UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Direct Processing of Compressed Volume Data

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Thomas Höllt

Erstgutachter:	Prof. Dr. Stefan Müller Institut für Computervisualistik, AG Computergraphik
Zweitgutachter:	Dipl.-Inf Matthias Raspe Institut für Computervisualistik, AG Computergraphik

Koblenz, im Mai 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☒	☐

....

(Ort, Datum) (Unterschrift)

Abstract

Using programmable graphics hardware (GPU) is the de-facto standard for real time volume rendering nowadays. In addition to that, GPUs are often used for non-graphical tasks to accelerate complex computations and also allow the direct rendering of (intermediate) results.
However, the amount of graphics memory can become a problem when working with large volume datasets. Even though todays graphics hardware provides more memory than ever before, the amount of data is also increasing rapidly.

In order to overcome this, lots of compression algorithms have been developed and some of them even are hardware-accelerated. As these implementations only support a small range of formats and often do not provide sufficient quality, custom algorithms have been implemented which often utilize shader programs for decoding and encoding. While this has proven useful for visualization, providing interactive framerates for direct volume rendering, the algorithms focus on displaying the data, not processing it.

In this thesis different compression techniques are compared with focus on their suitability for processing in the compression domain. A wavelet transform based compression scheme is implemented which allows lossless as well as lossy compression of volume data. Image processing operations are classified based on their applicability in the wavelet compression domain. Based on this classification different image operations are exemplarily implemented. Furthermore for visualization multi-planar reconstruction directly from the compressed data is presented.

The results of this thesis are compared to processing in the spatial domain, showing advantages and shortcomings. Concluding an outlook on possible future work is given.

Kurzfassung

Der Einsatz von programmierbarer Graphikhardware (GPU) ist heutzutage der de-facto Standard für Echtzeit Volumen Rendering. Auch werden GPUs ein Bereichen außerhalb der Computergraphik eingesetzt um komplexe Berechnungen zu beschleunigen und (Zwischen-) Ergebnisse zu visualisieren.
Die begrenzte Größe des Graphikspeichers kann bei der Verarbeitung von großen Volumendaten Probleme bereiten. Auch wenn die Speicherkapazität heutiger Graphikkarten größer ist als je zuvor so macht das rapide Wachstum der Datenmengen diese Entwicklung zunichte.

Um diesem Problem entgegenzuwirken wurde eine Reihe von Kompressionsalgorithmen, manche sogar hardwarebeschleunigt, entwickelt. Da diese Implementationen jedoch nur eine begrenzte Menge von Formaten unterstützen und oft keine ausreichende Qualität bieten wurden angepasste Algorithmen entwickelt, welche oft Shader Programme für De- und Encoding einsetzen. Auch wenn sich dies zur Visualisierung als nützlich erwiesen hat und interaktive Frameraten bei direktem Volumen Rendering ermöglicht bieten diese in erster Linie zur Visualisierung entwickelten Algorithmen keine Vorteile zur Verarbeitung von komprimierten Daten.

In dieser Arbeit werden verschiedene Kompressionstechniken mit Hauptaugenmerk auf ihre Einsatzmöglichkeit für direkte Verarbeitung in der Kompressionsdomäne verglichen. Als Ergebnis dieses Vergleichs wird ein Kompressionsverfahren basierend auf der Wavelet Transformation, welche sowohl verlustbehaftete als auch verlustfreie Kompression ermöglicht implementiert. Anschließend erfolget eine Klassifizierung von Bildverarbeitungsoperationen anhand ihrer Anwendbarkeit in der Kompressionsdomäne und basierend darauf werden exemplarisch einige dieser Operation implementiert. Weiterhin wird multiplanare Rekonstruktion zur Visualisierung der komprimierten Daten präsentiert.

Vor- und Nachteile der Ergebnisse dieser Arbeit werden durch den Vergleich mit Verarbeitung von nicht komprimierten Daten gezeigt und abschließend wird ein Ausblick auf zukünftige Arbeit in diesem Bereich gegeben.

Contents

List of Figures

Chapter 1

Introduction

This thesis describes development of techniques to directly process and visualize compressed volume data. Therefore, wavelet compression and decompression functions as well as algorithms to directly visualize and process wavelet compressed volume data were implemented. Interactive framerates for visualization could be achieved and processing algorithms which work directly in the wavelet domain were implemented. Some of them even outperform their counterparts in the spatial domain.

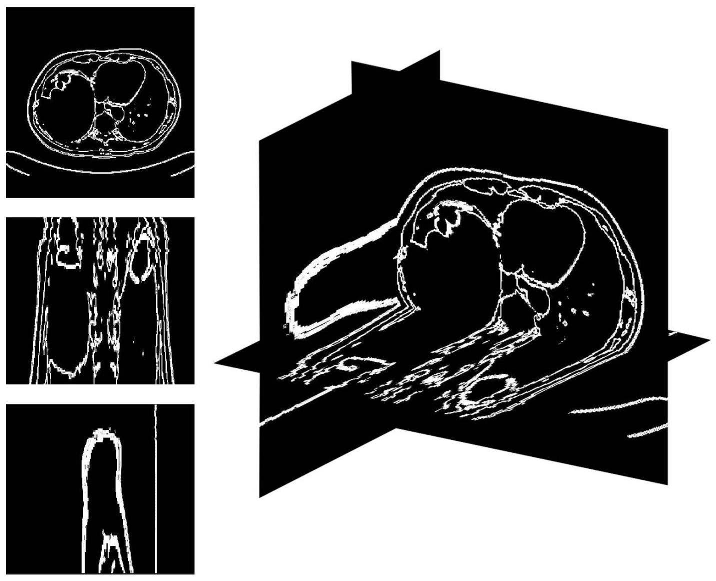


Figure 1.1: Visualization of a $512 \times 512 \times 128$ voxel dataset of a human torso. A binary threshold followed by a Laplace filter were applied to the compressed volume. Visualization directly from compressed data. Compression rate is $1 : 5$. Processing took combined about six seconds. Rendering at interactive framerates

1.1 Motivation

Over the last few years, there has been a tremendous amount of research in the field of volume visualization and processing, especially with the aid of programmable graphics hardware (GPUs). Nowadays, real time direct volume rendering and image processing multiple times faster than on the CPU is possible with the aid of graphics cards which can be found in most off-the-shelf consumer PCs.

Alongside this development, data acquisition techniques evolved to deliver more and more detailed datasets. Today a medical CT-scanner, like the Siemens SOMATOM Definition feature spatial resolution as low as 0.33mm.
Even though full body scans are still the exception due to the exposure to radiation, long-leg studies are acquired routinely, to plan and review leg artery bypass surgeries. These studies consist of approximately $2,000$ slices, each 512×512 pixels large. Stored in a 16bit datatype this results in datasets approximately one gigabyte in size.
While for medical applications the exposure to radiation has to be minimized, this is not the case for other areas like archaeology or industrial applications. For example, researchers from the Rosicrucian Museum and Stanford University created a 3D dataset of an egyptian child mummy, from $60,000$ slices, each as thin as 200 microns. The total size of this dataset is 92 gigabytes.
Another field that produces huge amounts of volume data is simulation. With more computing power, simulations have gotten highly detailed over the last few years. Even small flow simulations reach volume dimensions of $512\times512\times512$. At only a hundred time steps and 8bit per voxel such a dataset would take up 12.5 gigabytes of memory.

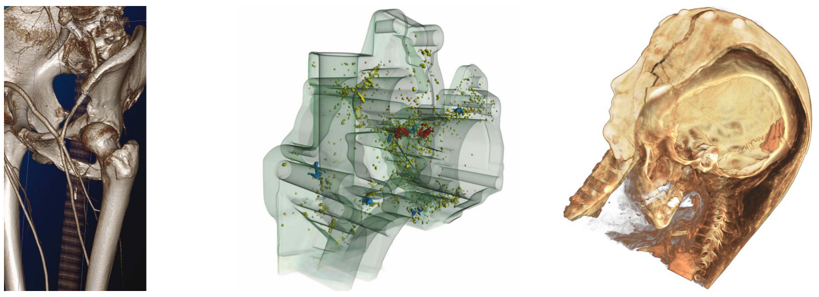

(a) Long leg study (b) Industrial CT (c) Mummy

Figure 1.2: Visualization of different large datasets, (a) a long leg study (courtesy of [Engel et al. 2006]), (b) pores in a cast housing and (c) the head of the before mentioned mummy, courtesy of Stanford Medical School, Volume Graphics

While the raw computing power of todays GPUs can easily handle these large datasets, the traditional bottlenecks of the graphics hardware still persist. The GPU of the first *GeForce* graphics card, introduced in 1999, consists of 22 million transistors. While this number was raised by more than the factor 60, to 1.4 billion on the just released *GeForce GTX 280* chips, the amount of memory and the transfer speeds grew much slower.
In comparison, the first *GeForce* featured 64MB of video RAM. Over the last ten years this number has doubled four times to reach 1GB of memory on the standard *GTX 280* Cards.
During the same time, there was only one major revision made to the graphics port which connects the graphics card to the CPU and main memory. In 2004, the Accelerated Graphics Port (AGP) was replaced by PCI express (PCIe). Between 1999 and 2004 there were only two minor revisions to AGP, increasing theoretical maximum transfer speeds to the GPU from 533MB/s (AGP 2x) to 2,133MB/s (AGP 8x). However, actual transfer rates usually were notably below these specifications, specifically downloading data from the GPU suffered from poor driver implementations. Upon introduction, PCIe featured a maximum of 16 lanes, each 250MB wide. The theoretical transfer speed was 2,000MB/s. Recently PCIe was updated to version 2.0, which features double the bandwidth per lane, totalling 4,000MB/s.
Meaning, if one takes the increase of transistors as a measurement for the increasing raw computing power, over the last ten years the development of computing power outperformed the development of transfer speed and accesible memory by a factor of four.

Compression can be used to handle the increasing amount of data, the comparably slow transfer speed to the graphics board and the small amount of memory. The vast processing power of today's graphics cards can be used for on the fly handling of the compressed data. This way the size of datasets can be decreased, to speed up transfer to the graphics hardware and make it possible to load datasets exceeding its memory capacity. Additional computations to access the compressed data can be handled by the GPU.

1.2 Problem Statement and Objectives

Today, sophisticated volume rendering solutions implement different methods to be able to render datasets, which would not fit into the graphics boards memory otherwise.

Probably the most commonly used approach is the so-called bricking. In a preprocessing step the volume is divided into bricks, which are dynamically loaded onto the graphics card if needed. Depending on the transfer function, or the iso-surface value some of the bricks would not contribute to the final rendering, as the opacity of all containing voxels is zero. Only the bricks that are contributing to the rendering are loaded into the GPUs memory. Only when the transfer function or the iso-surface value changes the bricks on the GPU are updated.
Other common approaches incorporate classical compression techniques, mostly typical image compression adapted to 3D volume data.
The volume processing framework *Cascada* opts for another approach to minimize transfers between the CPU and GPU memory. Image processing operations and visualization are modularly arranged in sequences. For a complete sequence of different operations and simultaneous visualization the volume data has only to be uploaded to the graphics board once. The user can interactively change parameters and the results are visualized directly on screen without the need for uploading new data to the GPU.

All approaches carry along some problems. Bricking can only be used for visualization purposes, as the compact form of the volume is achieved by only loading active blocks onto the graphics board. For volume processing, however, the whole volume is always of importance. Classical compression techniques have the benefit of small datasets, but visualization is quite costly and for some compression types the data has to be completely reconstructed before it can be accessed. In addition, currently there are no efforts to directly process compressed volume data on the GPU.
The sequencing approach used in *Cascada* efficiently reduces transfers from the main memory to the GPU memory and vice versa, however, the effective memory footprint of the volume datasets is not altered. Hence, only datasets which are not exceeding the GPU memory can be processed.

Within the scope of this thesis, *Cascada* has been extended to handle wavelet compressed volume data. Transfer time as well as the memory footprint of the datasets could be decreased. However, processing and visualization algorithms, suited to operate on uncompressed, regular grid volume data can not be used to operate on compressed data. Therefore, adapted processing and visualization algorithms which, if possible, work directly on the compressed data, or at worst include decompression and compression of fractions of the volume, to process a voxel have been developed.

1.3 Structure

This thesis is structured in the following way; In chapter 2, the fundamentals of this work are presented. The volume processing and visualization framework *Cascada*, which is the basis for the implementation of this thesis, will be presented first. The second part focuses on image compression techniques, to build a basis for the choice of the compression technique used in this thesis. In the third part common image processing operations will be presented.

Chapter 3 gives insight to the current state of the art with respect to compression of volume data, focusing on compression techniques adapted from the image compression techniques presented in chapter 2, as well as regarding processing of compressed images. It concludes with a summary and discussion, which shows the advantages and shortcomings of the different approaches and constitutes the choice of volumetric wavelet compression for this thesis.

The following chapters 4 and 5 form the main part of this work. Chapter 4 presents the Haar wavelet transform, that has been implemented in *Cascada* alongside the used data structure and closes with the documentation of the implementation details. In chapter 5 the focus is on the processing and rendering algorithms for wavelet compressed volume data. The chapter starts with mathematical considerations for different processing operations and concludes with the implementation details. This includes the data structure, as it is used on the GPU as well as the implemented visualization and processing algorithms for CPU and GPU.

The results of this work are presented in chapter 6. The performance of the implemented algorithms are discussed. Visual quality as well as processing time are compared to counterpart algorithms working on non-compressed data, which are part of *Cascada*.

Finally, chapter 7 concludes this thesis by giving a summary of the results and presenting an outlook into future work.

Chapter 2

Fundamentals

In the following chapter the fundamentals of this work are presented. In section 2.1 a short survey of the volume processing and rendering framework *Cascada* is given, which will serve as the basis for implementation of this thesis. To be able to relate to the choice of wavelet compression, section 2.2 gives an overview of different image compression techniques, including wavelet, cosine and fractal compression as well as vector quantization. This chapter concludes with section 2.3, which presents some of the most common image processing operations and classifies them according to their application in the spatial domain.

2.1 Cascada

Cascada is a cross platform volume processing and visualization framework developed at the University of Koblenz over the past few years. Its main focus is on object oriented modular implementation of volume processing algorithms on graphics hardware [Raspe et al. 2008b]. The main idea behind *Cascada* is to encapsule different processing and visualization modules in so called sequences (compare figure 2.1). These modules, named render passes, range from simple thresholding operations to more complex operations like region growing. A sequence can consist of multiple passes, or even sequences, which each can be executed once or multiple times. The result of one pass is then set to be used as the input of the next pass, by adjusting texture parameters. As denoted in section 1.1 the transfer of the data to and from the graphics hardware is a limiting factor of todays GPUs. *Cascada* tries to minimize these transfers with its modular design. Multiple operations can be applied to a volume dataset, meanwhile the user can change parameters for these operations and view the results live on screen, without the need for transferring the dataset from GPU

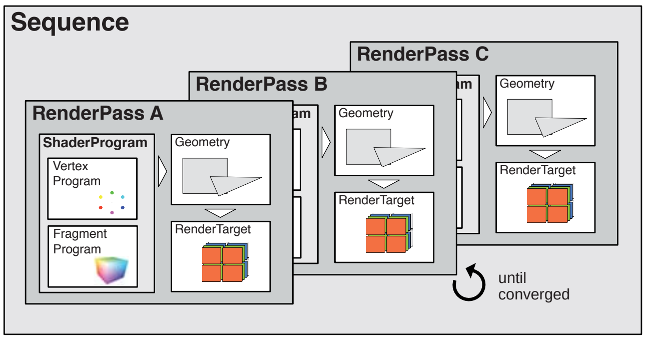


Figure 2.1: Hierarchy of rendering components in *Cascada* , courtesy of [Raspe & Müller 2007]

into CPU memory a single time.
In [Raspe & Müller 2007] some insight in the performance relation between GPU based operations in *Cascada* and CPU implementation is given. While the GPU outperforms the CPU easily when measuring the time needed for computation solely, when including the time needed to transfer the data between CPU and GPU memory especially simple algorithms like a binary thresholding operator are notably slower on the GPU. In [Raspe et al. 2008a] additionally, performance of more advanced sequences consisting of reasonable concatenations of varying operations are compared to the same operations implemented in software. A notable speedup could be shown by eliminating the costly data transfer to the GPU these operations. Thus making the GPU a well suited tool in the field of computation intensive medical image processing.

While the main focus in *Cascada* is on medical volume processing, one of the main advantages of processing on the GPU is that visualization is virtually free, with the data already on the graphics board. Thus, *Cascada* features direct volume rendering via raycasting and maximum intensity projection alongside multi-planar reconstruction. Additionally, one or two dimensional transfer functions can be used for feature classification.

For comfortable interaction alternative input devices are supported by *Cascada*. In addition to keyboard and mouse 3dconnexions 3D mouse *SpaceNavigator* as well as *Phantom* haptic input devices made by SensAble can be used to navigate in the volume.

At the moment version 2.0 of *Cascada* is in development. It is implemented from scratch and when finished will feature a much more modular structure than the current version 1.0. In addition to the current OpenGL based GPU programming model modern APIs like nVidias CUDA will be available. *Cascada* 2.0 will be extensible via a flexible plugin interface, for example to support alternative input devices, just like the *SpaceNavigator*. *Cascada* 2.0 will remain plattform independent (running on Windows, Linux and Mac OSX), CMake will be used to create project files for the different systems. During implementation a much higher level of quality control will be kept, for example, with consisten unit testing.

2.2 Image Compression Techniques

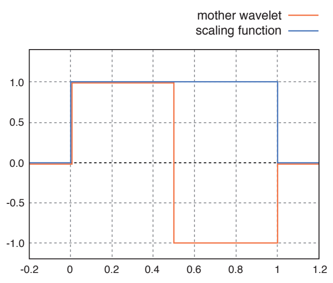
In the early stage of this thesis the choice of compression techniques was narrowed down to the four, which will be presented in the following. To constitute the decision for the wavelet compression a short overview of wavelet (sec. 2.2.1), cosine transform based (sec. 2.2.2) and fractal compression (sec. 2.2.3) as well as vector quantization (sec. 2.2.4) will be given in the following sections. For the sake of simplicity the two dimensional image based versions of these techniques are presented. It is assumed that the characteristics of these techniques do not fundamentally change in higher dimensions. For details of existing three dimensional data compression techniques and the adaptions needed to port the presented 2D modes refer to section 3.1.

2.2.1 Wavelet Image Compression

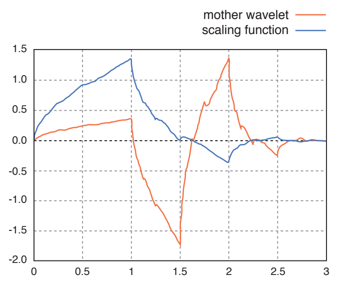
Wavelet compression is a transform coding technique which is used in the JPEG 2000 Standard [JPEG 2000] and consists of two passes. The first pass, the actual wavelet transform and the second pass is the quantization or compression step. After the transform the wavelet domain representation of the image consists of as much coefficients as the image of pixels. Transform coding techniques are used because quantization applied in the original image domain (or spatial domain) often result in poor quality. The same quantization technique applied to a properly transformed image exploiting certain key characteristics of the transformation domain can yield remarkable results.

The basis functions for the wavelet transform are translations and dilations. These functions are based on the function ψ, referred to as the mother wavelet. The one dimensional discrete wavelet transform of a function $f(x)$, with the discrete scale and translation step size a and b, and the discrete scale and translation variable m and n is defined by

$$\mathcal{W}_{\psi}^{m,n}(f) = |a_0|^{-\frac{m}{2}} \int f(x)\psi(a_0^{-m}x - nb_0)dx. \tag{2.1}$$



(a) Haar wavelet



(b) Daubechies wavelet

Figure 2.2: The Haar mother wavelet and the corresponding scaling function (a) and the Daubechies D4 mother wavelet with its scaling function (b)

Typical examples for mother wavelets are the very simple Haar wavelet

$$\psi(x) = \begin{cases} 1 & \text{for } 0 \leq x < 1/2 \\ -1 & \text{for } 1/2 \leq x < t \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

with its scaling function

$$\phi(x) = \begin{cases} 1 & \text{for } 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

or the Daubechies wavelet

$$\psi(x) = \sum_{k=0}^{N-1} a_k \phi(2x - k) \tag{2.4}$$

and the corresponding scaling function

$$\phi(x) = \sum_{k=0}^{M-1} b_k \phi(2x - k), \tag{2.5}$$

where $(a_0, \ldots, a_{N-1})$ and $(b_0, \ldots, b_{M-1})$ are appropriate finite sequences of real numbers.

