suitable weights for the linear and squared α values using the angle between the viewer and the surface normal. A first approach for generating this lookup texture is to store the distance minus one $d - 1$ between two slices as weight for the squared opacity. Although this roughly represents the correct alpha values, there is still a large error for small values, i.e. very transparent regions, as seen in figure 9. Since a large error in a very transparent region produces much more visual artifacts than a large error in a very opaque region as the error is accumulated in regions of high transparency, we optimize the lookup texture to minimize the relative mean square error ($(error/orig.alpha)^2$), by storing $t(d)$ instead of $d - 1$. The relative mean square error is given by

$$\int_{d=1}^{2} \int_{\alpha=0}^{1} \frac{(1 - t(d))\alpha + t(d)\alpha^2 - \alpha^d}{\alpha} \, d\alpha \, dd \quad \text{or} \quad (1)$$

$$\int_{\alpha=0}^{1} \frac{(1 - t(d))\alpha + t(d)\alpha^2 - \alpha^d}{\alpha} \, d\alpha \quad (2)$$

for a specific value of d. Since we don't have to specify $t(d)$ but only need its values for a limited number of distances d and with a limited accuracy we can minimize the previous equation numerically by trying all 256 possible values for $t(d)$. As already mentioned this reduces the error for all but the very opaque regions. The approximation of the exponential falloff now only produces the error seen in figure 10. The resulting visualization can be seen in figure 7b.

Although this approach removes most of the problems mentioned above, there are still some small visual artifacts if we move from one texture stack to another. The easiest way to remove this effect is to render the volume from all three directions and combine the resulting images using a weighted average. This can also be done very effectively using texture hardware. The major drawback of this approach is that we have to transfer all three texture stacks to the graphics adapter, resulting in a severe loss of speed.

Another way to render the dataset with correct transparencies is to utilize 3d textures and shells (small subsections of a sphere) which completely avoids the switching of texture stacks and perspective distortion. Unfortunately among the consumer hardware only the ATI Radeon, that has also been used for testing purposes, is able to handle 3d textures in hardware. On the other hand, the GeForce is able to do tri-linear interpolation between any two slices of our dataset, as shown by Rezk-Salama et al. [21], so all we have to do is make sure that no polygon needs any interpolation between three or more slices during their construction as seen in figure 8.
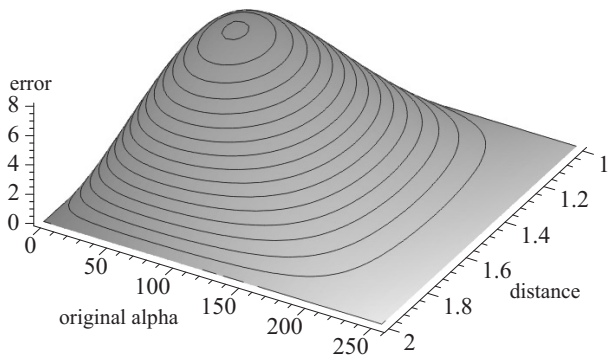
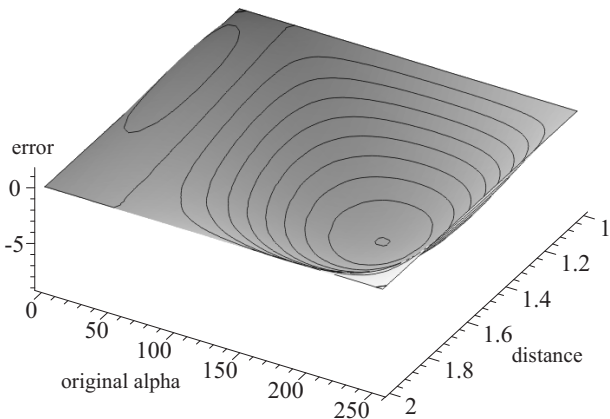Figure 9: Resulting absolute error introduced by using a simple lookup texture.

Figure 10: Resulting absolute error in the range of 0 to 255 introduced by using an optimized lookup texture, to produce less relative error ($error/orig.alpha$).

Using this simple but very effective trick, we are able to simulate 3d textures on a GeForce graphics adapter.

There is still one problem left, the undersampling along the viewing axis. We only sample at the resolution equal to the distance between two slices, as seen in figure 7c. The undersampling can only be removed by using more shells which is not very effective due to the limited precision of the frame buffer or by using raycasting. Although raycasting produces the best results (as seen in figure 7d), it can not be used for interactive visualization of our animated datasets using standard PC hardware due to the time consuming calculations that have to be carried out. Another option is to use a special purpose hardware within a standard PC such as the Vizard II [18] or the VolumePro [20]. However special purpose hardware is not as widely spread as the need for volume visualization.

5 Results

All tests have been carried out on an AMD K7 running at 800 MHz using a GeForce 2 graphics adapter (first configuration) or on an AMD K7 running at 1000 MHz using a Radeon graphics adapter (second configuration).

As expected the usage of higher order wavelets does not only reduce the size of the compressed volumes while enlarging the Peak-Signal to Noise Ratio (PSNR) as seen in table 2, but does also significantly improve the visual impression as seen in figure 11 and

wavelet	I-vol.	P-vol.	B-vol.	ratio
Haar	47.363	47.022	40.613	1:27.30
Daubechies 4	48.673	47.784	40.756	1:30.96
Coiflets 6	48.615	47.751	40.750	1:33.27
CDF 2/6	49.015	47.990	40.787	1:34.28

Table 2: PSNR comparison between different wavelets using maximum quality, the popular MPEG like sequence and arithmetic compression.
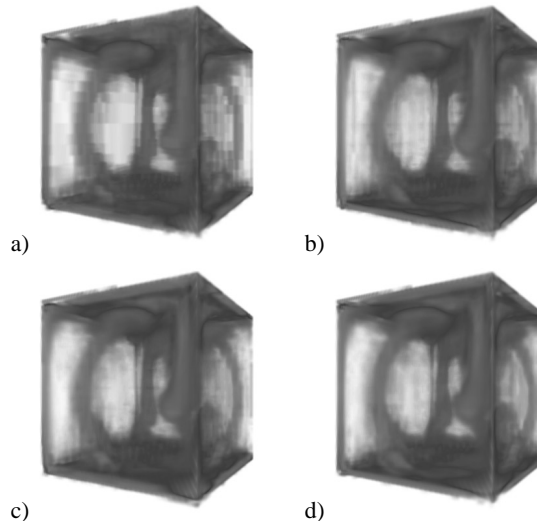

Figure 11: Static volume compressed at a ratio of about 40:1 using Haar wavelet , Daubechies wavelet (b, support 4), Coiflets (c, support 6) and CDF wavelet (d, support 2/6) wavelets.

better preserves features. The original volume can be seen in the color plates.

The PSNR of I- and P-volumes also depends on the compression ratio as seen in figure 12. The quality of the B-volumes on the other hand does only roughly depend on this quality setting (see figure 12), as these volumes are not reconstructed using wavelet transforms but using motion compensation only.

As already mentioned, the decompression times achieved using arithmetic encoding depend on the quantization used, as both the number of zeros increases and the depth of the tree that has to be updated dynamically decreases. Using LZH encoding, neither of these two properties does have any effect on the decompression times as seen in table 3. The LZH encoding performs a lot better than the arithmetic encoding in terms of speed (up to two times faster using high quality settings) but worse in terms of compression ratio (about 30% more compressed data at moderate to high quality). This allows for about 4 frames per second regardless of the chosen quality. Note that the speed on the second configuration is not limited by the decompression time, but rather by the memory bandwidth during the 3d wavelet transform.