The wavelet transform can be interpreted as a low and high pass transformation. As such, the resulting coefficients can be represented as a low and a high band. The low band coefficients usually consist of high energy and have no advantage over non transformed voxels regarding compression. Thus in most cases the transform is applied recursively on the low band. The number of recursions is called the level of transformation. For two dimensional images the 1D wavelet transform is applied line-wise, followed by the same transformation applied row-wise to the result. This procedure yields four subbands at each level: one is pass filtered twice and three are high pass filtered at least once. The representation (at level one) can be seen in figures 2.5(b) and 2.5(e). The low band is on the top left, the high bands on the right and/or bottom, corresponding to the direction the high pass filter was applied. The two dimensional transform can be applied in two ways, called the standard and non-standard transform. For the standard transform, first the rows are transformed till the desired level is reached, then the multi level 1D transform is applied to the columns. For the non standard transform, the 1D transforms are applied alternately at each level. Even though this representation yields no compression thus far (as many coefficients have to be stored as original pixel values), it often has certain advantages over the original image representation. Neighbouring pixels in natural images tend to have very similar or even the same values, meaning that the high pass filtered coefficients are often zero or at least very small (compare the histograms of single level wavelet decompositions in figures 2.5(c) and 2.5(f)). These characteristics of the coefficients can be used in the second pass to compress the image.

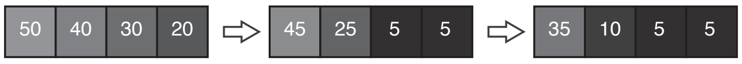


Figure 2.3: A simple example of a Haar wavelet decomposition. An image and its first and second level wavelet decompositions

Compression can then be done with a couple of more or less sophisticated methods. Commonly only the high passed filtered coefficients are compressed, while the low pass transformed remain unchanged. Thus, in the following when speaking of coefficients the high pass filter coefficients are meant.
The basic method is quantization, meaning using fewer bits to store the coefficients than the original image. As the coefficients are usually very small one can delimit the range to a smaller area around zero and use a smaller datatype to store the coefficients. The naive approach, to just cut off the bits of higher value might seem to be a good solution, as there are very few coefficients in the upper section of the range, but as these coefficients carry crucial detail information a more sophisticated approach is needed. Another possibility would be dividing the coefficients by a given divisor and only storing the integer value of the results. Quantization usually results in a loss of visual quality.
A second approach, the Wavelet Zerotree Encoding [Rogers 1998] is also a lossy compression scheme. The individual recursion steps in the wavelet transform can be interpreted as levels in a tree in which the residing low-band coef-

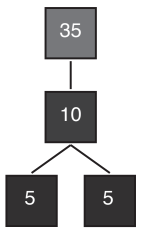


Figure 2.4: The image from figure 2.3 after two-level wavelet encoding in zerotree representation

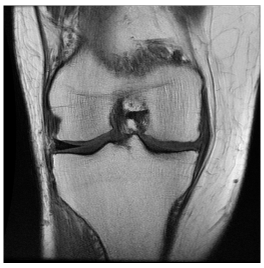
ficients after the last recursion step resemble the root node (in the above mentioned example $a_{0,0}$) and the coefficients after the first recursion step are the leafs. Each node, representing one coefficient of a decoded n-dimensional image has 2^n children. Typically there is a great similarity between a node and its children. In particular it is often the case that if a coefficient is zero its children will also be zero. This is used in the zerotree to cut off a branch once a coefficient is zero.

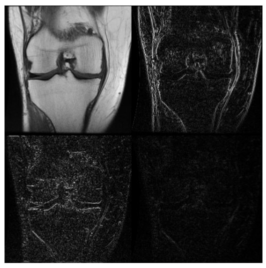
A third approach saves only the non-zero entries. However, as information of the position in the image is indirectly stored in the position in the array, which is lost when discarding the zero-elements, one has to store the position with every item taken from the array. This however, is only feasible when there is a large amount of zeroes amongst the coefficients, because every coefficient that needs to be stored (all non-zero coefficients) needs additional memory to store the position (i.e. $2 \cdot 16$bit for an image larger than 256x256 pixels).

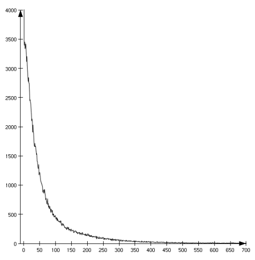
A modification to the last approach uses a binary significance map, which corresponds in size to the original image. At every position where the wavelet transformed original image is holding a non zero coefficient the importance map holds 1. Accordingly, the importance map is 0 on all positions where there is a zero coefficient in the transformed image. With this map the non zero coefficients can be stored sequentially in a queue without position information. To rearrange the coefficients for decoding one goes over the importance map the same way as over the encoded image, when building the queue and pops the first element out of the queue at every 1 in the importance map and puts it in the corresponding position in the target image.

The last two methods are lossless by default, but compression rate can be further enhanced for all four methods by setting a certain amount of small non zero coefficients to zero. This is a lossy process and thus leads to worse visual quality. But as many of the coefficients are very close to zero this often largely increases the compression rates.

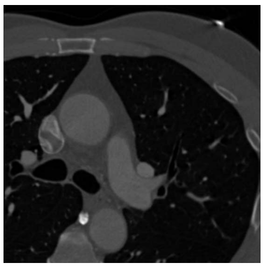
Decompression is straightforward. First one has to look up the coefficients belonging to the current block. Regarding to the compression technique used this

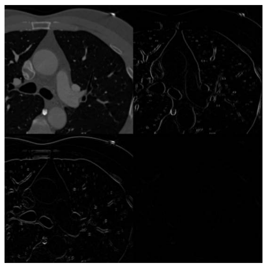

(a) The original image. Courtesy of [OsiriX]

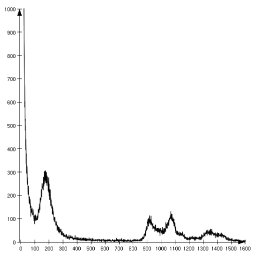

(b) Wavelet decomposition



(c) The histogram


(d) The original image. Courtesy of [OsiriX]


(e) Wavelet decomposition



(f) The histogram

Figure 2.5: A slice from an MRI-image of a knee (a), respectively from a cardiac CT scan (d), the single level 2D wavelet decomposition of the images, the coefficients have been brightened better visualization ((b) and (e)), and the histogram of the wavelet decompositions, clamped to the interesting regions ((c) and (f))

can be a more or less complex step. After that the reconstruction can be done according to the chosen mother wavelet. If the number of levels used is larger than one these steps have to be done iteratively from the last to the first level.

2.2.2 Discrete Cosine Transform

Just like the wavelet transform, the discrete cosine transform (DCT) is a transform coding technique which is used for image compression. In fact the DCT is a widely used image coding technique due to its use in the well known JPEG compression [JPEG], which is the de facto standard for compression of photographic images today.

A brief description of the DCT used in the JPEG compression standard can be found in [Wallace 1991]. The DCT-based encoder works in three steps. First a forward discrete cosine transform (FDCT) is applied to the image, then the resulting cosine coefficients are quantized and finally run through an entropy encoder.

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	157	158

(a) source image i

235.6	-1.0	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0.0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1.0	-1.0
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

(b) DCT coefficients i_{DCT}

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	50
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	98
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(c) quantization table q

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(d) normalized quantized coefficients i_q

240	0	-10	0	0	0	0	0
-24	-12	0	0	0	0	0	0
-14	-13	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(e) denormalized quantized coefficients

144	146	149	152	154	156	156	156
147	150	152	154	156	156	156	156
155	156	157	158	158	157	156	155
160	161	161	162	161	159	157	155
163	163	164	163	162	160	158	156
163	164	164	164	162	160	158	157
160	161	162	162	162	161	159	158
158	159	161	161	162	161	159	158

(f) reconstructed image i'

15 0 -2 -1 -1 -1 0 0 -1 0 0 0 0 0 0 ...

(g) *Zig-zagged*, normalized, quantized coefficients

Figure 2.6: DCT and Quantization Examples[Wallace 1991]

This process is not done on the image as a whole but on 8x8 blocks. For color images each channel is transformed separately. Therefore the image usually is transformed into YCrCb (luminance-chrominance) color space.
The decompression incorporates inverse versions of all three steps in inverse order. First the compressed image data is entropy decoded, then dequantized and finally inverse discrete cosine transform (IDCT) is applied. A complete transform including its reversal can be seen in figure 2.6.

Mathematically FDCT (equ. 2.6) and IDCT (equ. 2.7) on 8x8 sample blocks can be described as follows:

$$i_{\text{DCT}}(u,v) = \frac{1}{4}C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} i(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \tag{2.6}$$

$$i(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)i_{\text{DCT}}(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \tag{2.7}$$

with

$$C(u),C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u,v=0 \\ 1 & \text{otherwise} \end{cases} \tag{2.8}$$

and i_{DCT} the image in the DCT domain and i the original image in the spatial domain.
Equation 2.6 is designed to operate on input data, $i(x, y)$ centered around 0 rather than the usual image range from 0 to 2^{bits}, so 2^{bits-1} has to be subtracted from each image value $i(x, y)$. Then each coefficient $i_{\text{DCT}}(u, v)$ can be computed with the given equation. The computed coefficients from an exemplary source image i (figure 2.6(a)) can be seen in figure 2.6(b). The top left coefficient $i_{\text{DCT}}(0, 0)$ represents the lowest frequency in the original image, while frequencies get larger to the lower right corner.

The FDCT is followed by quantization. The 8x8 coefficient matrix is divided component-wise by a quantization table, like the one shown in figure 2.6(c) and the resulting values are converted to integer values (see equation 2.9). The table is chosen according to the desired compression ratio. A set of quantization tables optimized for human perception can be found in [CCIR]. As the human eye is more sensitive to lower frequencies the quantization tables represent this by containing larger quotients in the lower right area and smaller ones in the upper left region.

$$i_q(u,v) = \begin{cases} \lfloor i_{\text{DCT}}(u,v)/q(u,v) \rfloor & \text{for } \frac{i_{\text{DCT}}(u,v)}{q(u,v)} >= 0 \\ \lceil i_{\text{DCT}}(u,v)/q(u,v) \rceil & \text{for } \frac{i_{\text{DCT}}(u,v)}{q(u,v)} < 0 \end{cases} \tag{2.9}$$

After the quantization the normalized quantized coefficients (figure 2.6(d)) consist of a lot of zeroes which are usually divided from the nonzero coefficients by a diagonal line. This is used for the so called *zig-zagging*. Instead of coding the block line- or rowwise the coefficients are rearranged in a *zig-zag* line from the upper left to the lower right (compare figure 2.7).
Now the low frequency coefficients, which are more likely to be non zero are placed before the higher frequency coefficients which helps to make the coding more efficent. The resulting sequences then get compressed with a Huffman coding [Knuth 1985].

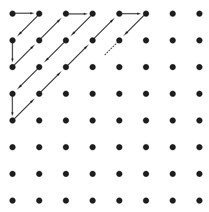


Figure 2.7: *Zig-zagging*

2.2.3 Fractal Compression

Fractal image compression was introduced in [Barnsley & Hurd 1993] as a vector based technique. It is a lossy compression technique, which compresses an image by storing it as a set of transformations. Ideally an image can be stored as only one transformation function. The unique fixpoint of this function then must be (a close approximation of) the image itself.In order to to store such a function one usually needs only a fraction of the storage amount needed to store the original image. Decompression is then done by iterative application of this function on an arbitrary starting image until the original image is received.

This concept can be illustrated very easily with one of the simplest fractals, the Sierpinski triangle (figure 2.8). The Sierpinski triangle is self similar, as one can copy scaled versions of the triangle into its corners and receive the same image. In other words, if ω_1 is the transformation that maps the triangle in its upper corner and respectively ω_2 and ω_3 map the triangle to the lower corners, the Sierpinski triangle can be described as the fixed point of the transformation

$$\omega = \omega_1 \cup \omega_2 \cup \omega_3 \tag{2.10}$$

as the transformation applied to the triangle leaves it unchanged. The transformation ω is contractive, meaning the distance between two points p and q

$$d(p,q) = (p_x - q_x)^2 + (p_y - q_y)^2 \tag{2.11}$$

after a transformation is never larger than before the transformation, because all of the transformations ω_i are contractive. With the Sierpinski triangle being the fixpoint to the transformation ω and ω being contractive according to the *Banach Contractive Fixed-Point Theorem* [Valero 2005], the Sierpinski triangle is

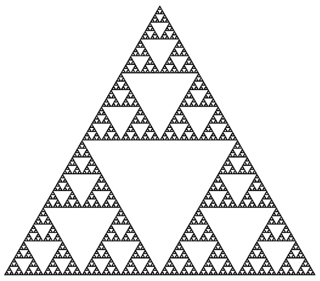


Figure 2.8: The Sierpinski Triangle

a limit of the sequence

$$X, \omega X, \omega^2 X, ... \tag{2.12}$$

with X representing an image and ω^i the composition of ω with itself i-times meaning by iteratively applying ω on any image one will at some point receive the Sierpinski triangle with any rate of accuracy (compare figure 2.9). Thus, one can say that the transformation ω alone can be used as a representation for the image of the Sierpinski triangle. The representation is also known as an Iterated Function System (IFS).

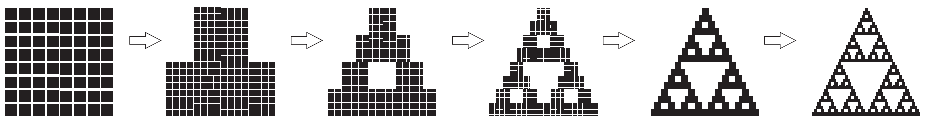


Figure 2.9: Multiple iterations from a random image to the Sierpinski Triangle

Such a compact form as in equation 2.10 can only be achieved for fractals like the Sierpinski triangle but many images like photographs do have areas of self similarity. Therefore, Partitioned Iterated Function System (PIFS) based on IFS can be applied to real world images.
The image is partitioned in a grid commonly known as range blocks. These range blocks are all the same size and non-overlapping. For each of these blocks a region in the image has to be found which can be transformed with a set of contractive functions to match the original range block as close as possible. In a typical application the rangeblocks are matched to a fixed set of domain blocks in the image. These blocks are usually double the size as the range blocks and are not arranged in a fixed grid, i.e. they can overlap. The transformations applied to the domain blocks consist of scaling, translating, rotating and reflecting, as well as adjustments in brightness and contrast. As only the transformations are stored and not the domain blocks, the codebook formed by these blocks is called virtual.

To decompress the image a new image with the size of the original image is created. The functions of the PIFS are then applied iteratively to the starting image until the changes in the image between two iterations are smaller than a defined threshold. Due to the nature of fixed points the starting image can be random, however, the time to compose the final image or the number of iterations may vary depending on the starting image, yet the image quality will be the same.

2.2.4 Vector Quantization

Just like fractal image compression vector quantization [Gersho & Gray 1992] is a vector based image compression technique. By nature the vector quantization is a lossy compression scheme as its main concept is to approximate a range of values by a single value. An example for a one-dimensional vector quantization is rounding to the nearest integer: all values in the range $]x - 0.5 \ldots x + 0.5]$ are mapped to the single value x.

An example for a two dimensional vector quantization can be seen in figure 2.10. Each of the stars, called codevectors represents all vectors lying in the surrounding region, the so-called encoding regions. All codevectors together form the codebook and all encoding regions form the partitioning.

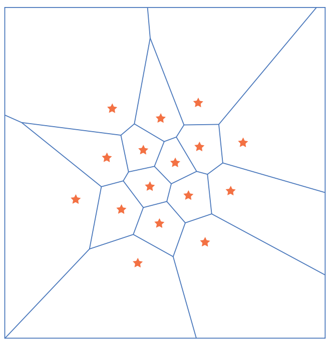


Figure 2.10: Partitioning for a two-dimensional vector quantization

For a set of input vectors $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\}$ and a given codebook $C = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_N\}$ the optimal partitioning, meaning the partitioning with the smallest error is the set of all regions, where every region S_n contains all input vectors which are closer to the enclosed codevector than to any other codevector (equation 2.13).

$$S_n = \{x \mid \|x - c_n\|^2 \leq \|x - c_{n'}\|^2 \ \forall n' = 1, 2, \ldots, N\} \tag{2.13}$$

These regions are also called Voronoi regions and the whole partitioning builds a Voronoi diagram.

In image compression vector quantization is used in the following way: The input image is partitioned into a regular grid of non-overlapping blocks, just as is done for fractal compression. These blocks form the set of input vectors. Then the input vectors are matched with the codebook as described above. The codebook can be a set of vectors from the input image, similar to the virtual codebook in fractal compression or a global set of vectors defined independently from the input image. Contrary to fractal compression, input vectors and codevectors need to be of the same size. When the closest match is found, rather than the block of pixels a pointer to the vector in the codebook is stored.

In contrast to the fractal compression the codebook needs to be stored to lookup the blocks in the decompression step in addition to a lookup map, which is the vector quantization equivalent to the PIFS. Thus the compression ratio mostly depends on the size of the codebook (the size of the lookup map is defined by the size of the input image, the size of the input partitioning and the size of the pointers to the codebook entries), meaning the crucial part in this procedure is to find a codebook which is small enough to yield the desired compression ratio, yet big enough to map all input vectors with minimal error.

The decompression is a simple inversion of this process. For all entries in the lookup table the corresponding codebook vectors are fetched and placed in the appropriate position in the target image (which is defined by the position of the pointer in the lookup table).

Figure 2.11 shows an example of a vector quantization. The input image on the left is partitioned in 1x2 blocks which are the input vectors. These are matched (red arrows) against the codebook (in the middle) and instead of the input vectors the address in the codebook (green arrows) is stored in the lookup table (bottom), which has the extents of the input image scaled by the vector size. The $(20, 30)^T$ codevector is the nearest match for the $(20, 25)^T$ input vector, so they get matched. For decompression (blue arrows) codevectors are placed in the target image at the appropriate positions given by the lookup table.

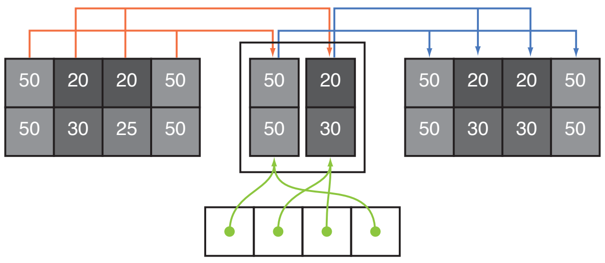


Figure 2.11: Vector quantization example

2.3 A Classification of Image Processing Operations

Lehmann et al. [1997] break image processing operations down to three classes: point based operations, local operation and arbitrary or global operations. This partitioning is the basis for the classification presented in this section. The three classes are further subdivided and common image processing operations are presented and assigned to the classes. For the sake of simplicity, just as it was the case in the previous chapter the operations are presented for two dimensional images only. It is assumed that the classification will still be valid for 3D operations.

2.3.1 Point Based Operations

Point based image operations are operations based on the current pixel only. No neighbouring data is taken into account. They can be interpreted as a special case of local operations (section 2.3.2) with a 1×1 neighborhood. Universally those operations can be described by

$$f^{\star}(x,y) = T_{x,y}\left(f(x,y)\right) \tag{2.14}$$

with $f(x,y)$ the gray value at position (x,y) in the input image and $f^{\star}(x,y)$ the result for the corresponding position in the target image. The index $_{x,y}$ on the transformation T denotes dependance on the position in the image which is non-mandatory. If the transformation is position dependent it is called inhomogeneous, (homogeneous otherwise) and can be written without the index

$$f^{\star}(x,y) = T\left(f(x,y)\right). \tag{2.15}$$

Homogeneous Operations

Homogeneous point operations can easily be calculated by a gray value characteristic. For every input gray value $g \in \mathbf{G}$ the characteristic provides the target gray value $g^{\star}$:

$$g^{\star} \in \mathbf{G} = \{0, 1, \ldots G-1\} : g^{\star} = T(g) \tag{2.16}$$

For discrete gray values the characteristic can be replaced by a simple look up table.

Inversion

Inversion maps the maximum value of the input to zero and the minimum value to the original maximum value minus the minimum value. Values in between the minimum and maximum value are mapped linearly between these two. Every inverted value can be computed by a single difference operation.

$$T(g) = g_{\text{max}} - g \tag{2.17}$$

The only value that must be known besides the input gray value g is the maximum gray value of the dataset g_{max}. The inversion is reversible without any data loss by repeated appliance.

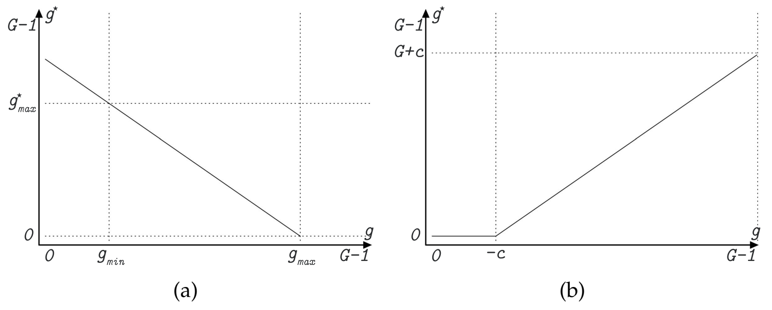


Figure 2.12: The characteristic for the inversion operator (a) and for the histogram shift operator with a negative c (b)

Histogram Shift

The histogram shift is a simple method to adjust the brightness of an image. By adding a constant value to all gray values in the image the image is brightened or darkened. The histogram shift can be computed with

$$T(g) = \text{clamp}(0,\, G-1,\, g+c) \tag{2.18}$$

where

$$\text{clamp}(a,b,x) = \begin{cases} a & \text{for } x < a \\ b & \text{for } x > b \\ x & \text{otherwise} \end{cases} \tag{2.19}$$

due to the clamp-operator histogram shift is not reversible except for the special case when $g + c$ is in between 0 and $G - 1$ for all pixels in the image.

Histogram Spread

Histogram spreading is a more advanced brightness regulation technique. It increases the contrast by mapping the actual minimum and maximum gray values (g_{min} & g_{max}) of the original image to new boundaries, usually the boundaries of the target image range. The values in between are mapped linearly.

$$T(g) = (G-1) \cdot \frac{g - g_{\text{min}}}{g_{\text{max}} - g_{\text{min}}} \tag{2.20}$$

Histogram spreading is a useful tool for converting images to different datatypes (i.e. converting a CT image with 12 bit depth to an 8 bit gray image). The histogram spread is reversible by applying histogram compression but due to rounding errors only in the continuous grey space without error.

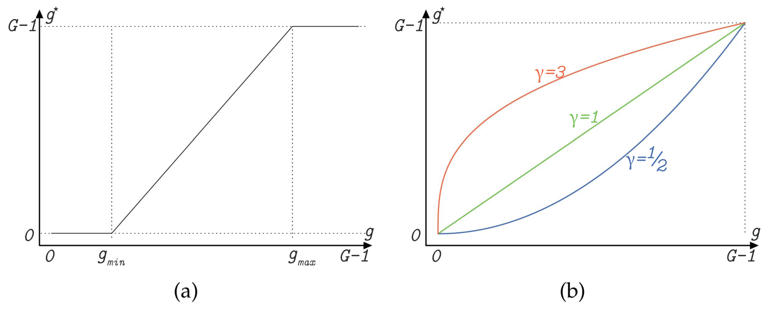


Figure 2.13: The characteristic for the histogram spread operator (a) and for the γ-correction with different γ–values (b)

Gamma Correction

The gamma correction is used to yield the same luminosity perception for an image on displays with different luminance settings. The gamma correction results in an overall darkening or brightening of the input image while preserving the brightest $(G-1)$ and darkest value (0). With the exponent $\gamma < 1$ the result is brighter than the input, with $\gamma > 1$ it is darker. If $\gamma = 1$ the image is unmodified (see figure 2.13(b) for characteristics for different γ-values).

$$T(g) = (G-1) \cdot \left(\frac{g}{G-1}\right)^{\gamma} \tag{2.21}$$

The gamma correction can be reversed exactly only in the continuous gray space by using $\frac{1}{\gamma}$ instead of γ. For the discrete case due to rounding errors the reverse can only be approximated.

Thresholding

Thresholding is a very basic segmentation algorithm based solely on gray values. One or more thresholds can be used to partition the image into regions of similar gray values. The simplest form is binary thresholding with only one threshold t. If the input gray values is smaller than t it is set to zero, else to the maximum value of the gray range.

$$T(g) = \begin{cases} 0 & \text{for } x < t \\ G-1 & \text{for } x \geq t \end{cases} \tag{2.22}$$

Multiple thresholds can be used to partition the image into more different regions. Thresholding is not reversible.

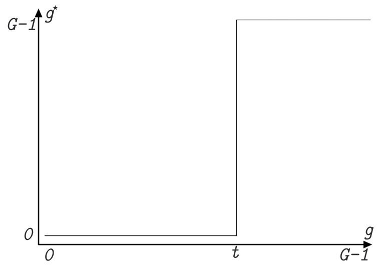


Figure 2.14: The characteristic for a thresholding operator with threshold t

Inhomogeneous Operations

All of the above presented operations are homogeneous point based. Inhomogeneous operations are used rather infrequently compared to homogeneous. Since the result of such an operation is not solely dependent on the gray value they cannot be written as functions of g. Thus, these operations also cannot be represented by a gray value characteristic.

Difference Image Operator

The difference image operator uses two images (f_0 and f_1) as input and delivers the absolute difference between the gray values of the two pixels with the same position in the input images as result.

$$f^{\star}(x,y) = |f_0(x,y) - f_1(x,y)| \tag{2.23}$$

The operation is often used to compare an image after the appliance of an operation with the original image, for example to measure the error of a lossy compression. The difference image operator is not reversible due to the use of the *absolute*-operation.

Correction of Inhomogeneous Illumination

Inhomogeneous illumination can be caused by several reasons when acquiring an image. Many imponderabilities have influence on the acquired image. For example dust on the lens or unbalanced sensitivity of the sensor can cause a bias. If the bias is known, or can be measured by acquiring a neutral image with the same recording device, it can be corrected by point wise division of the input image f by the bias image f_b.

$$f^{\star}(x,y) = (G-1) \cdot \frac{f(x,y)}{f_b(x,y)} \tag{2.24}$$

In the discrete gray space the reverse can only be approximated, in the continuous the operation is fully reversible.

2.3.2 Local Operations

Point based operations as described in section 2.3.1 can only be used to describe a very small set of image operations. A more general approach is to use information of the neighborhood of a pixel to compute the transformation of the pixel. One can comprise only the direct neighbors or the complete neighborhood of the pixel. Usually a neighborhood of 3×3 or 5×5 pixels is considered for such an operation. Neighborhood based operations are not limited to two dimensional images and neighborhoods. Typically n-dimensional data is processed on the basis of an x^n neighborhood. Local operations are also referred to as filtering.

Local operations can be divided in linear and non-linear operations. For linear operations a kernel or mask is created which assigns a weight, $h(i,j)$ to every pixel of the desired neighborhood by the following scheme:

$$\boxed{\begin{array}{ccc} h(-1,-1) & h(0,-1) & h(1,-1) \\ h(-1,\ 0) & h(0,\ 0) & h(1,\ 0) \\ h(-1,\ 1) & h(0,\ 1) & h(1,\ 1) \end{array}}$$

Hereby the pivot point, which usually is the center of the mask has the coordinates $(i, j) = (0, 0)$. The application of such a mask $\mathbf{M}$ on an image f is called convolution (denoted $f * h$) and can (without normalization) be described as follows:

$$f^{\star}(x, y) = \sum_{(i,j)\in\mathbf{M}} f(x - i, y - j) \cdot h(i, j) \tag{2.25}$$

Whether the result of the convolution is normalized depends on the different masks. Usually if the sum of all items in the kernel is equal to zero no normalization is needed else the result gets divided by this sum.
Linear filters can be further subdivided in separable and non separable filters. Separable means an n-dimensional filter of edge length k can be applied by consecutively applying n one-dimensional filters with k elements. If implemented as sequence of one dimensional filters separable filters can be computed much faster as non separable filters, as only $2k$ operations instead on k^2 operations are needed. Generally filter masks with the rank 1 are separable, mask with a rank $\neq 1$ are not.

Box Filter

The box filter is used for noise reduction by inscribing the arithmetic mean of the neighborhood to the current pixel. The box filter can be written as a 3×3 mask

$$\mathbf{M} = \frac{1}{9} \cdot \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \tag{2.26}$$

While single points with extreme gray values are flattened the disadvantage of the box filter is the fact that it does *smear* the image. It smoothes edges and small periodically repeating edges could be lost completely.
The box filter is seperable and can be computed with the following two filter masks:

$$\mathbf{M_1} = \frac{1}{3} \cdot \begin{matrix} 1 & 1 & 1 \end{matrix} \qquad \text{and} \qquad \mathbf{M_2} = \frac{1}{3} \cdot \begin{matrix} 1 \\ 1 \\ 1 \end{matrix} \tag{2.27}$$

Gaussian

The Gaussian filter is the discrete version of the probability density function of the normal distribution (normalized, with mean $\mu = 0$ and the variance $\sigma^2 = 1$)

$$f(x) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{x^2}{2}}. \tag{2.28}$$

The Gaussian filter is being used for noise reduction and image smoothing. It is not as strong as the box filter but still capable of smoothing extreme gray value changes. As a 3×3 mask the Gaussian filter looks as follows:

$$\mathbf{M} = \frac{1}{16} \cdot \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix} \tag{2.29}$$

The Gaussian, just like the box filter, is seperable. It can be computed by two one-dimensional Gaussian filters:

$$\mathbf{M_1} = \frac{1}{4} \cdot \begin{matrix} 1 & 2 & 1 \end{matrix} \qquad \text{and} \qquad \mathbf{M_2} = \frac{1}{4} \cdot \begin{matrix} 1 \\ 2 \\ 1 \end{matrix} \tag{2.30}$$

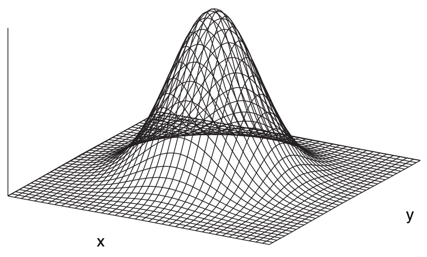


Figure 2.15: 2D Gaussian filter kernel

Differential Edge Detection

Differential edge detection is used to emphasize edges in an image. In the continuous an edge is defined by local extrema in the differential quotient of the input signal. The edge direction is then perpendicular to the direction the input signal was differentiated in. In the discrete this can be approximated by differencing, however, the smallest distance between two points is 1.

$$f^{\star}(x, y) = f(x, y) - f(x - 1, y) \tag{2.31}$$

and

$$f^{\star}(x, y) = f(x, y) - f(x, y - 1) \tag{2.32}$$

emulate the partial derivative in x- (equ. 2.31) respectively y-direction (equ. 2.32). The corresponding filter masks look as

$$\mathbf{M}_x = \begin{matrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{matrix} \quad \text{respectively} \quad \mathbf{M}_y = \begin{matrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}. \tag{2.33}$$

These edges in the resulting image have to be interpreted as shifted by half a pixel in the particular direction. Alternatively symmetric versions of the kernels are used:

$$\mathbf{M}_x = \begin{matrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{matrix} \quad \text{respectively} \quad \mathbf{M}_y = \begin{matrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{matrix}. \tag{2.34}$$

All the four kernels are merely one dimensional vectors in matrix notation, so they are all separable to the used 1D vector and the vector $(0, 1, 0)$ respectively $(0, 1, 0)^T$. However, only using the original 1D vector is more efficient. These operations, besides detecting edges also amplify differences caused by noise. The following edge detection operators (Prewitt and Sobel) include noise suppression to minimize this problem.

Prewitt Operator

The Prewitt operator is a combination of the symmetric difference operator for edge detection and an averaging over the neighboring pixels of the pivot point in the supposed edge direction for noise reduction. It can be represented for the x- and y-direction by the following masks:

$$\begin{aligned} \mathbf{M}_x &= \begin{matrix} -1 & 0 & 1 \end{matrix} * \begin{matrix} 1 \\ 1 \\ 1 \end{matrix} = \begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix} \\ &\text{and} \\ \mathbf{M}_y &= \begin{matrix} -1 \\ 0 \\ 1 \end{matrix} * \begin{matrix} 1 & 1 & 1 \end{matrix} = \begin{matrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{matrix} \end{aligned} \tag{2.35}$$

It is obvious that the two kernels are separable, as they both are derived from two one dimensional filter masks.
Edges in both directions can be combined in a single result image by applying

both kernels to the input image and adding the absolute results.

$$f^\star(x,y) = \sqrt{(\mathbf{M}_x * f(x,y))^2 + (\mathbf{M}_y * f(x,y))^2} \tag{2.36}$$

Sobel Operator

The Sobel operator is very similar to the Prewitt operator, however, instead of averaging a one dimensional Gaussian is used to suppress noise:

$$\begin{aligned} \mathbf{M}_x &= \begin{matrix} -1 & 0 & 1 \end{matrix} * \begin{matrix} 1 \\ 2 \\ 1 \end{matrix} = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \\ \text{and} & \\ \mathbf{M}_y &= \begin{matrix} -1 \\ 0 \\ 1 \end{matrix} * \begin{matrix} 1 & 2 & 1 \end{matrix} = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix} \end{aligned} \tag{2.37}$$

The Sobel operator can also be used for diagonal directions:

$$\mathbf{M}_/ = \begin{matrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{matrix} \quad \text{and} \quad \mathbf{M}_\backslash = \begin{matrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{matrix}. \tag{2.38}$$

A direction independent edge image can be received analogous to the Prewitt operator, as shown in equation 2.36. Just as the Prewitt operator the orthogonal versions M_x and M_y are separable, however, the diagonal versions are not, as their rank is three.

Laplace Operator

The aforementioned Prewitt and Sobel operators are referred to as first order edge detection operators. The Laplace operator is a second order edge detector. Meaning, the Laplace operator is based on the the second partial derivate of the input signal. The continuous version is defined by the sum of the second order partial derivates:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.39}$$

The discrete pendant is obtained by the sum of iterations of the first order derivates from equations 2.31

$$\begin{aligned} f^\star(x,y) &= (f(x+1,y) - f(x,y)) - (f(x,y) - f(x-1,y)) \\ &= f(x+1,y) - 2f(x,y) + f(x-1,y), \end{aligned} \tag{2.40}$$

for x-direction and 2.32

$$f^{\star}(x,y) = f(x,y+1) - 2f(x,y) + f(x,y-1), \tag{2.41}$$

for y-direction. The resulting kernel the looks like

$$\mathbf{M} = \begin{array}{ccc} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{array}. \tag{2.42}$$

The Laplace operator is rotation independent, thus emphasizes edges in all directions. Homogenous areas and constant gray ramps are ignored. As the operator does not incorporate noise suppression, like the Prewitt and Sobel operator it emphasizes noise alongside edges. Thus, often a Gaussian operator is used before the Laplace. The combination is referred to as Laplacian of Gaussian (LoG) or due to the form of the combined kernels mexican hat operator. The rank of the Laplace operator is 2, hence it is not separable.

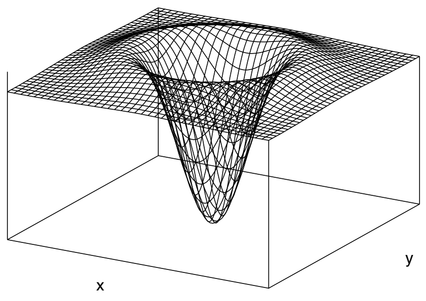


Figure 2.16: Laplacian of Gaussian filter kernel

Symmetric Nearest Neighbor

The symmetric nearest neighbor (SNN) filter is a non-linear procedure to reduce noise. In a 3×3 neighborhood, two opposing pixels build a pair, as denoted with equal letters in equation 2.43.

$$\begin{array}{ccc} c_1 & b_1 & d_1 \\ a_1 & x & a_2 \\ d_2 & b_2 & c_2 \end{array} \tag{2.43}$$

For each tuple, the element whose gray value is closer to the gray value of the pivot point (the nearest neighbor) is stored, the other one discarded.

$$\mathbf{N} = \{a, b, c, d\} \quad \text{with} \quad n = \begin{cases} n_1 & \text{for} \quad n_1 - x < n_2 - x \\ n_2 & \text{else} \end{cases} \tag{2.44}$$

The average of these four nearest neighbors then is inscribed in the pivot point.

$$x = \frac{1}{4}(a + b + c + d) \tag{2.45}$$

The SNN filter smoothes single pixels with a large gray value difference compared to their neighborhood, as the pivot point is not used for calculating the result. The main advantage of the SNN filter compared to linear smoothing filters is that edges and details are retained in the process. Drawbacks of this method are that it leads to an artificial look and very blocky images.

Median Filter

The median filter is used to remove outliers, meaning pixels whose gray value differ greatly from the neighboring pixels. Instead of using the arithmetic mean of the surrounding pixels, the pixels from the neighborhood are sorted according to their gray value and the pivot point is replaced with the center element of the sorted sequence. An example is given in equation 2.46. The center pixel with the grayvalue of 89, clearly an outlier, is detected as such by the median filter and replaced with the median element of the corresponding neighborhood, which is 14.

$$\begin{matrix} 15 & 16 & 15 \\ 14 & \mathbf{89} & 13 \\ 13 & 12 & 12 \end{matrix} \mapsto \{12, 12, 13, 13, \mathbf{14}, 15, 15, 16, 89\} \mapsto 14 \tag{2.46}$$

The median filter is an excellent tool to dispose salt and pepper noise, which are single, randomly distributed pixels with no connection to their neighborhood, for example caused by broken sensor elements. The disadvantage of most median implementations is that they are very performance consuming for larger masks due to the sorting, which has to be done for every pixel. However, [Weiss 2006] presents an algorithm which decreases computational complexity from $O(r)$, where r is the radius of the mask to $O(\log r)$.

Morphological Operations

Morphological operations describe a set of operations, mainly used on binary images (images with the range $\mathbf{G} = \{0, 1\}$), to modify the form of features in the image. The most common morphological operations are erosion ($\ominus$), dilation ($\oplus$), opening ($\circ$) and closing ($\bullet$).

For morphological operations, images, as well as the structure elements, which are the morphological counterparts of the kernels described before, are considered as sets of coordinates, rather than signals. Such a set of coordinates is given by

$$\mathbf{P} = \{\mathbf{p} \in \mathbb{Z}^2\}, \qquad \mathbf{p} = (x, y) \tag{2.47}$$

The input image is defined as $\mathbf{I}$ and the structure element as $\mathbf{S}$. The morphological operations can then be described with set algebra.

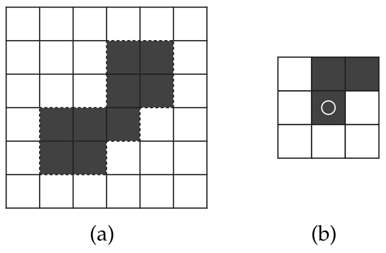


Figure 2.17: A binary input image (a) and a morphological structure element (b), the field marked with the circle is the pivot point

For the erosion, the structure element is moved over the image, the pivot point $\mathbf{p}$ is part of the resulting image, if all points of the structure element at this position are also element of the input image.

$$\mathbf{I} \ominus \mathbf{S} = \{\mathbf{p} \in \mathbb{Z}^2 | \mathbf{S}_\mathbf{p} \subseteq \mathbf{I}\} \tag{2.48}$$

Dilation is the counterpart to erosion. If at least one point in the structure element at pivot point position $\mathbf{p}$ is also element of the input image, the point also belongs to the result.

$$\mathbf{I} \oplus \mathbf{S} = \{\mathbf{p} \in \mathbb{Z}^2 | \mathbf{S}_\mathbf{p} \cap \mathbf{I} \neq \emptyset\} \tag{2.49}$$

Erosion and dilation can be used for noise reduction in binary images, however, the size of the remaining structures is considerably altered after the operation. By applying the counterpart operation with the same structure element the size can approximately be restored. The combination of erosion followed by dilation is called opening and defined by

$$\mathbf{I} \circ \mathbf{S} = (\mathbf{I} \ominus \mathbf{S}) \oplus \mathbf{S}. \tag{2.50}$$

Dilation followed by erosion is referred to as closing and can be written as

$$\mathbf{I} \bullet \mathbf{S} = (\mathbf{I} \oplus \mathbf{S}) \ominus \mathbf{S}. \tag{2.51}$$

Opening will remove thin parts sticking out of an object and split up thin connections between objects. Closing gaps in an objects shape will be closed and close objects will be combined.