As seen in figure 13 the PSNR of the B-volumes (the dotted line) is significantly lower for high quality volume animations. Removing the motion compensation at these high quality settings results in an improvement of the PSNR but also increases the size of the compressed data, as seen in table 4. At a lower quality setting, the PSNR increases if we use motion compensation again while also decreasing the size of the compressed data. Testing different qualities and sequences demonstrated that the popular MPEG sequence (the first sequence in figure 4) should always be used with only one
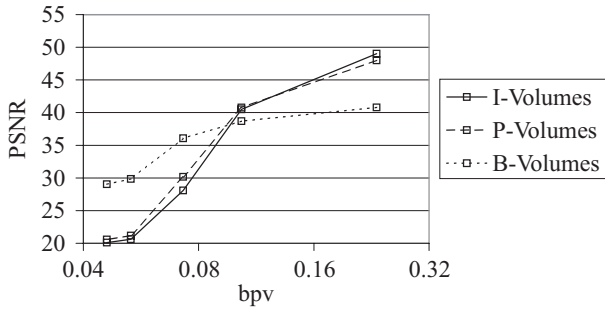
Figure 12: Average PSNR of each different kind of volume against bits per voxel (CDF wavelet with support 2/6 and popular MPEG like sequence).

quality	arithmetic	fps1	fps2	LZH	fps1	fps2
255	34.28:1	1.58	1.77	25.89:1	3.23	3.62
127	77.28:1	2.06	2.34	56.10:1	3.25	3.62
63	109.69:1	2.35	2.74	82.32:1	3.25	3.62
31	150.34:1	2.64	3.01	134.59:1	3.27	3.62
15	173.64:1	2.80	3.17	171.15:1	3.28	3.62

Table 3: Comparison regarding compression ratio and frames per second (using 2d textures on system 1 and 2) between different encoders using various qualities, the popular MPEG like sequence and the CFD wavelet with support 2/6.

exception. If we desire a nearly lossless compression of every volume, rather than every third volume, we should use only I-volumes or I-volumes and P-volumes as this will result in the highest possible PSNR.

Although the 2d textures are a lot faster on both configurations than the 3d textures (see table 5), we still get interactive frame rates on both configurations that do not have a heavy impact on the playback speed of a compressed volume animation. On the other hand, the optimized 2d textures running on the GeForce2 graphics adapter do not need any additional time and sometimes even produce less visual artifacts as the 3d textures (see volume borders in figure 14).

If we wish to compress a volume animation without generating too many noticeable visual artifacts that will playback at interactive frame rates of about 4 frames per second on our testing configurations, we achieve compression ratios of about 50:1 using CDF wavelets, a quality setting of about 127, the popular MPEG sequence and LZH encoding. If we need high compression ratios rather than fast visualization, we are able to reach a ratio of about 75:1 by replacing the LZH encoding by arithmetic encoding with-
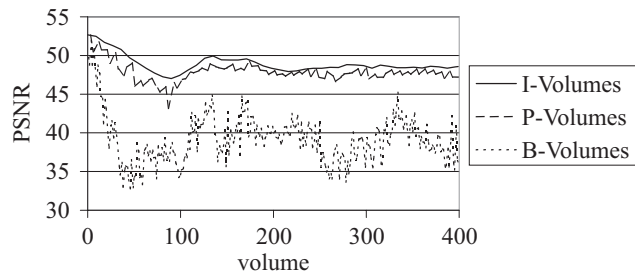

Figure 13: PSNR of each kind of volume of the first 500 volumes of a 2000 volume animation (CDF wavelet with support 2/6, maximum quality and popular MPEG like sequence).

sequence	ratio (255)	PSNR	ratio (15)	PSNR
IBBPBBPBB	34.28:1	43.308	173.64:1	26.159
IBB	28.84:1	43.702	160.70:1	26.040
IPP	13.77:1	48.408	168.85:1	20.396
I	10.09:1	49.004	153.43:1	20.089