For greyscale images the morphological operators can be implemented similar to the median operator. Instead of the center element of the sorted sequence for erosion the first and for dilation the last element is assigned to the pivot point.

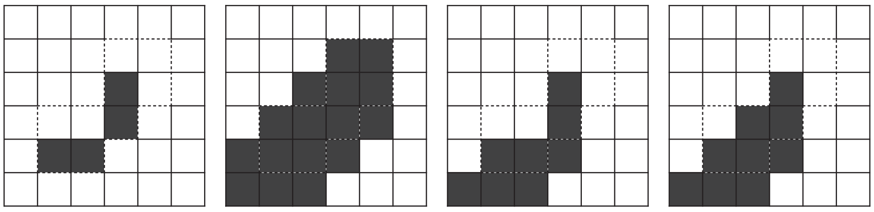

(a) erosion (b) dilation (c) opening (d) closing

Figure 2.18: The image from figure 2.17 after applying morphological operators

2.3.3 Global Operations

Just like point based operations, global operations can be described as a special case of local operations. Basically, a global operation is a neighborhood based operation which uses the whole image as the neighborhood.
Even though they are usually implemented on local subimages, typical global operations are the transform coding operations presented in sections 2.2.1 and 2.2.2 in this chapter. Another transform coding technique which is a global operation is the fourier transform (although usually implemented as *fast* fourier transform via multiple local operations, as well).
Other arbitrary operations include image scaling, rotation, translation or warping. However, these kinds of operations are not in the focus of this thesis and will not be described here in detail.

Chapter 3

State of the Art

Data acquisition techniques, as well as the accuracy of simulations were greatly improved over the last several years. This is a mixed blessing; For example, medical images or flow simulations are now extremely detailed but the amount of storage space needed has vastly increased.

With the limited amount of memory on today's graphics boards, which are utilized for a lot of volume visualization and processing tasks, efficient volume compression techniques are needed to load these large datasets into the graphics boards memory. Many of the existing compression techniques, mostly those for image compression, have been adapted to 3D for volume as well as video compression. The most important work regarding volume compression will be presented in section 3.1.

With the compressed data on the graphics board the visualization and processing algorithms, which are based on regular grid volume data, will not work anymore. Section 3.2 will give a short overview of the important research done to directly visualize and process compressed image and volume data. Most of the research regarding processing in the compression domain has been done for two dimensional image data. Thus, the presented processing approaches are based on 2D images but it is assumed that these techniques can be adapted to 3D volume data. The visualization methods presented, however are all for 3D data.

In the concluding section 3.3, advantages and disadvantages of the compression techniques with regard to processing will be discussed and the decision to choose wavelet compression as the basis for this thesis will be reasoned.

3.1 Compression of Volume Data

In [Schröder 1996] a brief overview of wavelet transforms and compression in computer graphics applications, including volume rendering is given. Volume wavelet transformation can easily be done by transforming the volume slice by slice with a two dimensional wavelet transform. However, it usually is better to exploit the similarities in the third dimension by a real three dimensional transform. This can be done by consecutively applying three one dimensional wavelet transforms on the volume, one along each axis. Shigeru Muraki discusses the application of three dimensional orthogonal wavelet transforms to volume data in [Muraki 1993]. Since then the main focus in this area has been on wavelet compression especially suited for volume rendering. Westermann [1994] and Guthe et al. [2002] present multiresolution frameworks for volume rendering which incorporate the implicit multiresolution representation in the wavelet decomposition with hierarchical structures. Kim & Shin [1999], Ihm & Park [1999] and Rodler [1999] focus on fast random access of voxel values in the compression domain without decompressing the volume as a whole to optimize the compressed data for volume rendering. Finally in [Hopf & Ertl 1999] a system for wavelet transforms utilizing graphics hardware is presented. The authors come to the conclusion that for simple (de-)composition the graphics hardware does not perform much better than software, however a lot of time can be saved in rendering applications due to the fact that the data transfer between CPU and GPU can be reduced severely.

For discrete cosine and discrete fourier transformation based compression techniques the extension to 3D and volume data can theoretically be made by using 3D versions of the cosine and fourier transformations. The 2D fourier transfor-

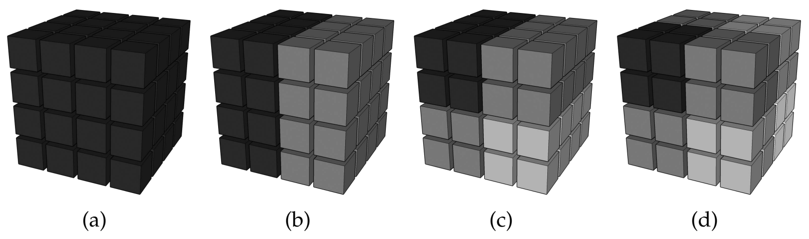


Figure 3.1: The 3D wavelet decomposition in 3 steps. (a) shows the original volume, (b), (c) and (d) the volume after wavelet transform in x-direction, followed by y- and z-direction. Note: the different shades of gray denote different subbands, not actual gray values in the volume data

mation

$$F(u,v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y) e^{-j2\pi(ux+vy)} dx dy \tag{3.1}$$

for example would be replaced by the 3D version

$$F(u,v,w) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y,z) e^{-j2\pi(ux+vy+wz)} dx dy dz. \tag{3.2}$$

In practice, however, these forms are not used for 2D nor 3D fourier transforms. Usually for the n-dimensional case the fourier transform is applied iteratively n times to the input data, just as described before for the wavelet transformation. For discrete data, like images the discrete fourier transformation

$$F(u) = \sum_{x=0}^{N-1} f(x) e^{-\frac{j2\pi}{N} ux}. \tag{3.3}$$

is used. Implementing this straightforwardly, however, would perform very poorly. Instead, the discrete fourier transform is usually computed via fast fourier transforms, like the Cooley-Tukey algorithm [Cooley & Tukey 1965]. This decreases the number of computations needed for n points from $2n^2$ to $2n \cdot log_2(n)$. The cosine transform is very similar, mainly differing in the use of the cos-function rather than e-functions. Porting the DCT defined by equation 2.6 to 3D results in

$$i_{\text{DCT}}(u,v) = \tfrac{1}{4} C(u) C(v) C(w) \cdot \\ \left[\sum_{x=0}^{7} \sum_{y=0}^{7} \sum_{y=0}^{7} i(x,y,z) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \cos \frac{(2z+1)w\pi}{16} \right] \tag{3.4}$$

However, just like the discrete wavelet transform and the discrete fourier transform the discrete cosine transform is usually computed by applying 1D transforms for each dimension. In addition, faster algorithms similar to the fast fourier transform do exist.

Chan et al. [1991] present volumetric compression using a three dimensional discrete cosine transform alongside quantization, adaptive bit-allocation and Huffman encoding. The authors conclude that while compression efficiency was better compared to slice-wise 2D transformation, the visual quality of the 3D transformation data was at least as good as of the 2D data. As random voxel access in the cosine and fourier domain require inverse transformations of the whole image, the authors of [Chiueh et al. 1997] partition the volume and apply the fourier transformation separately for every partition. Accessing voxels then requires only the inverse transformation of the circumjacent partition.

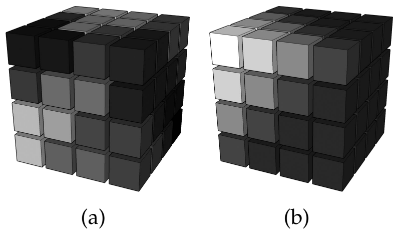


Figure 3.2: A random volume (a) and the much more sorted 3D cosine transform with the typical zig-zag structure (b)

Fractal compression can be adapted for volumes easily by matching three dimensional range- and domain-blocks instead of two dimensional. The additional dimension results in a broader pool of transform functions, benefiting in better compression rates and better visual quality but the trade off is an increase in time needed for the pairing tests as more possible matches have to be tested. In [Cochran et al. 1996] a fractal volume compression system is presented. The system reaches slightly higher compression rates at similar signal to noise ratios for real 3D compression compared to slice-wise compression (i.e $22.27 : 1$ compared to $18.06 : 1$ for a CT scan of a head). In [Erra 2005] an implementation of (two dimensional) fractal compression using graphics hardware is presented. The authors state that their GPU implementation can performs about 21 million pairing tests per second, whereas the CPU version yields about 220 thousand tests per second. Combining the use of 3D blocks with a GPU implementation could thus eliminate the largest disadvantage of the three dimensional fractal compression.

Vector quantization can make use of 3D structures by choosing 3D blocks as input- as well as codebook-vectors. Goldberg & Sun [1986] present an image sequence compression approach based on 2D blocks. It uses label replenish-

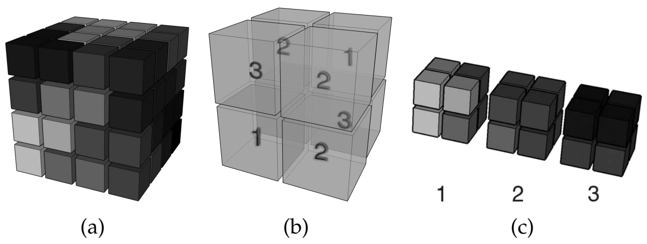


Figure 3.3: A volume (a) with a lookup volume for vector quantization based on $2 \times 2 \times 2$ input blocks (b) and the 3D codebook (c)

ment in temporally stationary regions to exploit similarities in the time dimension. They claim to achieve less error at same compression rates compared to the three dimensional approaches based on slice-wise coding. In [Choi & Chan 1996] a more advanced method for image sequence coding is introduced. It incorporates spatiotemporal correlation alongside motion information and uses $2 \times 2 \times 2$ blocks as vectors. Qian et al. [1995] present a 3D compression procedure, based on vector quantization, for hyperspectral images. In addition to the two spatial dimensions hyperspectral images have a third dimension, containing spectral information. They can be used to differentiate materials, which have similar or equal visual properties, but different properties in the part of the spectrum not visible for the human eye.

3.2 Processing in the Compression Domain

A lot of work has been done in the field of image denoising in the wavelet domain. Consider a noise free image, most of the wavelet coefficients would be zero. The non-zero coefficients likely would be comparatively large, as they would result from edges in the original image. Noise, however, would in the wavelet representation be transformed to small, non-zero coefficients. Consequentially in [Donoho & Johnstone 1995] an image denoising approach in the wavelet domain named SureShrink is presented, which incorporates soft-thresholding to remove small coefficients identified as noise directly in the wavelet domain. The SureShrink defines the threshold adaptively by minimizing the Stein Unbiased Risk Estimate (SURE). Another similar thresholding approach is presented in [Chang et al. 2000]. However, here the threshold is defined via Bayesian risk minimization. Finally the authors of [Kaur et al. 2002] propose a soft-thresholding method named NormalShrink, which outperforms the other approaches in visual quality as well as computation time. Another competing approach is Wiener filtering [Wiener 1949] in the wavelet domain. Instead of thresholding, linear Wiener filtering is applied in the wavelet domain. Work in this area has been presented by numerous authors [Strela 2000; Portilla et al. 2002; Asefa et al. 2006].

The authors of [Dorrell & Lowe 1995] as well as [Drori & Lischinski 2003] present more general image processing methods in the wavelet domain. In [Dorrell & Lowe 1995] the application of scalar multiplication and scalar addition as well as the addition of two images in the wavelet domain is shown. Furthermore arbitrary linear operators are ported into the wavelet domain. Therefore the operator (O) as well as the wavelet transform ($\mathcal{W}$) and the inverse wavelet transform ($\mathcal{W}^{-1}$) are represented as matrix multiplications. Con-

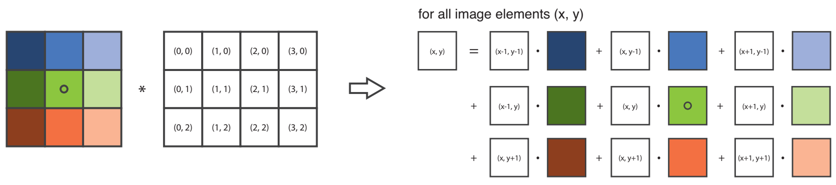


Figure 3.4: The standard convolution algorithm in the spatial domain

sequentially the operation transformed into the wavelet domain can be represented by

$$\tilde{f}(x,y)^{\star} = (\mathcal{W}\mathcal{O}\mathcal{W}^{-1})\ \tilde{f}(x,y), \tag{3.5}$$

where the ˜ denotes that the image is in wavelet form. [Drori & Lischinski 2003] the application of wavelet transform for warping, blending and convolution of images and image sequences is covered. Instead the usual implementation of moving the convolution kernel **M** over the image, which is illustrated in figure 3.4, here for a convolution kernel of edge length k convolution is described by creating k^2 copies of the original image **I**. For every position in the kernel one image has to be shifted by the inverted position (based on the pivot point of the kernel being the origin). Now the convolution is the sum of these images scaled by the corresponding kernel entry.

$$\mathbf{M} * \mathbf{I} = \sum_{i,j \in [-l,l]} k_{i,j} \mathbf{I}^{i,j} \tag{3.6}$$

where $l = \lfloor k/2 \rfloor$ and $\mathbf{I}^{i,j}$ denotes the image translated by i, j. This technique is illustrated in figure 3.5. The pixels marked with the dotted line are not part of the original image domain. They are the counterparts of the posi-

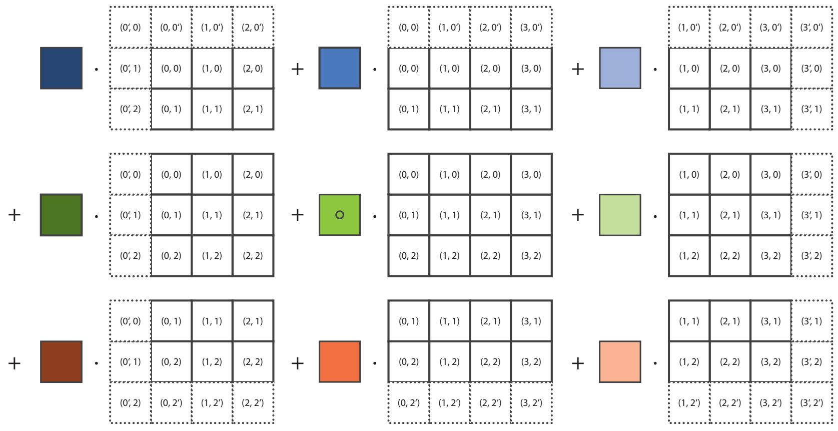


Figure 3.5: The alternative convolution algorithm in the spatial domain

tions, where the pivot of the convolution kernel overhangs out of the image boundary. Usually these pixels are extrapolated from the boundary pixels. As already shown in the previously mentioned publications, scalar multiplication as well as image addition can easily be transferred into the wavelet domain. The shift operation, however, is not trivial in the wavelet domain. Thus, the shift is done in the spatial domain and the resulting images are then transformed into the wavelet domain. However, a number of people have proposed modified wavelet transforms which allow shift operations in the wavelet domain. Although the technique is mostly the same or at least very similar for all publications the approaches are identified by different names; the à trous algorithm [Shensa 1992], shift invariant wavelet transform [Cohen et al. 1997; Bradley 2003], undecimated discrete wavelet transform [Guo & Burrus 1996], or overcomplete discrete wavelet transform [Sebe et al. 2002]. As indicated by the name of the last one the basic idea behind these approaches is over-sampling the input signal and computing the translation by interpolating the over-sampled data.
For direct visualization of wavelet compressed data, the already mentioned publications [Kim & Shin 1999; Ihm & Park 1999; Rodler 1999] (in section 3.1), presenting fast random access in the wavelet domain are of interest. [Grosso et al. 1996] focuses on efficient data structures to implement volume rendering in the compression domain. In [Lippert & Gross 1995] an approach based on the fourier slice theorem (see next section, respectively figure 3.7 for more information) is presented.

[Smith & Rowe 1993] introduces image manipulation algorithms in the DCT domain. The authors show how to apply pixel addition, pixel multiplication as well as scalar addition and scalar multiplication directly on the quantized blocks in the frequency domain. They show that these operations can be applied much faster in the DCT domain than in the spatial domain due to the sparseness of the transformed arrays. In the subsequent publication [Smith & Rowe 1996] this work is extended to the application of linear local and global operations, such as convolution, scaling, rotation etc. Images, as well as the image operations are formulated as tensors. The first part of the JPEG-compression pipeline up to the entropy coding can be described as a product of the tensors for the cosine transform, the zig-zag scanning and scaling. The image is formulated as a two dimensional array of first rank tensors named SC-vectors, which represent the 8×8 blocks introduced in section 2.2.1. For processing the compression pipeline has to be reversed until the SC-vectors are accesible. Then the JPEG decompression without entropy coding, the desired operation, and the re-compression can be formulated by the product of the corresponding tensors. However, in this representation the operations are a lot more expensive than in the spatial domain and the entropy decoding and coding which has to be done before and after the operations is also quite costly. For example a shrink by two requires an average of 256 multiplies per pixel in the DCT domain, compared to only four in the spatial domain. To reduce cost a method called condensation is presented. The sparseness of the tensors is exploited by modifying the compressed domain operators so that the results are similar, but the operator is more sparse and thus less costly to compute. In [Dorrell 1996] a similar approach is presented. Additionally an architecture for a JPEG codec with a programmable processing unit as coprocessor is presented. Shen & Sethi [1996] present another similar approach especially suited for edge detection with large Laplacian of Gaussian masks. The authors conclude, that not only a speedup, compared to non-compressed processing can be reached, but also their approach yields better edges (see figure 3.6). While they focused on edge detection with LoG, the method can be used universally with linear filter kernels. In the subsequent publication [Shen et al. 1998] the application to video editing in the compressed domain is presented.

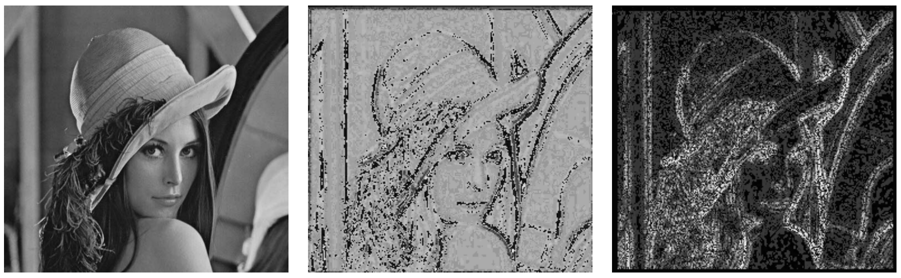

(a) Original (b) DCT convolution (c) Spatial domain convolution

Figure 3.6: Edges in the well known Lena image. Courtesy of [Shen & Sethi 1996]

For visualization [Dunne et al. 1990] and [Malzbender 1993] use the fourier projection-slice theorem to render images directly from the data in the fourier domain (see figure 3.7). The fourier projection-slice theorem states that for an m-dimensional image an n-dimensional slice in the frequency domain through the origin is equal to the n-dimensional fourier transformation of the original image projected on an n-dimensional slice. Thus, for volume rendering one can simply place a plane into the fourier transformed volume which contains the origin and is parallel to the viewing plane. Inverse fourier transform applied to the image embedded in this plane then yields a projection of the original volume onto the viewing plane. While this approach is faster than traditional volume rendering modes it is limited to projection, which results in X-ray like visualization only.

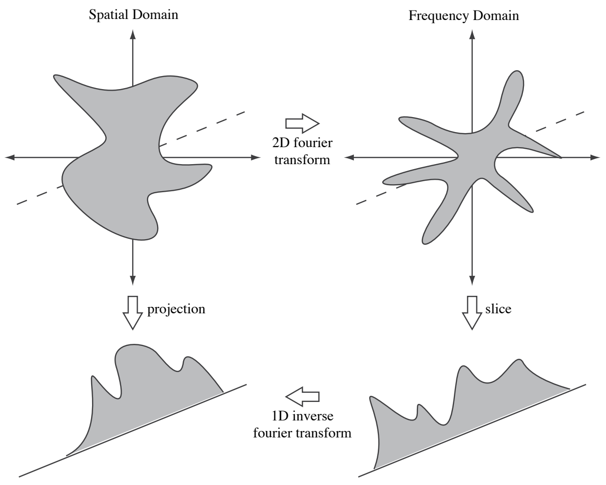


Figure 3.7: Volume projection with the fourier projection-slice theorem

Currently, there are no publications dealing with processing of fractal compressed data. Due to the virtual form of the codebook in fractal compression neighborhood based processing in the compression domain seems impossible. Homogeneous, pixel based operations might be possible as the functions contain brightness and contrast adjustments, but one would need to investigate the behavior of adjustments in these functions in the iterative process in the image reconstruction.
Partial image reconstruction followed by standard processing on the reconstructed part and re-compression also seems improbable as the iterative process of reconstruction is quite time consuming and the re-compression would need at least a large part of the processed image to find a new domain block matching the processed range block.
Finally, a divide and conquer approach like partitioning the original image into subimages which are then treated as separate images until the final reconstruction step would be possible. Every subimage would have to be completely decompressed processed and re-compressed in the processing step. However, this would in most cases result in notably worse image quality as the set of domain blocks to match the range blocks is much smaller for each subimage and thus often much worse matches would have to be used to build the iterative function system.

A broad overview of image processing in combination with vector quantization is given in [Cosman et al. 1993]. Specifically point based operations are found to be suited for vector quantized data. These operations can be applied directly to the codebook. As different codewords might have the same shape after processing, though, rebuilding the codebook might be necessary to achieve optimal compression. Local operations usually incorporate a neighborhood significantly exceeding the size of the codewords, thus making this class of operations working directly on the codebook impossible. The authors propose an alternate approach to edge detection with variable rate vector quantization (VRVQ). In VRVQ higher bitrates are used for areas with high activity, whereas homogeneous or inactive regions are compressed with fewer bits. This implicit information can then be used to classify codewords with higher bitrates as edges. The results, as shown in figure 3.8, however, are not comparable to the standard edge detection algorithms based on linear filtering.

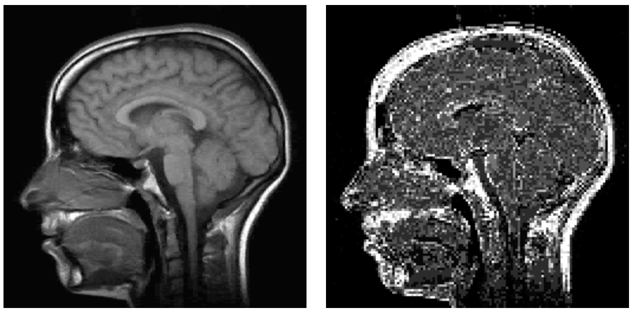

(a) Compressed image (b) After edge detection

Figure 3.8: At 1.8 bits per pixel vector quantization compressed MRI Scan and the result of the proposed VRVQ based edge detection. Images courtesy of [Cosman et al. 1993]

In [Ning & Hesselink 1992] a volume rendering system using vector quantized data is introduced. In the subsequent publication [Ning & Hesselink 1993] the authors extend their system with volume shading by also compressing the surface normal field gathered from the volume and a modified raytracing approach, which first performs the raytracing block-wise on the codebook from the desired viewing direction and then adds the blocks along the ray through the volume. To achieve fast rendering, however, each block must contain enough information to interpolate inter-cell values. Due to this additional information, only very low compression rates can be achieved. In combination with the proposed pre-integrated shading the resulting amounts of data are just as large as the input data.
With the increasing use of graphics hardware in volume rendering, vector quantization based raycasting was also ported to the GPU [Schneider & Westermann 2003].

3.3 Summary and Discussion

In the previous two sections, four volume compression methods were presented. Additionally, where available recent developments of processing and visualization techniques for compressed image and volume data were discussed.

Fractal compression seems to promise high compression rates and since fractal compressed data is virtually resolution independent, because the iterated function system can be applied at any resolution make it a very interesting approach. In addition, the GPU accelerated compression system, presented in [Erra 2005] improves the computation performance remarkably. However, fractal compression has several drawbacks, which make it unfeasible to use it as compression technique in this thesis. It is virtually impossible to compress data with fractal compression without losing information. This is a large disadvantage for a system which operates on medical data, where it is important to alter the data as little as possible. Furthermore, as discussed in the previous section there seems to be no probable way to operate directly on the compressed data and even intermediate solutions do not appear to be practical.

The two transform coding techniques, the wavelet and cosine transform, do have very smilar properties. Both have proven to be valuable in widely used image compression formats (JPEG and JPEG 2000), where both transforms are used as the first step of a compression pipeline. The compression results, both compression rate and visual quality highly depend on the steps after the transform itself. Approaches to directly perform commonly used image processing algorithms have been presented for the wavelet transform [Dorrell & Lowe 1995; Drori & Lischinski 2003], as well as the cosine transform [Smith & Rowe 1993, 1996]. Both transformations are not lossy in principle. Whether the compression will be lossless or not depends on the quantization and compression used for the coefficients. The biggest difference between the two transforms is the size of the blocks the image is divided into to perform the transformation. While for a single level Haar wavelet transform blocks of $2 \times 2 \times 2$ are used and every voxel can be reconstructed by a combination of eight coefficients, for the cosine transform typically blocks of $8 \times 8 \times 8$ voxels are used for transformation. This means to reconstruct a single voxel from the transformed data in the worst case 512 coefficients have to be used for computation (note that usually most of the coefficients are zero and thus do not contribute to the calculation). This means, for an operation which might require (partial) decompression of the volume, or random access to single voxels the wavelet transform is less computational expensive than the cosine transform.

Due to the fact that the vectors in vector quantization consist of items from the original image domain, vector quantization seems the most promising compression technique with regard to processing directly on the compressed data. Additionally, visualization from vector quantized data was presented in numerous publications [Ning & Hesselink 1992, 1993; Schneider & Westermann 2003]. Even though vector quantization is especially suited for point based operations, local operations suffer from the small vectors, which usually are not any bigger than $2 \times 2 \times 2$ voxels. Thus, multiple vectors are needed to apply a local operation to one voxel. A major drawback of vector quantization when modified for visualization, however, is the fact that compression rates are quite low.

When comparing vector quantization and wavelet compression directly, the main advantage of vector quantization is that all point based operations can be applied directly to the codebook vectors (even though the codebook might not be the optimal codebook after the operation), however, most of these operations can also be applied in the wavelet domain. For local operations and random access both techniques are about equal as the blocks used in both approaches are the same size. However, the reconstruction needed for (single level) wavelet transformed data slightly increases computational cost. (For now, the special case of linear filtering, directly on shift invariant wavelet transformed data is ignored and it is assumed that a neighborhood based operation requires block-wise reconstruction of the data.)
With vector quantization, optimized according to [Ning & Hesselink 1993], as well as single level wavelet compression rates will be limited. Wavelet compression, however will offer the possibility for lossless compression if needed.

Thus far, fractal compression as well as cosine transform based compression were eliminated as possible compression techniques to function as the basis for this thesis. Wavelet compression and vector quantization, even though very different approaches are very similar regarding the aforementioned features. Finally, for this thesis wavelet compression has been favored because of the future prospects. With possible higher level wavelet transforms the compression rates can be enhanced considerably. Additionally the option to implement a shift invariant wavelet transform, making linear filtering possible directly in the compression domain, would be a large advantage over vector quantization, even though it is not realized in this thesis.

Chapter 4

Wavelet Compression of Volume Data

In this chapter, the specifics of the wavelet compression implemented in this thesis are described. Section 4.1 gives a more practical overview of the Haar wavelet transform, which was used compared to the general overview in section 2.2.1. In the following section 4.2 an overview of the data structures used in the implementation to exploit the sparseness of the wavelet transformed array is given. Concluding, in section 4.3 the implementation details for the framework *Cascada* are presented.

4.1 The Haar Wavelet Transform

The Haar wavelet transform can be described as an averaging and detail process [Welstead 1999].
Consider a four pixel, one dimensional image $i = \{i_1, i_2, i_3, i_4\}$. Averaging over two pixels

$$\begin{aligned} a_{1,0} &= (i_1 + i_2)/2 \\ a_{1,1} &= (i_3 + i_4)/2 \end{aligned} \tag{4.1}$$

(the first subscript always denotes the level, the second the position in the subband) results in a coarser image with only two pixels: $i_{\text{avg}} = \{a_{1,0}, a_{1,1}\}$. Obviously information is lost if i_1 and i_2 respectively i_3 and i_4 have different values, as one can reconstruct the original image only to $\{a_{1,0}, a_{1,0}, a_{1,1}, a_{1,1}\}$.

To achieve exact reconstruction the differences (or coefficients)

$$\begin{aligned} d_{1,0} &= (i_1 - i_2)/2 \\ d_{1,1} &= (i_3 - i_4)/2 \end{aligned} \tag{4.2}$$

have to be added to the image representation, resulting in the wavelet form $\tilde{\imath}_{level1} = \{a_{1,0}, a_{1,1}, d_{1,0}, d_{1,1}\}$. The original image can now be reconstructed by

$$\begin{aligned} i'_1 &= (a_{1,0} + d_{1,0}) \\ i'_2 &= (a_{1,0} - d_{1,0}) \\ i'_3 &= (a_{1,1} + d_{1,1}) \\ i'_4 &= (a_{1,1} - d_{1,1}). \end{aligned} \tag{4.3}$$

Usually the averaging and differencing process is done recursively on the average image i_{avg} until the number of averages, which are not compressed, as described in section 2.2.1, is as small as desired:

$$\begin{aligned} a_{2,0} &= (a_{1,0} + a_{1,1})/2 \\ d_{2,0} &= (a_{1,0} - a_{1,1})/2. \end{aligned} \tag{4.4}$$

This results in the final transformed representation $\tilde{\imath}_{level2} = \{a_{2,0}, d_{2,0}, d_{1,0}, d_{1,1}\}$. The number of recursion needed to get to a certain coefficient is usually called the level at this point in the wavelet transform. Note that the image extents must be multiples of 2^{level} to reach that level in the transform.

To adapt the Haar wavelet transform to three dimensional data the averaging and differencing process is first done line-wise in x-direction, on the resulting image the process in done row-wise in y-direction and finally again on the result slice-wise in z-direction (compare figure 3.1 in the previous chapter). This process can be subsumed to the following set of calculations for each $2 \times 2 \times 2$ block

$$\begin{aligned}
\tilde{\imath}_{000} &= (i_{000} + i_{100} + i_{010} + i_{110} + i_{001} + i_{101} + i_{011} + i_{111}) \;/\; 8 \\
\tilde{\imath}_{100} &= (i_{000} - i_{100} + i_{010} - i_{110} + i_{001} - i_{101} + i_{011} - i_{111}) \;/\; 8 \\
\tilde{\imath}_{010} &= (i_{000} + i_{100} - i_{010} - i_{110} + i_{001} + i_{101} - i_{011} - i_{111}) \;/\; 8 \\
\tilde{\imath}_{110} &= (i_{000} - i_{100} - i_{010} + i_{110} + i_{001} - i_{101} - i_{011} + i_{111}) \;/\; 8 \\
\tilde{\imath}_{001} &= (i_{000} + i_{100} + i_{010} + i_{110} - i_{001} - i_{101} - i_{011} - i_{111}) \;/\; 8 \\
\tilde{\imath}_{101} &= (i_{000} - i_{100} + i_{010} - i_{110} - i_{001} + i_{101} - i_{011} + i_{111}) \;/\; 8 \\
\tilde{\imath}_{011} &= (i_{000} + i_{100} - i_{010} - i_{110} - i_{001} - i_{101} + i_{011} + i_{111}) \;/\; 8 \\
\tilde{\imath}_{111} &= (i_{000} - i_{100} - i_{010} + i_{110} - i_{001} + i_{101} + i_{011} - i_{111}) \;/\; 8
\end{aligned} \tag{4.5}$$

for first level decomposition, respectively

$$\begin{aligned}
i'_{000} &= (\tilde{\imath}_{000} + \tilde{\imath}_{100} + \tilde{\imath}_{010} + \tilde{\imath}_{110} + \tilde{\imath}_{001} + \tilde{\imath}_{101} + \tilde{\imath}_{011} + \tilde{\imath}_{111}) \\
i'_{100} &= (\tilde{\imath}_{000} - \tilde{\imath}_{100} + \tilde{\imath}_{010} - \tilde{\imath}_{110} + \tilde{\imath}_{001} - \tilde{\imath}_{101} + \tilde{\imath}_{011} - \tilde{\imath}_{111}) \\
i'_{010} &= (\tilde{\imath}_{000} + \tilde{\imath}_{100} - \tilde{\imath}_{010} - \tilde{\imath}_{110} + \tilde{\imath}_{001} + \tilde{\imath}_{101} - \tilde{\imath}_{011} - \tilde{\imath}_{111}) \\
i'_{110} &= (\tilde{\imath}_{000} - \tilde{\imath}_{100} - \tilde{\imath}_{010} + \tilde{\imath}_{110} + \tilde{\imath}_{001} - \tilde{\imath}_{101} - \tilde{\imath}_{011} + \tilde{\imath}_{111}) \\
i'_{001} &= (\tilde{\imath}_{000} + \tilde{\imath}_{100} + \tilde{\imath}_{010} + \tilde{\imath}_{110} - \tilde{\imath}_{001} - \tilde{\imath}_{101} - \tilde{\imath}_{011} - \tilde{\imath}_{111}) \\
i'_{101} &= (\tilde{\imath}_{000} - \tilde{\imath}_{100} + \tilde{\imath}_{010} - \tilde{\imath}_{110} - \tilde{\imath}_{001} + \tilde{\imath}_{101} - \tilde{\imath}_{011} + \tilde{\imath}_{111}) \\
i'_{011} &= (\tilde{\imath}_{000} + \tilde{\imath}_{100} - \tilde{\imath}_{010} - \tilde{\imath}_{110} - \tilde{\imath}_{001} - \tilde{\imath}_{101} + \tilde{\imath}_{011} + \tilde{\imath}_{111}) \\
i'_{111} &= (\tilde{\imath}_{000} - \tilde{\imath}_{100} - \tilde{\imath}_{010} + \tilde{\imath}_{110} - \tilde{\imath}_{001} + \tilde{\imath}_{101} + \tilde{\imath}_{011} - \tilde{\imath}_{111})
\end{aligned} \tag{4.6}$$

for reconstructing the image. $\tilde{\imath}_{xyz}$ denotes the value in the wavelet transformed block at position (x, y, z) and i_{xyz} respectively i'_{xyz}, the values in the image and recompositioned image at position (x, y, z) in local block coordinates. The average or three times low pass filtered coefficient is $\tilde{\imath}_{000}$ whereas a 1 means that the coefficient belongs to the high pass sub band for this direction.

For this thesis, level one transformations are always used. This is a tradeoff between compression rate and fast random accessibility as the number of computation increases quadratically with the number of levels, as for some operations which require the decomposition of the original voxel values fast access is indispensable. A maximum of eight coefficient lookups and seven additions is needed to reconstruct a voxel in 3D, as can be seen in equation 4.6 when using level one Haar wavelet transforms. At level two an additional 64 lookups and 56 additions are needed to reconstruct all level one coefficients needed to reconstruct the desired voxel. The downside of single level decomposition is that the average subband which is one eighth of the original volumes size persists uncompressed, thus a theoretical maximum compression rate of $1 : 8$ can be reached. As will be shown later on, approximate compression rates of $1 : 4$ to $1 : 5$ can be reached with this approach at good visual quality. The focus of most wavelet compression approaches is on maximization of compression rates, thus as few residual average coefficients as possible are preserved. However the denoted compression rates seem an acceptable compromise to preserve fast access, yet allow it to load significantly larger datasets into the GPUs memory.
However, the processing algorithms presented in this thesis are fully compatible with higher levels of wavelet decomposition, meaning with a few adjustments the decomposition can always be extended to support these. The data structure had to be extended by importance volumes and entry point maps for all consecutive levels. Compression and decompression algorithms as well

as the operations requiring these would have to be applied recursively/iteratively to the residual average coefficients. The adjustments to the operations working directly in the transformation domain are typically insignificant, if needed at all. For example, for inverting or scalar multiplication no adaption is needed.

4.2 Data Structure

After the wavelet decomposition, the first step for compression is to separate the average coefficients which will remain uncompressed from the rest of the coefficients. Initially the average coefficients are always at position $(0, 0, 0)$ of a $2 \times 2 \times 2$ block. This means every second coefficient in every second row in every second slice is an average coefficient, as denoted by the green blocks in figure 4.1(b). To store the average coefficients a volume with half the size in every dimension is used (see figure 4.1(c)). A coefficient at position (x, y, z) in the original volume is then stored at position $(x/2, y/2, z/2)$ in the average volume. In memory the volume is stored as a simple one dimensional vector. The averages are sorted row-wise from the first to the last slice. Thus, a coefficient with the 3D coordinates (x, y, z) has the index

$$idx = x + y \cdot \text{rowsize} + z \cdot \text{slicesize} \tag{4.7}$$

in the vector. Rowsize, denotes the extent of the average volume in x-direction and slicesize the number of coefficients in a single slice in the average volume. The remaining coefficients are stored in a different data structure. To exploit the sparseness of the coefficient volume, the fixed volume grid needs to be broken up and the coefficients smaller than the defined compression threshold must be discarded. This can be simply done by only storing the coefficients larger than the threshold in a one dimensional, linear vector. Doing so, how-

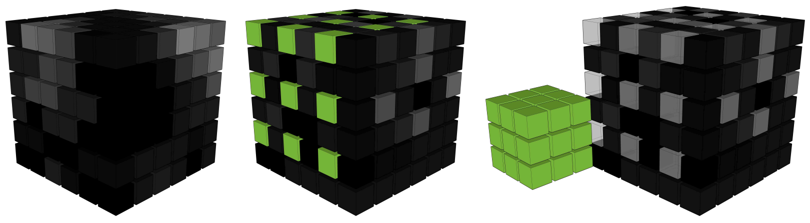

(a) Input volume (b) Wavelet decomposition (c) Detached averages

Figure 4.1: Separating average coefficients and high band coefficients. The averages are marked green

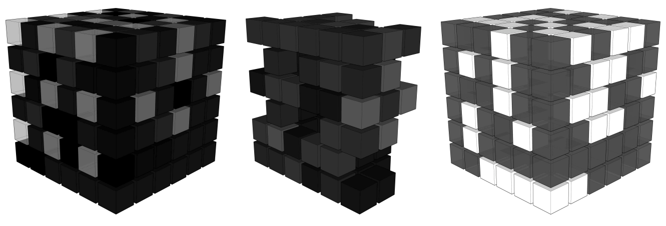


Figure 4.2: The remaining coefficients, the sparse coefficient volume and the importance volume

ever, the implicit information of the position in the original volume would get lost. Thus, to be able to relocate a coefficient to its original position, its orignal coordinates are stored alongside the coefficient. When trying to access a coefficient at a desired position (x, y, z) in the volume, it is inevitable to search the whole set of coefficients up to the position of the desired coefficient. This can be quite time consuming, especially when the coefficient is in the rear end of the vector.

To minimize search time a modified approach based on [Grosso et al. 1996] is used. Instead of storing all coefficients as a single stream, the coefficients with the same (x, y) coordinates are stored as a group. When using a 2D array of different sized vectors the sparse volume representation would look as shown in figure 4.2(b). Instead of storing the complete position (x, y, z) with every coefficient it is sufficient to store only z, as the x and y coordinates are preserved in the (x, y) coordinates of the 2D array. A coefficient at position (x, y, z) can then be accessed by looking up the vector at position (x, y) and searching for the coefficient with the position z attached.