Table 4: Comparison between different sequences using CDF wavelet with support 2/6, arithmetic encoding and maximum (intermediate) quality.

configuration	2d textures	3d textures
GeForce2, AMD K7 800 MHz	78.62 fps	12.03 fps
Radeon, AMD K7 1000 MHz	116.26 fps	10.01 fps

Table 5: Time needed for visualization of a static volume dataset on the two different configurations (without any decompression).

out any additional loss of data. Using a transfer function of high contrast will emphasize visual artifacts. Our example animation (figure 11) has an absolute derivative of 5 in the alpha component of the transfer function. A transfer functions of lower contrast will allow for a compression ratio of 100:1 and beyond without visual artifacts.

6 Conclusion & Future Work

In this paper we have shown a very efficient approach to decompress and visualize animated volume datasets in real time on standard pc hardware. The favored compression scheme uses a quality setting of 63 and the popular MPEG sequence with either arithmetic or LZH coding. The presented algorithm does not exploit the possibility for parallelization of the wavelet transform or the motion compensation and therefore leaves a lot of room for further optimization using a single processor (3DNow or SSI instructions) and multiple processors. Although the sole visualization of each volume is quite fast, this part can also be split up into several subvolumes that are to be rendered using a cluster of standard PCs.

Replacing the wavelet transform with the corresponding lifting steps according to Daubechies and Sweldens [8] is a further possible optimization that also enables us to implement a lossless compression scheme using integer wavelet transforms as supposed by Calderbank [5]. However lifting steps only pay off for wavelets with longer support that have not been examined in our experiments by now.

7 Acknoledgemets

References

[1] Radeon sdk: Ext_texture3d extension. *http://www.ati.com/ na/pages/resource_centre/dev_rel/devrel.html*.

[2] K. Akeley. RealityEngine graphics. *Computer Graphics*, 27(Annual Conference Series):109–116, 1993.

[3] Marc Antonini, Michel Barlaud, Pierre Mathieu, and Ingrid Daubechies. Image coding using wavelet transform. *IEEE Transactions on Image Processing*, 2(1):205–220, April 1992.
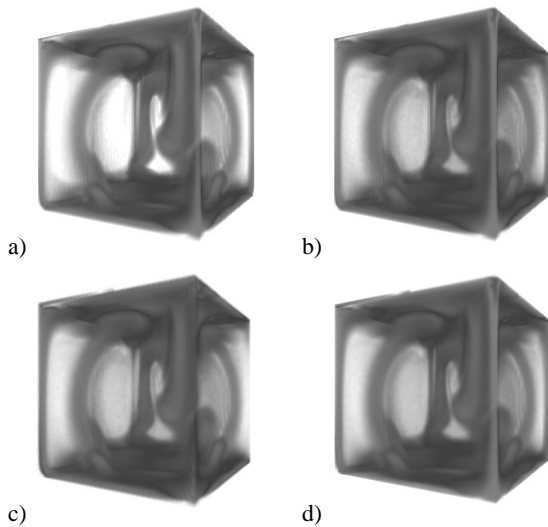
Figure 14: Comparison between different kinds of visualization using 2d textures (a), optimized 2d textures (b), 3d textures (c) and raycasting (d).

[4] Martin L. Brady, Kenneth Jung, HT Nguyen, and Thinh Nguyen. Two-phase perspective ray casting for interactive volume navigation. In *IEEE Visualization '97*, October 1997.

[5] R. Calderbank, Ingrid Daubechies, Wim Sweldens, and Boon-Lock Yeo. Wavelet transforms that map integers to integers. Technical report, Department of Mathematics, Princeton University, 1996.

[6] Albert Cohen, Ingrid Daubechies, and Jean-Christophe Feauveau. Biorthogonal bases of compactly supported wavelets. *Comm. Pure Appl. Math.*, 45:485–560, 1992.