In addition, to avoid searching, when the coefficient is zero and thus not in the sparse volume, a binary importance volume of the same size as the original volume is used (figure 4.2(c)). Only when the bit at the desired position is set, the coefficient is searched. This importance volume can also be used to determine the index of a coefficient in the sparse volume representation, making it unnecessary to store any position information with the coefficients. When trying to access a coefficient at position (x, y, z), first this position is looked up in the importance array. If the bit is set the number of bits set with the same x and y and a smaller z coordinate denotes the index of the coefficient in the vector at (x, y) in the sparse volume representation.

As an array of differently sized vectors is not available on the GPU another step is performed to yield the final representation for the compressed coefficients. The sparse volume vectors are stored continuously in a simple 1D vector. To preserve fast access to the (x, y) coordinates a two dimensional array with the size of a single slice is created. Every position (x, y) in this entry point array corresponds to the vector with the same x and y coordinates in the sparse volume representation and holds the index of this vector inside the coefficient vector.

The complete procedure to access a given coefficient at volume position (x, y, z) is shown in figure 4.3. A unique color was assigned to every row in the volume for easier reference in the coefficient vector, for easier allocation in the coefficient map. Additionally for larger z values brighter versions of the color are used. The coefficient array is shown as a 2D field solely for visualization. In the actual implementation it is stored as describe above in a one dimensional vector.

First it is verified that the bit at position (x, y, z) in the importance map is set. Then, the number of bits set between (x, y, z) and $(x, y, 0)$ is counted. This results in the $+2$ entry denoting the index of the coefficient in the corresponding sparse volume vector. The offset of this vector in the coefficient vector can now be looked up at position (x, y) in the entry points map. The first item of the sparse volume vector is at position 15 in the coefficient vector. The coefficient can now be fetched from position $15 + 2$ in the coefficient vector.

In a nutshell, the data structures used in thesis consists of the pseudo 3D average volume, which for single level deomposition is always one eighth of the size of the original volume. For fast access the binary significance map, another pseudo 3D volume (stored in a standard 1D vector), is used, whose size is $1/16$ of the original volume (for 16-bit datatypes). Additionally, the 2D entry points map takes up the size of one slice. Finally, the coefficients are stored in a 1D vector. Its size solely depends on the number of non-zero coefficients.

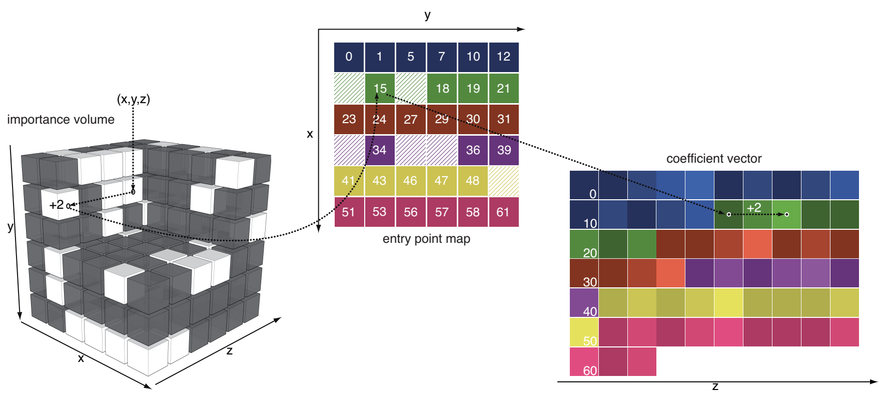


Figure 4.3: Accessing a coefficient with volume coordinates (x, y, z) in the presented data structure. Also compare listing 4.1 for pseudo code.

Listing 4.1: The function to fetch a coefficient in pseudo code

```
getCoefficient( x, y, z ){

    // only if this position is important
    if( importanceMap[ x, y, z ] ){

        // get the entry Point
        index = entryPointsl[ x, y ];

        // the number of importance bits set
        offset = 0;

        // from current z position to 0
        for( zz = z − 1; zz >= 0; zz−− ){

            // if the importance bit is set...
            if( importanceMap[ x, y, zz ] ){

                // ...increase the offset
                offset++;
            }
        }
        // fetch the coefficient at index + offset
        return Coefficients[ index + offset ];
    }
    // if importance bit at ( x, y, z )  is not set return 0.0
    return 0.0;
}
```

4.3 Implementation Details

The most important parts added to *Cascada* for compression and decompression can be seen in figure 4.4. The *CompressedVolume* class implements the data structure presented in section 4.2. The class includes constructors to create an empty volume of minimal size, an empty volume of a given size and a volume as a copy of another compressed volume.

The required data fields are implemented as private one dimensional STL vectors of the required type. The data can be accessed via the corresponding get functions which return pointers to the fields. The flag *m_changes* is used to denote what kind of changes an operation caused to the volume. When a processing operation is applied to the volume the flag is changed from *NO_CHANGES* to *MAJOR_CHANGES* or *MINOR_CHANGES*. *MAJOR_CHANGES* means all data fields were changed during processing, *MINOR_CHANGES* means only the averages and coefficients were touched, thus the importance volume and the entry points map do not have to be updated.

The *m_threshold* variable is used to save the compression threshold used to initially compress the volume. If an image processing operation requires (partial) decompression the threshold is needed for re-compression.

The minimum and maximum gray values of the volume are needed for linear scaling during rendering stage. It is quite costly to compute them from the compressed data, as full decompression would be needed. Thus it is advised

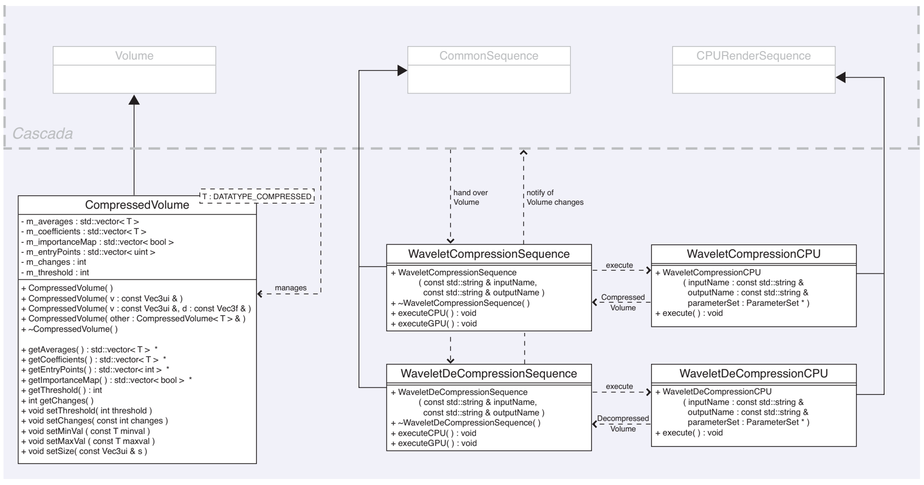

Volume
CommonSequence
CPURenderSequence
Cascada
T : DATATYPE_COMPRESSED
CompressedVolume
- m_averages : std::vector< T >
- m_coefficients : std::vector< T >
- m_importanceMap : std::vector< bool >
- m_entryPoints : std::vector< uint >
- m_changes : int
- m_threshold : int
+ CompressedVolume( )
+ CompressedVolume(v : const Vec3ui &)
+ CompressedVolume(v : const Vec3ui &, d : const Vec3f &)
+ CompressedVolume(other : CompressedVolume< T > &)
+ ~CompressedVolume( )
+ getAverages( ) : std::vector< T > *
+ getCoefficients( ) : std::vector< T > *
+ getEntryPoints( ) : std::vector< int > *
+ getImportanceMap( ) : std::vector< bool > *
+ getThreshold( ) : int
+ int getChanges( )
+ void setThreshold(int threshold)
+ void setChanges(const int changes)
+ void setMinVal (const T minval)
+ void setMaxVal (const T maxval)
+ void setSize(const Vec3ui & s)
manages
hand over Volume
notify of Volume changes
WaveletCompressionSequence
+ WaveletCompressionSequence (const std::string & inputName, const std::string & outputName)
+ ~WaveletCompressionSequence( )
+ executeCPU( ) : void
+ executeGPU( ) : void
execute
Compressed Volume
WaveletCompressionCPU
+ WaveletCompressionCPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *)
+ execute( ) : void
WaveletDeCompressionSequence
+ WaveletDeCompressionSequence (const std::string & inputName, const std::string & outputName)
+ ~WaveletDeCompressionSequence( )
+ executeCPU( ) : void
+ executeGPU( ) : void
execute
Decompressed Volume
WaveletDeCompressionCPU
+ WaveletDeCompressionCPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *)
+ execute( ) : void

Figure 4.4: The compression class structure

to track these values during compression and processing operations, manipulating gray values. Therefore the *setMinVal* and *setMaxVal* functions are used to save the values alongside the compressed volume data. The member variables to store these values are already present in the *Volume* class, however here they can not be set from outside the volume but rather are updated through the *updateStatistics* function, which scans the whole volume for minimum and maximum gray values.
The compression and decompression routines themselves are implemented as sequences, which is the standard implementation for all volume operations in *Cascada*. A sequence is instantiated by the sequence manager, with an input and output volume name given. The sequence gets the volumes associated with these names from the volume manager. When the volume is processed the volume manager gets notified of the changes. Based on the current system state the sequence decides whether to execute the CPU or the GPU version of the related operation. As the main focus of this thesis is on processing of compressed data, it was decided to implement the wavelet compression and decompression only as CPU versions. Thus when *executeGPU* is called it falls back to *executeCPU*. Through this the implementation costs where acceptable, while receiving a compression which fitted the needs of this thesis.

Compression is divided into two main sections, the wavelet decomposition and the actual compression step. It can be visualized by a simple pipeline, shown in figure 4.5.
In the first part the input volume is decomposed into the wavelet average- and difference-coefficients. The average coefficients are already sorted into the av-

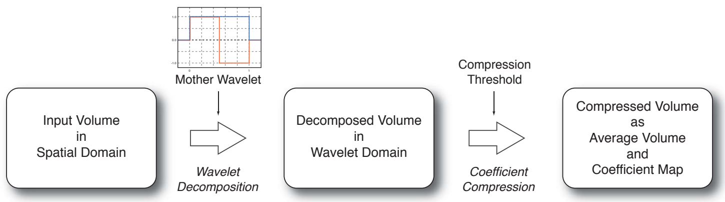

Mother Wavelet
Compression Threshold
Input Volume in Spatial Domain
Wavelet Decomposition
Decomposed Volume in Wavelet Domain
Coefficient Compression
Compressed Volume as Average Volume and Coefficient Map

Figure 4.5: The wavelet compression pipeline

erage volume at this point. The difference coefficients, however, for the time being remain in volume representation at their original position. At this stage the average volume and importance volume are created in their final forms. Pseudo code of the implementation is shown in listing 4.2. The algorithm processes the input volume block-wise, based on the wavelet 2^3 block structure. The eight voxels belonging to one block are fetched from the input volume and transformed into wavelet representation according to equation 4.5 (p. 46). The first coefficient is immediately saved in the *averageVolume*, while the seven remaining coefficients are subject to the subsequent first part of the compression step. All coefficients whose absolute value is smaller than the threshold, defined by the user are discarded. The coefficients larger than this threshold are saved in the *temporaryCoefficientVolume* at their original position. For every co-

Listing 4.2: The wavelet decomposition

```
for ( int x, y, z = 0; x, y, z < volumeSize; x, y, z += 2 ){

    // fetch the 2x2x2 block which will be decomposed
    waveletBlock[8] = fetchVoxelsIn( [ x, y, z .. x+1, y+1, z+1 ] );

    // decompose to wavelet coefficients
    coefficients[8] = decomposeToWavelet( waveletBlock );

    // extract the average coefficient
    averageVolume[ x/2, y/2, z/2 ] = coefficients[0];

    // sort the other coefficients into the temporary coefficient volume
    for( remaining seven coefficients ){

        // if coefficient is not discarded for compression
        if( abs( coefficient ) > compressionThreshold ){


            // the volume position of the current coefficient is needed
            position = volumePosition( coefficient );

            // set the important bit
            importanceMap[ position ] = true;

            // put the coefficient in the temporary volume
            temporaryCoefficientVolume[ position ] = coefficient;
        }
    }
}
```

efficient stored, the corresponding bit in the *importanceMap* is set.
Multiple iterations of this loop on partial regions of the volume can be run simultaneously without side effects. In the implementation this fact is used to parallelize the process with the *omp parallel for* construct of the OpenMP library. For more information on the OpenMP library refer to [OpenMP].

The entry point map and the coefficient vector are filled in the second step from the *temporaryCoefficientVolume*. The procedure is demonstrated in listing 4.3. First, the index to access the coefficient in the *coefficients* vector is initialized to 0. For the current (x, y) coordinate the *index* value is recorded into the *entryPoints* map. Then, the non zero coefficients from $(x, y, 0)$ to (x, y, sizez) are put consecutively into the *coefficients* vector. For every coefficient put into the vector *index* has to be increased by one. After the processing of one line in z-direction it is switched to the next (x, y) coordinate. The updated *index* is recorded in the *entryPoints* map and so on. As it is crucial to put the coefficients into the vector in the right order no parallelization for this loop can be applied.

After the decomposition and reordering of the coefficients the *temporaryCoefficientVolume* is deleted. Minimum and maximum gray values as well as the compression threshold are set in the compressed volume and, as all data fields were altered, the *m_changes* flag is set to *MAJOR_CHANGES*. After that the volume manager is notified that the volume has changed and the render mode is

Listing 4.3: The coefficient reorder process

```
// the index in the coefficient vector for every coefficient
index = 0;

// loop over the first slice
for ( int x, y = 0; x, y < sliceSize; x, y ++ ){

    // set the entry Point at position ( x, y )
    entryPoints[ x, y ] = index;

    // sort the coefficients from ( x, y, 0 ) to ( x, y, sizez )
    for ( int z = 0; z < sizez; z ++ ) {

        // get the coefficient from the Volume created in listing 4.2
        coefficient = temporaryCoefficientVolume[ x, y, z ];

        // if the coefficient is 0 it is discarded , else...
        if ( coefficient != 0 ){

            // ... it is put in the coefficient vector
            coefficients[ index ] = val;

            // the index is increased with every coefficient stored
            index++;
        }
    }
}
```

Listing 4.4: The wavelet decompression

```
for ( int x, y, z = 0; x, y, z < volumeSize; x, y, z += 2 ){

    // fetch the average coefficient
    average = averageVolume[ x/2, y/2, z/2 ];

    // fetch the remaining coefficients
    for( remaining seven coefficients ){

        // get the coefficients as shown in listing 4.1
        getCoefficient( current Position );
    }

    // recpompose the Voxel data
    composeVoxelValues( average, coefficients );
}
```

automatically switched to render from compressed data.

Decompression is straightforward. After getting the input and output volumes from the volume manager the image composition is accomplished block-wise by fetching the average value for the block from the *averageVolume* and the corresponding coefficients with the function shown in listing 4.1 on page 51. The voxel gray values are then composed with the equations shown in equ. 4.6 (p. 47) and recorded in the proper positions in the not compressed target volume. At the end of the decompression process the volume manager is notified of the changes and it is switched back to rendering from uncompressed data. The main decompression loop is shown in listing 4.4. It can be parallelized with the aid of *omp parallel for* just as the decomposition loop.

Chapter 5

Processing of Compressed Volume Data

This chapter presents the processing and visualization sequences added to *Cascada* for wavelet compressed data. In the first section (5.1) it will be shown mathematically how to transform certain processing operations into the wavelet space, if possible. The operations are then classified based on their application in wavelet form. In section 5.2 the implementation of some different operations will be presented exemplarily.

5.1 Mathematical Considerations

As shown in section 4.1 (equ. 4.6) every voxel (in the re-composed volume) can be described by a simple sum of coefficients. Different local positions in the wavelet block only differ by the signs of the coefficients in the sum.

$$f(u,v,w) = \sum_{n=0}^{8} s \cdot c_n \qquad \text{with} \qquad s \in \{-1, 1\} \tag{5.1}$$

(u, v, w) denote the local position in the wavelet block, $f(u, v, w)$ the gray value at this point and c the corresponding eight coefficients. For convenience it is differentiated between the low pass filtered average coefficient and the seven high band coefficients. As the average coefficient is always positive the sum can be formulated as

$$f(u,v,w) = c_{\text{avg}} + \sum_{n=0}^{7} s \cdot c_n. \tag{5.2}$$

In section 2.3.1 the application of a point a based operation T is described by

$$f^{\star}(x,y,z) = T_{x,y,z}\left(f(x,y,z)\right). \tag{5.3}$$

Combining the two equations 5.2 and 5.3 delivers

$$f^{\star}(x,y,z) = T_{x,y,z}\left(c_{\text{avg}} + \sum_{n=0}^{7} s \cdot c_n\right). \tag{5.4}$$

The most simple operations are addition of a constant value and scaling the gray value by a constant factor. Most homogeneous point based operations can be described as combinations of these two. Usually the operations are different for the wavelet sub bands. Therefore, the operator T usually is mapped to two separate operators T_{avg} and T_c for application on the average respectively high band coefficients.

Addition of a constant is defined by $T(g) = g + k$. Inserted in equation 5.4 this yields

$$\begin{aligned} f^{\star}(x,y,z) &= \left(c_{\text{avg}} + \sum_{n=0}^{7} s \cdot c_n\right) + k \\ &= \left(c_{\text{avg}} + k\right) + \sum_{n=0}^{7} s \cdot c_n. \end{aligned} \tag{5.5}$$

Thus adding a constant value to all pixels can be applied in the wavelet domain by adding a constant value to the average coefficients.

$$T_{\text{avg}}(c) = c + k \tag{5.6}$$

Multiplication with a constant value in the spatial domain is defined by $T(g) = g \cdot l$. Again, in combination with equation 5.4 this leads to

$$\begin{aligned} f^{\star}(x,y,z) &= \left(c_{\text{avg}} + \sum_{n=0}^{7} s \cdot c_n\right) \cdot l \\ &= \left(c_{\text{avg}} \cdot l\right) + \sum_{n=0}^{7} s \cdot \left(c_n \cdot l\right). \end{aligned} \tag{5.7}$$

Consequentially, the two wavelet operations are defined as

$$T_{\text{avg}}(c) = T_c(c) = c \cdot l \tag{5.8}$$

Combinations of addition and multiplication can be expressed by nesting operators. For example the operator $T(g) = (g + k) \cdot l$ can also be applied as $T_\cdot(T_+(g))$ with $T_+(g) = g + k$ and $T_\cdot(g) = g \cdot k$.

Also, it can easily be shown that the inhomogeneous **addition of two images/volumes** can be done in the wavelet domain simply by adding the two wavelet transformed volumes. For the one dimensional case consider the two discrete signals $\{a, b\}$ and $\{c, d\}$. It has to be shown that the sum of these two signals in the spatial domain $\{a + c,\ b + d\}$ yields the same result as the sum of the wavelet transformed signals. The Haar wavelet transforms of the two signals are defined by $\left\{\frac{a+b}{2},\ \frac{a-b}{2}\right\}$ respectively $\left\{\frac{c+d}{2},\ \frac{c-d}{2}\right\}$. Component-wise addition leads to

$$\left\{\frac{a+b}{2}, \frac{a-b}{2}\right\} + \left\{\frac{c+d}{2}, \frac{c-d}{2}\right\} = \left\{\frac{a+b+c+d}{2}, \frac{a-b+c-d}{2}\right\}. \tag{5.9}$$

Inverse wavelet transform results in

$$\begin{aligned}
&\left\{\frac{a+b+c+d}{2} + \frac{a-b+c-d}{2}, \frac{a+b+c+d}{2} - \frac{a-b+c-d}{2}\right\} \\
=&\left\{\frac{a+b+c+d+a-b+c-d}{2}, \frac{a+b+c+d-(a-b+c-d)}{2}\right\} \\
=&\left\{\frac{2a+2c}{2}, \frac{2b+2d}{2}\right\} \\
=&\{a+c,\ b+d\} \quad = \quad \{a,b\} + \{c,d\}
\end{aligned} \tag{5.10}$$

As the wavelet transform for higher dimensional data is nothing more than the repeated application of the one dimensional transform this is also valid for the addition of higher dimensional images. Note that the images must be of the same size and transformed to the same level.
Other operations such as inhomogeneous multiplication, potentiation or relational operators can not be transformed like this to be applicable in the wavelet domain.

In the following, when possible, wavelet transformed versions of the spatial domain image operators shown in section 2.3 are presented. For this thesis there will be mainly four classes of operations. The first two classes consist of the point based operations which can be applied within the wavelet domain. These can be further subdivided into those operations with $T_c(0) = 0$

and those which de not necessarily map zero to zero. Operations of the first class can be applied directly to the sparse coefficient representation without zeroes (i.e. after the last step in the pipeline). The second class requires the reversion of the last step as access to all coefficients including those whose value is zero is needed. The coefficients have to be re-compressed after processing. Point based operations which are applied in the spatial domain and thus require full re- and de-composition of the current voxel form the third class. The fourth class consists of operators which additionally require the re-composition of voxels, other than the current.

The **invert operator** in the spatial domain

$$T(g) = g_{\text{max}} - g \tag{5.11}$$

at length can be written as the combination of a simple addition and multiplication:

$$T(g) = g_{\text{max}} + (-1) \cdot g. \tag{5.12}$$

The combination of equations 5.6 and 5.23 results in the two wavelet operators

$$T_{\text{avg}}(c) \quad = \quad g_{\text{max}} + (-1) \cdot c \tag{5.13}$$

$$\text{and} \qquad T_c(c) \quad = \quad -c. \tag{5.14}$$

According to the above defined classification inversion is a class one operation.

Histogram shift, without clamping to minimum an maximum gray values in the spatial domain is a simple addition

$$T(g) = g + k. \tag{5.15}$$

As such the operation in the wavelet domain is specified by T_{avg} only, just as shown in equation 5.6.

$$T_{\text{avg}}(c) = c + g_{\text{offset}} \tag{5.16}$$

Histogram shift as defined here is a class one operation. However, it lacks the clamp operation as shown in section 2.3.1, which as a combination of relational operators, can not be implemented in the wavelet domain. As long as the results stay within the range of the datatype used in the actual implementation the clamping can be done live in the rendering stage and during decompression. This also has the advantage that as long the data remains in wavelet form the histogram shift is fully reversible.

To convert the **histogram spread** operation into wavelet representation, first the form of the operator is written slightly different to reach a representation that needs as few combinations of the multiplication and addition operators as possible:

$$\begin{aligned} T(g) &= (G-1) \cdot \frac{g - g_{\text{min}}}{g_{\text{max}} - g_{\text{min}}} \\ &= g \cdot \frac{(G-1)}{g_{\text{max}} - g_{\text{min}}} - \frac{g_{\text{min}}}{g_{\text{max}} - g_{\text{min}}} \end{aligned} \tag{5.17}$$

with the two constants

$$k = \frac{(G-1)}{g_{\text{max}}} \qquad \text{and} \qquad l = \frac{g_{\text{min}}}{g_{\text{max}} - g_{\text{min}}}. \tag{5.18}$$

this finally yields a simple combination of one addition and one multiplication

$$T(g) = g \cdot k - l \tag{5.19}$$

and thus the two wavelet operators are defined by

$$T_{\text{avg}}(c) \;=\; c \cdot \frac{(G-1)}{g_{\text{max}} - g_{\text{min}}} - \frac{g_{\text{min}}}{g_{\text{max}} - g_{\text{min}}} \tag{5.20}$$

$$T_c(c) \;=\; c \cdot \frac{(G-1)}{g_{\text{max}} - g_{\text{min}}}. \tag{5.21}$$

Just like inversion and histogram shift, histogram spread is a class one operation.

The only operation presented in section 2.3.1 which in the wavelet transformed form is a class two operation is the **difference image/volume operation**. It is a combination of the inhomogeneous volume addition and homogeneous multiplication by (-1), applied to the the second volume. As the difference volume operation is inhomogeneous it can not be described as a transformation on gray values, but as a transformation of gray values at a certain position (x, y, z).

$$\begin{aligned} T(f(x,y,z), e(x,y,z)) &= |f(x,y,z) - e(x,y,z)| \\ &= |f(x,y,z) + e(x,y,z) \cdot (-1)| \end{aligned} \tag{5.22}$$

Similarly, to the clamp operator in the histogram shift operation, the absolute operator ($|x|$) can not be applied in the wavelet domain, however, in most cases it is sufficient to apply it right at the rendering stage or during decompression. The operation in wavelet form without the absolute operator is defined by:

$$T_{\text{avg}}(c_1(x,y,z), c_2(x,y,z)) = c_1(x,y,z) + c_2(x,y,z) \cdot (-1) \quad (5.23)$$

$$T_c(c_1(x,y,z), c_2(x,y,z)) = c_1(x,y,z) + c_2(x,y,z) \cdot (-1) \quad (5.24)$$

Gamma Correction, **thresholding** and **correction of inhomogeneous illumination** all fall in class three. Gamma corrections can not be transformed in the wavelet domain due to the potentiation applied to the gray values. For thresholding to compare the gray value with the threshold wavelet form must be re-composed to voxel values and the correction of inhomogeneous illumination applies inhomogeneous multiplications to the volume, which also can not be transformed into the wavelet domain. Thus all three operations require the complete reversion of the wavelet pipeline (at least locally).

As shown in [Drori & Lischinski 2003] (see section 3.2) linear **local operations** can be described as a combination of the homogeneous multiplication operation and the inhomogeneous volume addition. However, it would require either the translation of the volume in the spatial domain or a shift invariant wavelet transform to translate the volumes directly in the wavelet domain. Decompressing the volume as a whole to translate it in the spatial domain, one the one hand, would increase the memory footprint too much for the desired application and as such makes this approach improbable. On the other hand, the implementation of a shift invariant wavelet transform is out of the scope of this thesis (and additionally has other drawbacks, like inferior compression rates). Hence, linear and non-linear local operations are treated the same, and thus are classified as class four operations. To process a single gray value the needed neighborhood is completely decompressed, the operation is applied in the spatial domain and the value gets re-compressed. The memory footprint increases only very slightly as only small parts of the volume need to be decompressed.

Computationally, class one operations will likely outperform their counterpart operations in the spatial domain. The operations themselves are very similar in the wavelet domain, however, they have to be applied only to the set of non zero coefficients which usually is much smaller than the number of voxels in the original volume. class two operations can be expected to perform slightly worse compared to the non compressed versions, as the last step in the wavelet pipeline has to be reversed and re-applied. class three and four operations usually should be noticeable slower, as the reversion of the complete wavelet pipeline is quite costly.

5.2 Implementation Details

To load the compressed dataset onto the graphics hardware with standard OpenGL the data fields have to be represented as textures. Subsection 5.2.1 gives an overview how this affects the data structure. In the following sections 5.2.2 and 5.2.3 the implementation details for rendering and processing in the compression domain are given.

5.2.1 The Data Structure on the GPU

To be able to access the compressed volume data on the GPU the data structure has to be slightly adapted. All data fields are wrapped either in two or three dimensional textures. To be fully compatible with older graphics cards all data fields into whom it is written need to be represented as 2D textures since write support for 3D textures is implemented for nvidia G80 based GPUs or newer only.

Depending on whether the position in the volume is necessary for processing or not and if the operation can be applied in the transform domain the input and output data differs.
Operations which were classified as class one in the previous section solely depend on the gray value and can be applied in the wavelet domain. For these operators only the coefficient data fields are needed for processing. Depending on the operation both, or only the average coefficients are loaded onto the graphics board. Therefore the the three dimensional average volume, as well as the one dimensional coefficient vector are represented as two dimensional RGBA textures. By using four channel RGBA textures four values can be processed at once by the vector units of the GPU.
All other operations including rendering require the complete set of data fields as input. For intuitive access the 3D average volume and importance volume are represented as three dimensional textures with one channel. The 2D entry Point map also is a single channel two dimensional texture. For read only access solely the 1D coefficient vector is transformed to a single channel 2D texture. The result of the operation is then written into a four channel 2D texture which has to be rearranged into the input data structure in a second step (which is performed in software). Therefore the slices of the original volume are laid out into a large 2D texture. This is also done for the non compressed

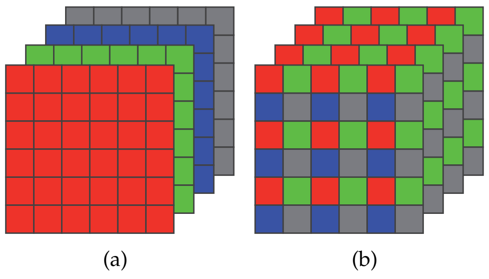


Figure 5.1: Flat 3D RGBA texture representation for uncompressed volumes (a) and for writing wavelet compressed coefficients

volumes. Here every slice consists four slices, one slice is in every channel of the RGBA texture. For the compressed approach this would mean that every RGBA texel would consist of elements from two wavelet blocks. Thus, to compute the operation result for one texel two $2 \times 2 \times 2$ wavelet blocks would have to be reconstructed. Therefore, to increase performance in compressed processing the texture is laid out, such that every RGBA texel holds one slice of a $2 \times 2 \times 2$ wavelet block.
A problem affecting compression rates comes along with the binary importance volume. As standard OpenGL does not offer a texture format for a binary datatype, at the moment the binary importance volume is stored in an eight bit datatype. However, due to the fact that GLSL is missing bitwise operations it does not make sense to store multiple 1bit values as a single 8bit texel. For the moment this means that the importance volume, when needed on the GPU, reduces compression rates substantially. It does not appear that textures using a binary datatype will be available sometime soon, yet nvidias CUDA [CUDA 2.0] allows bitwise operations which could be used to resolve this problem.

5.2.2 Visualization

To visualize the compressed data, without decompressing it as a whole, multiplanar reconstruction with on the fly fragment wise decompression was implemented. Contrary to the compression and decompression sequences, visualization is done by means of the GPU only. Therefore, as shown in figure 5.2 *Cascada* was extended by the SimpleCompVolumeRenderingSequence class and the SimpleCompVolumeRenderingGPU class which is called by the first. SimpleCompVolumeRenderingGPU mainly differs in the shader files which are loaded and the parameters given to the shaders, compared to its counterpart for non compressed rendering. The volume handling itself is done in the shader manager which is notified automatically to switch to compressed ren-

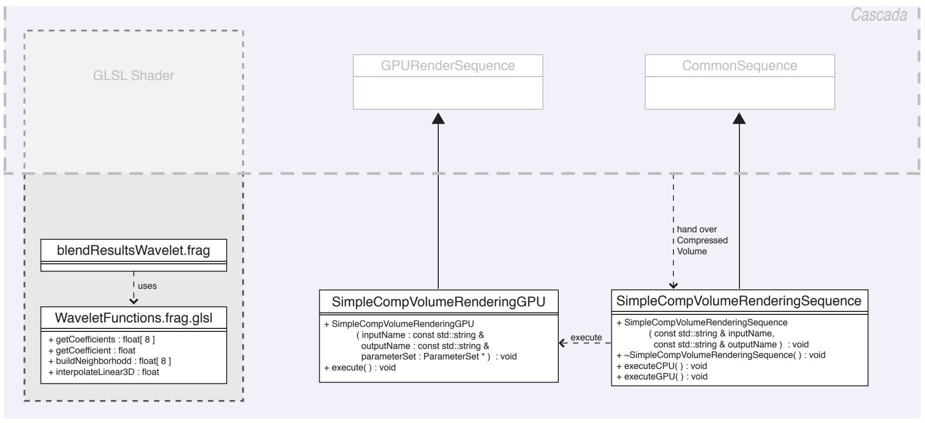

Cascada
GLSL Shader
GPURenderSequence
CommonSequence
blendResultsWavelet.frag
uses
WaveletFunctions.frag.glsl
+ getCoefficients : float[8]
+ getCoefficient : float
+ buildNeighborhodd : float[8]
+ interpolateLinear3D : float
SimpleCompVolumeRenderingGPU
+ SimpleCompVolumeRenderingGPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
execute
hand over Compressed Volume
SimpleCompVolumeRenderingSequence
+ SimpleCompVolumeRenderingSequence (const std::string & inputName, const std::string & outputName) : void
+ ~SimpleCompVolumeRenderingSequence( ) : void
+ executeCPU( ) : void
+ executeGPU( ) : void

Figure 5.2: The compressed rendering class structure

dering when a volume gets compressed.

For visualization the main difference between compressed and non compressed data is the fetch of the current voxel. Instead of a simple texture lookup the needed voxel has to be reconstructed from the compressed data in the compressed mode. This means the eight surrounding coefficients of the related $2 \times 2 \times 2$ wavelet block need to be fetched and the current voxel is received by the invert discrete wavelet transform of these coefficients.
The hardware native texture lookup features trilinear interpolation which is used to improve visual quality when rendering from non compressed data. For the compressed mode the interpolation has to be done manually. For fast rendering from the compressed data no interpolation is used by default. However, if desired the value for each fragment can be interpolated trilinear. Yet this requires reconstruction of up to seven neighboring wavelet blocks in the worst case, leading to significantly slower rendering.

The necessary functions to fetch a single coefficient or a complete wavelet block are swapped out into the separate WaveletFunctions fragment shader file, which has to be loaded alongside the blendResultsWavelet fragment shader. Hereby the functions can be reused in all fragment shaders, which require the lookup of coefficients from wavelet compressed data, i.e. the processing operations, which require local decompression (compare figure 5.3). The function to fetch a single coefficient follows the pseudo code previously shown in listing 4.1. The function to fetch the complete block just calls the function to fetch a single coefficient for all coefficients of the block the current position belongs to and returns a matrix with all eight entries.

5.2.3 Processing

Processing of the compressed data differs from visualization in the fact that it is a read and write operation, whereas for rendering only read access to the volume is needed. Thus a data structure for writing the result of the operation must be held ready. Not all operations need the complete data structure for processing, i.e. homogeneous point based operations, which can be applied in the compression domain can be executed with the average volume and the coefficient vector only. When no reordering of the coefficients is needed (all coefficients which initially were zero remain zero), which is the case for homogeneous addition and multiplication the results can be written directly into the average volume respective coefficient map. For all other operations the result must be written in a temporary coefficient volume containing all coefficients including these smaller than the threshold. The compressed coefficient map than has to be rebuild in a postprocessing step.

All illustrated operations which are presented in this thesis are shown in in the class diagram in figure 5.3. All operations were implemented in for CPU and GPU, thus for every operation a class for the sequence and one for the CPU and one for the GPU version are added to *Cascada* .

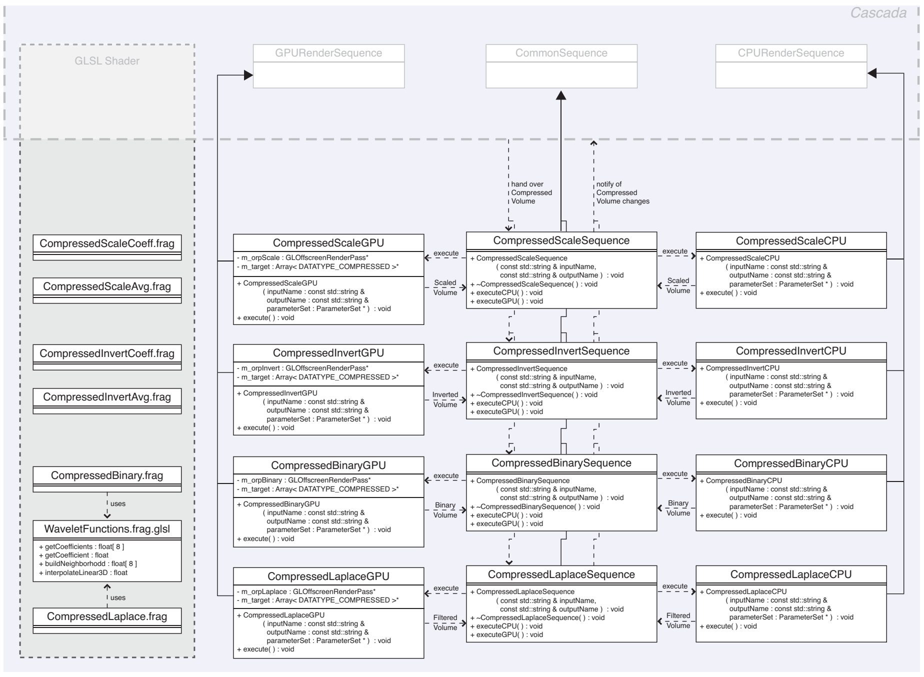

Cascada
GLSL Shader
GPURenderSequence
CommonSequence
CPURenderSequence
hand over Compressed Volume
notify of Compressed Volume changes
CompressedScaleCoeff.frag
CompressedScaleAvg.frag
CompressedInvertCoeff.frag
CompressedInvertAvg.frag
CompressedBinary.frag
uses
WaveletFunctions.frag.glsl
+ getCoefficients : float[8]
+ getCoefficient : float
+ buildNeighborhodd : float[8]
+ interpolateLinear3D : float
uses
CompressedLaplace.frag
CompressedScaleGPU
- m_orpScale : GLOffscreenRenderPass*
- m_target : Array< DATATYPE_COMPRESSED >*
+ CompressedScaleGPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
CompressedScaleSequence
+ CompressedScaleSequence (const std::string & inputName, const std::string & outputName) : void
+ ~CompressedScaleSequence( ) : void
+ executeCPU( ) : void
+ executeGPU( ) : void
CompressedScaleCPU
+ CompressedScaleCPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
execute
Scaled Volume
CompressedInvertGPU
- m_orpInvert : GLOffscreenRenderPass*
- m_target : Array< DATATYPE_COMPRESSED >*
+ CompressedInvertGPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
CompressedInvertSequence
+ CompressedInvertSequence (const std::string & inputName, const std::string & outputName) : void
+ ~CompressedInvertSequence( ) : void
+ executeCPU( ) : void
+ executeGPU( ) : void
CompressedInvertCPU
+ CompressedInvertCPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
Inverted Volume
CompressedBinaryGPU
- m_orpBinary : GLOffscreenRenderPass*
- m_target : Array< DATATYPE_COMPRESSED >*
+ CompressedBinaryGPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
CompressedBinarySequence
+ CompressedBinarySequence (const std::string & inputName, const std::string & outputName) : void
+ ~CompressedBinarySequence( ) : void
+ executeCPU( ) : void
+ executeGPU( ) : void
CompressedBinaryCPU
+ CompressedBinaryCPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
Binary Volume
CompressedLaplaceGPU
- m_orpLaplace : GLOffscreenRenderPass*
- m_target : Array< DATATYPE_COMPRESSED >*
+ CompressedLaplaceGPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
CompressedLaplaceSequence
+ CompressedLaplaceSequence (const std::string & inputName, const std::string & outputName) : void
+ ~CompressedLaplaceSequence( ) : void
+ executeCPU( ) : void
+ executeGPU( ) : void
CompressedLaplaceCPU
+ CompressedLaplaceCPU (inputName : const std::string & outputName : const std::string & parameterSet : ParameterSet *) : void
+ execute( ) : void
Filtered Volume

Figure 5.3: The class structure for compressed processing on the CPU and GPU

Processing in the Compression Domain

Two similar operations, the invert operator and the histogram spread operator, which operate directly on the compressed data are presented in this thesis. The main difference between the two operations is that the histogram spread operator incorporates a user adjustable parameter, while the invert operator is only dependent on the actual volume data. Meaning, that for the invert operator input and output volume can be the same volume, while for the histogram spread operator a second volume has to be created as the target for the operation to be able to access the original volume data when changing the parameter.

As shown in the previous section 5.1 both operations can then be applied as gray value transformations on the wavelet coefficients.
For the software implementation this simply means that in a loop over the average volume every average value is transformed with T_{avg} ($\cdot(-1) + g_{\text{max}}$ for invert and $\cdot\frac{(G-1)}{g_{\text{max}}-g_{\text{min}}} - \frac{g_{\text{min}}}{g_{\text{max}}-g_{\text{min}}}$ for the histogram spread) and in a second loop the entries of the coefficient map are transformed with T_{c} ($\cdot(-1)$ for invert and $\cdot\frac{(G-1)}{g_{\text{max}}-g_{\text{min}}}$ for the histogram spread). These two loops can be computed in multiple threads as there are no dependencies between items in the data fields.
Implementation on the GPU is a bit more complex. To load the data onto the GPU both data fields are converted to two dimensional RGBA textures. By using four channel RGBA textures, later on four values can be computed in a single step. Therefore, the closest square which fits one fourth of the corresponding data field is computed and a texture of this size filled with the data from the average volume or the coefficient map is created as input data. Additionally, an offscreen renderpass is created and the viewport is set to the same size. The results of the operation are then rendered into this viewport. For both operations each data field must be treated as a single operation, as not only the operations T_{avg} and T_{c} differ, but also the data fields are of different size. The shader programs for the operations themselves are rather simple one liners, as shown in listings 5.1 and 5.2.