[7] I. Daubechies. Orthonormal bases of compactly supported wavelets. *Comm. Pure Applied Math.*, XLI(41):909–996, November 1988.

[8] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. Technical report, Bell Laboratories, Lucent Technologies, 1996.

[9] Ingrid Daubechies. *Ten Lectures on Wavelets*, volume 61 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[10] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets: II. variations on a theme. *SIAM J. Math. Anal.*, 24:499–519, 1993.

[11] Ronald A. DeVore, Björn Jawerth, and Bradley J. Lucier. Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, 38(2 (Part II)):719–746, March 1992.

[12] R.F. Hess and E.R. Howell. The threshold contrast sensitivity function in strabismic amblyopia: Evidence for a two type classification. In *Vision Research, 17*, pages 1049–1055, 1977.

[13] Insung Ihm and Sanghun Park. Wavelet-based 3D compression scheme for very large volume data. In *Graphics Interface*, pages 107–116, June 1998.

[14] ISO/IEC. MPEG-1 coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s. *ISO/IEC 11172*, 1993.

[15] ISO/IEC. MPEG-2 generic coding of moving pictures and associated audio information. *ISO/IEC 13818*, 1996.

[16] T. Kim and Y. Shin. An efficient wavelet-based compression method for volume rendering. In *Proceedings Pacific Graphics*, pages 147–157, 1999.

[17] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(Annual Conference Series):451–458, July 1994.

[18] M. Meißner, U. Kanus, and W. Straßer. VIZARD II: A PCI-card for real-time volume rendering. In Stephen N. Spencer, editor, *Proceedings of the Eurographics / Siggraph Workshop on Graphics Hardware (EUROGRAPHICS-98)*, pages 61–68, New York, August 31–September 1 1998. ACM Press.

[19] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[20] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings,*, Annual Conference Series, pages 251–260, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman.

[21] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In Stephan N. Spencer, editor, *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 109–118, N. Y., August 21–22 2000. ACM Press.

[22] F. Rodler. Wavelet based 3D compression with fast random access for very large volume data. In *Proceedings Pacific Graphics*, pages 108–117, 1999.

[23] Jerome M. Shapiro. An embedded hierarchical image coder using zerotrees of wavelet coefficients. In James A. Storer and Martin Cohn, editors, *Proceedings DCC'93 (IEEE Data Compression Conference)*, pages 214–233, Snowbird, UT, USA, April 1993.

[24] J. Spitzer. Geforce 256 register combiners. *http://www.nvidia.com/Developer/*.

[25] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgann Kaufmann, San Francisco, CA, 1996.

[26] Martin Vetterli and Jelena Kovačević. *Wavelets and Subband Coding*. Prentice Hall, New Jersey, 1995.

[27] John D. Villasenor, Benjamin Belzer, and Judy Liao. Wavelet filter evaluation for image compression. *IEEE Transactions on Image Processing*, 4(8):1053–1060, August 1995.

[28] H. Watanabe and S. Singhal. Windowed motion compensation. In *Proc. SPIE's Visual Comm. and Image Proc.*, volume 1605, pages 582–589, 1991.

[29] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23:337–343, May 1977.
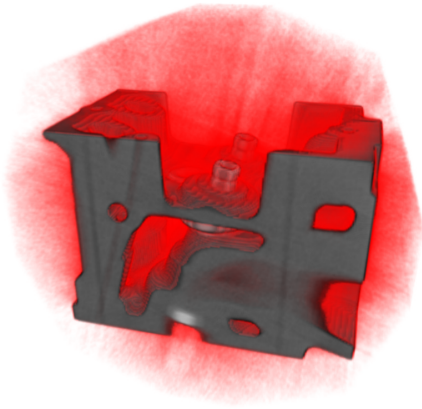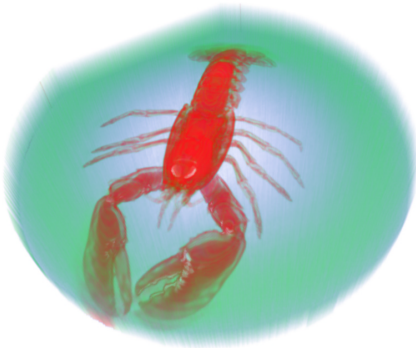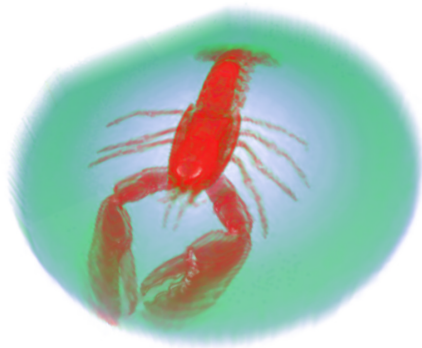
Original engine dataset
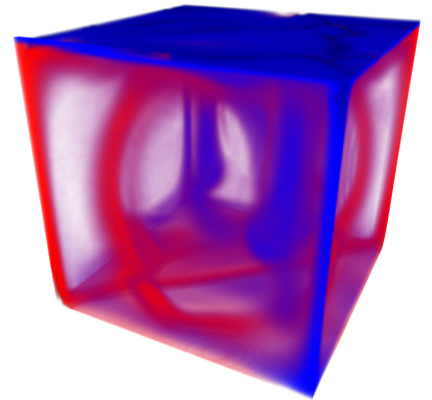
Compression ratio 200:1, Haar wavelet

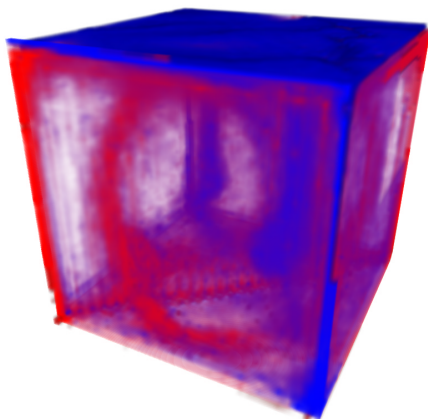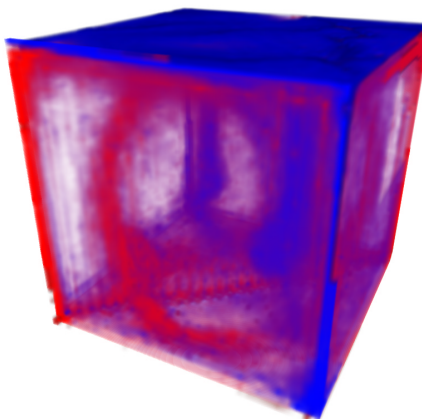Compression ratio 200:1, CDF wavelet

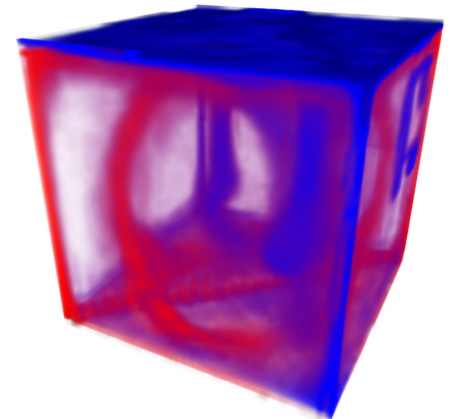Original lobster dataset

Compression ratio 100:1, CDF wavelet

Original volume 1000 of an animation of 2000 volumes

I-only sequence, quality 63, compression ratio 31:1, I-Volume

P-only sequence, quality 63, compression ratio 50:1, P-Volume

MPEG sequence, quality 63, compression ratio 110:1, B-Volume