Listing 5.1: The code for invertion

```
invert averages
{
    gl_FragColor = maxVal - texture2D( avgTex, gl_TexCoord[0].st );
}

invert coefficients
{
    gl_FragColor = - texture2D( coeffTex, gl_TexCoord[0].st );
}
```

Listing 5.2: GLSL code for histogram spread

```
// param = { g_min/(g_max-g_min), (G-1)/(g_max-g_min) }

histogram spread averages
{
    gl_FragColor = ( texture2D( avgTex, gl_TexCoord[0].st ) - param.x ) * param.y;
}

histogram spread coefficients
{
    gl_FragColor = ( texture2D( coeffTex, gl_TexCoord[0].st ) ) * param.y;
}
```

Processing with Block-wise Decompression

Binary thresholding and Laplace filtering were implemented as examples for operations which require block-wise decompression of the data. As a point based operations binary thresholding was classified as a class three operation in section 5.1 and the Laplace filter, as a linear filter falls in the category of class four operations. For both operations sequences as well as CPU and GPU off-screen render passes were added to *Cascada* (compare figure 5.3).
As denoted in section 5.2.1 operations requiring decompression are applied in two steps. In the first step, the block which is processed is decompressed followed by the application of the operation in the spatial domain and backtransformation into the wavelet domain. In the second step the resulting coefficient field has to be reordered.
To minimize cost of the decompression and recompression operations in software the complete eight-voxel wavelet block is decompressed, processed and retransformed. For binary thresholding this means that one indirect coefficient fetch is needed, instead of a simple array fetch per voxel. An additional seven additions are needed to transform each voxel in the block from the wavelet into spatial domain. After that the threshold operator is applied just like for the non compressed operation followed by retransformation into the wavelet domain which requires another seven additions.
On the GPU not all eight coefficients can be computed at once due to the fact that the result of every operation of a fragment shader is a color which offers at most four channels (in case of RGBA colors). However, to compute the four voxels in the spatial domain all eight coefficients of the corresponding block are needed as are all eight voxels of the block needed to compute the four resulting coefficients of the retransformation. Accordingly, the effective cost per voxel for decompression and retransformation into the wavelet representation doubles compared to the CPU version, meaning two coefficient fetches and 28 additions are needed per operation and voxel.

The application of Laplace filter proceeds very similar to the binary threshold operation. The only difference besides the operator itself is that the neighborhood of the current block has to be fetched. The Laplace filter is implemented as a $3 \times 3 \times 3$ mask. Only the six direct neighbors of the pivot and the pivot itself of the Laplace filter kernel do not equal zero. Such only the six directly neighboring voxels of each processed voxel contribute to the result and thus have to be fetched. Thinking in wavelet blocks this means additionally to the current block the six direct neighbors have to be fetched, however, for the $3 \times 3 \times 3$ filter mask only a $4 \times 4 \times 4$ voxel neighborhood has to be reconstructed for processing the current block. In software this results in seven coefficient fetches and 21 additional additions per voxel. On the GPU for the same reason as mentioned before this number effectively doubles to 14 fetches and 42 additions for reconstruction and recompression.

For a $5 \times 5 \times 5$ Laplace filter kernel the 12 neighboring blocks sharing one edge with the block being processed have to be fetched and reconstructed increasing the number of fetches to 19, respectively 38 and the number of additions to 70 per voxel on the CPU and 140 on the GPU. A general $3 \times 3 \times 3$ filter mask would require 27 fetches and 28 additions on the CPU and twice the amount on the GPU.

The indirect coefficient fetches are computationally very expensive, compared to simple array lookups/texture fetches, such a significant performance decrease has to be expected for these neighborhood based operations.

The actual code for both operations on the CPU as well as on the GPU are very similar to the code for the uncompressed versions only differing in the data structure being used and the operation itself being framed by the coefficient lookup and reconstruction beforehand and retransformation into the wavelet domain afterwards.

After the operation the reordering of the coefficient map is done just as in the wavelet compression, described in section 4.3 and listing 4.3.

Chapter 6

Results

In this chapter, the results of this work are presented. Section 6.1 compares visual quality of compressed datasets and lists the performance for compression of different data sets at multiple compression rates. In Section 6.2 the performance of the different processing operations is compared to the performance of the non compressed counterpart operations.

If not mentioned otherwise the performance was measured using the following system:

- Dual Processor PC, 2×4-core intel Xeon E5462 (2.8Ghz)
- 6GB fully buffered ECC RAM
- nVidia GeForce 8800 GT (g92)
- 512MB Video RAM
- Windows Vista x64
- *Cascada* compiled in 32bit mode

As mentioned in the implementation chapters (4.3 and 5.2) the main processing loops of all operations, including compression itself, were threaded with help of the OpenMP library. This is not the case for all operations in the spatial domain. Hence, for better comparability results for all CPU operations are shown with multithreading disabled alongside the threaded versions. Performance of the compression itself, however was always measured with threading enabled.

6.1 Compression

In the following, renderings of different datasets compressed at multiple rates are presented to compare visual quality. Additionally, difference volumes were created to get a better impression of the difference between compressed and non compressed datasets. All renderings were created with MevisLab [MVL] based on datasets created with *Cascada* by compressing and then decompressing the original datasets. The three datasets listed in table 6.1 were used for comparison. The compression times are averaged over 100 runs. The renderings of the difference volumes were created by direct volume rendering with a simple linear ramp as transfer function. Darker pixels represent higher deviation. For all comparisons the maximum and mean differences are given in the associated tables alongside the threshold used to reach the given compression rate and the time needed for compression. A compression rate of $1 : n$ means the original dataset is n-times the size of the compressed dataset.

Table 6.1: The Datasets

Dataset	Size	Max Value	Mean Value
Tooth	$256 \times 162 \times 256 \times 16$bit	1300	193.88
Orange	$256 \times 256 \times 64 \times 8$bit	228	18.83
Head	$512 \times 512 \times 460 \times 16$bit	4095	1081.8

The Tooth Dataset

The tooth dataset is an industrial micro CT scan, courtesy of GE Aircraft Engines, Evendale, Ohio, USA. With a maximum intensity of $1,300$ it consists of a quite small range, meaning the thresholds to reach the desired compression

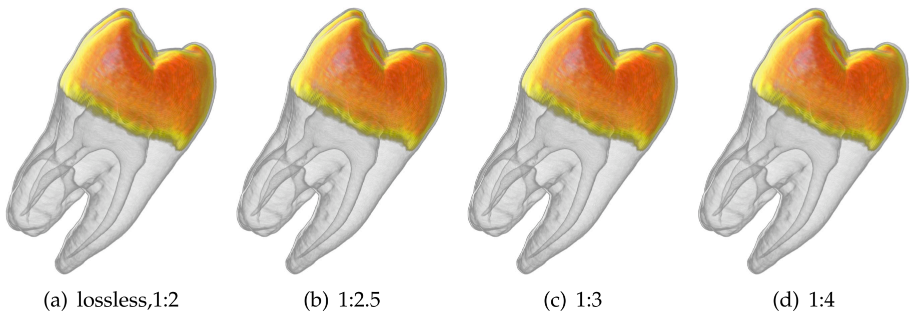
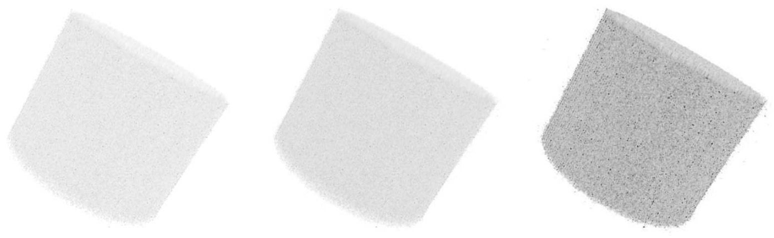

(a) lossless,1:2 (b) 1:2.5 (c) 1:3 (d) 1:4

Figure 6.1: The tooth dataset at different compression rates



(a) 1:2.5 (b) 1:3 (c) 1:4

Figure 6.2: Difference volumes of the tooth dataset

rates are the smallest of the two 16bit datasets presented in this section. About a third of a second is needed to compress the dataset.
The renderings shown in figure 6.1 seem nearly identical, in fact it is nearly impossible to find a difference just by visual examination. Figure 6.2 shows renderings of the difference volumes created by subtracting the compressed volume from the original volume and storing the absolute value of the result. The appended table shows the maximum (Max Diff) and mean value (Mean Diff) of the difference volumes at all compression rates. The difference is the sum of the initial values of all coefficients set to zero in a single block. Thus, the maximum difference usually is equal to seven times the threshold. The difference volumes seem to contain only homogeneous noise but as indicated by the maximum deviation which is only about 3.5% of the maximum value of the original volume at very low intensity. Thus, the impact on the actual renderings is very low.

Table 6.2: Compression Performance for the Tooth Dataset

Compression Rate	Time [ms]	Threshold	Max Diff	Mean Diff
lossless, $\approx 1:2$	359	0	0	0
$\approx 1:2.5$	359	2	14	0.76
$\approx 1:3$	328	3	21	1.28
$\approx 1:4$	312	5	35	2.19

The Orange Dataset

The orange dataset is an MRI scan, courtesy of Bill Johnston and Wing Nip of the Information and Computing Sciences Division, Lawrence Berkeley Laboratory, USA. The original dataset only uses eight bits per voxel so the range is notably smaller compared to the other two datasets presented here. Also,

Table 6.3: Compression Performance for the Orange Dataset

Compression Rate	Time [ms]	Threshold	Max Diff	Mean Diff
lossless, $\approx 1:1.1$	218	0	0	0
$\approx 1:2.5$	187	0.75	5	0.73
$\approx 1:3$	171	1.2	7	0.99
$\approx 1:4$	156	1.9	12	1.27

even though the absolute difference values are the smallest in relation to the volumes maximum and mean values these are the biggest of all presented datasets. This leads to the difference volumes shown in figure 6.4. Specifically at the compression rate of $1:4$ the noise mainly in the region of the orange is much more dense than at the lower compression rates. By visual examination small differences in the renderings shown in figure 6.3 can be seen, mainly in the region of the orange zest. However, it should be noted that a maximum difference of 12 is still only about five percent of the volumes range.

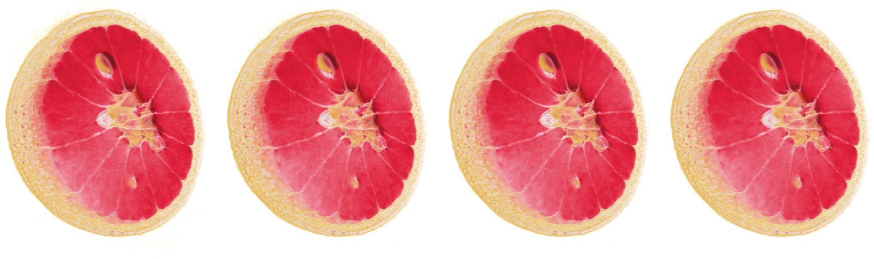

(a) lossless, 1:1.1 (b) 1:2.5 (c) 1:3 (d) 1:4

Figure 6.3: The orange dataset at different compression rates

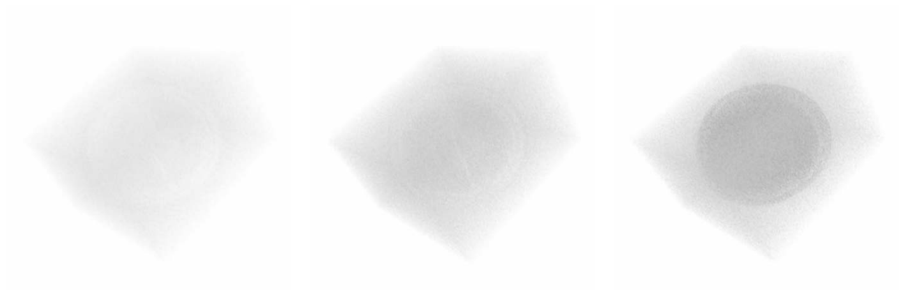

(a) 1:2.5 (b) 1:3 (c) 1:4

Figure 6.4: Difference volumes of the orange dataset

With only $4, 194, 304$ voxels the orange dataset is the smallest of all presented datasets, hence it is not surprising that compression time is the shortest, ranging from 156 to 218 milliseconds.

The Head Dataset

The Head dataset is a medical CT angiography taken from the Osirix sample image website [OsiriX]. Depending on the compression rate it takes approximately 3.5 to 5 seconds to compress the dataset. The dataset uses the complete range of the original 12bit with a maximum intensity value of $4, 095$. Even though the maximum value as well as the absolute mean value are the biggest of the presented datasets, the maximum and mean difference values are very similar to the values of the tooth dataset. Consequentially the renderings presented in figure 6.5 do not show any notable differences. When comparing the renderings of the difference volumes shown in figure 6.6 it seems a bit surprising that the renderings are so similar as for all compression rates the images are quite solid. However, at a compression rate of $1 : 4$ the maximum deviation is only about 1% of the maximum gray value in the original dataset meaning

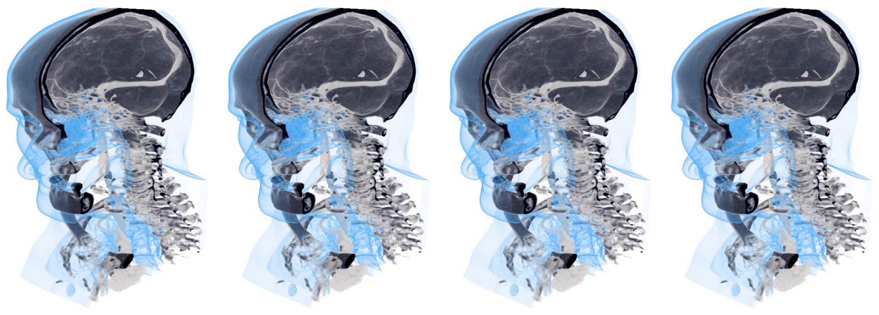

(a) lossless, 1:1.5 (b) 1:2.5 (c) 1:3 (d) 1:4

Figure 6.5: The head dataset at different compression rates

(a) 1:2.5 (b) 1:3 (c) 1:4

Figure 6.6: Difference volumes of the head dataset

Table 6.4: Compression Performance for the Head Dataset

Compression Rate	Time [ms]	Threshold	Max Diff	Mean Diff
lossless, $\approx 1:1.5$	5195	0	0	0
$\approx 1:2.5$	4305	1	9	0.47
$\approx 1:3$	3962	2	14	0.84
$\approx 1:4$	3604	6	42	1.86

even though a lot of the voxels in the compressed dataset differ from the original values the difference is so small that it is hardly notable.

Conclusion
Concluding, it can be said that all three presented test datasets can be compressed at least up to $1:4$ with minimal loss or even without affecting visual quality at all. A small threshold can also increase image quality. Considering the seven coefficients as high pass filtered voxel values it becomes clear that in homogeneous regions of the input data the coefficients will be zero, edges on the other hand will be mapped to large coefficients. Noise in the input data, however, will typically be transformed to small coefficients. Thus, when using a threshold homogeneous regions and edges will not be altered while noise will be removed by discarding small coefficients.
Typically MRI data contains much more noise than CT data. The more noise a dataset contains, the less coefficients will be zero, meaning a higher threshold will be necessary to reach the same compression rates compared to datasets with less noise. However, the higher the threshold the more likely actual features of the dataset will be affected by discarding coefficients. This behaviour can be seen when comparing the orange dataset to the other two datasets. In relation to the range of input volume the thresholds are much higher for the orange dataset than for the other two. Consequentially, the orange dataset is the only dataset where (even though very small) differences between the non compressed and compressed renderings are visible.
When comparing the time needed to compress the datasets it becomes clear that for the same compression rates the time increases approximately linearly with the size of the dataset. The fact that the time decreases when increasing compression rates is simply due to the fact that the second step of the compression pipeline, the reordering, has to be done on fewer voxels for higher compression rates and thus demands less time.

6.2 Processing

To compare performance of the processing operations the tooth dataset and the orange dataset shown in the previous section 6.1 were used. The tooth dataset was cropped to the tooth only without the surrounding air resulting in a volume of $120 \times 160 \times 100$ voxels. Four operations, invert (Inv), histogram spread (Hist), binary thresholding (Thresh) and a Laplace filter (Lap) were applied to the volume. All operations were applied by means of the CPU and GPU on the original datasets as well as the datasets compressed at different rates. As mentioned earlier, all CPU operations for compressed data are multithreaded and benefit strongly from the eight cores of the test system. As this is not the case for the operations working on non compressed data, for better comparability all operations were additionally benchmarked with multithreading disabled. This also gives good insight how well the parallelization works. The Laplace filter implemented on the CPU, although leading to plausible results with very small test datasets, performs very slowly. For the two small datasets presented here the filter takes multiple minutes until it yields a result. Even though it is expected that the neighborhood based operations perform slow it seems that the current implementation is flawed and as such, no performance measurements are listed in the corresponding tables. All remaining times are averaged over 100 runs.

Table 6.5 shows the performance of the CPU operations on the tooth dataset. As expected, the invert and histogram spread operators which both are applied directly on the wavelet coefficients perform better than the corresponding operators for the non compressed data. For the cutout only about 14% of the coefficients are discarded. Thus the speedup by the factor three respectively four can be explained by the fewer values which have to be partially processed. As shown in equation 5.13 and 5.20, the high pass filtered coefficients which are about six sevenths of the remaining coefficients are multiplied by a constant, whereas the original operations each involve an additional addition. Thus, the main speedup seems to be caused by the more simple operations on the high pass filtered coefficients. Consequentially, with higher compression rates the performance does not scale linearly, as the quota of the low pass filtered average coefficients which are processed with the same operator as the original values, gets larger with larger.
The threshold operator in the spatial domain is in the same class as the first two operators. In the wavelet form, however decompression is required. Depending on the compression rate the operations requires between 13 and 40 times the time of the operation in the spatial domain. The most expensive part of the

Table 6.5: Processing Times for the Tooth Dataset (CPU)

Mode	Inv [ms]	Hist [ms]	Thresh [ms]	Lap [ms]
Singlethreaded				
Uncompressed	45.6	68.5	31.47	271.8
Lossless Compressed	15.6	18.1	1235.4	n/a
Compressed @ 1 : 2.5	10.0	12.5	797.1	n/a
Compressed @ 1 : 3	7.8	10.3	623.9	n/a
Compressed @ 1 : 4	5.5	8.3	404.0	n/a
Multithreaded				
Lossless Compressed	2.3	2.5	337.0	n/a
Compressed @ 1 : 2.5	1.3	1.6	205.9	n/a
Compressed @ 1 : 3	1.1	1.4	168.5	n/a
Compressed @ 1 : 4	0.6	0.9	120.1	n/a

decompression is the coefficient lookup. Consequentially, at higher compression rates where fewer coefficients have to be fetched the performance scales comparably to the performance of the invert and histogram spread operators. All three operations benefit from multithreading. However, where the invert and histogram spread operations use the eight cores to full capacity, the thresholding only gains a speedup of about $4\times$.
The GPU operations, listed in table 6.6, behave a little different compared to the CPU versions. Whereas the uncompressed versions of the invert and histogram spread operators outperform the CPU versions by the factor 17 respectively 52, the compressed versions perform approximately on the same level on the GPU and CPU (single threaded). This means that on the GPU, in contrast to the CPU, the compressed versions perform worse than their non compressed

Table 6.6: Processing Times for the Tooth Dataset (GPU)

Mode	Inv [ms]	HistS [ms]	Thresh [ms]	Lap [ms]
Uncompressed	2.7	1.3	1.4	24.0
Lossless Compressed	15.4	15.4	46.0	582.8
Compressed @ 1 : 2.5	9.4	8.3	42.6	451.8
Compressed @ 1 : 3	7.0	6.6	39.6	417.8
Compressed @ 1 : 4	5.3	4.4	32.3	334.1

counterparts. This is caused by the data handling of these two versions. The textures for these two operations differ from the textures used by default and thus some overhead is caused by preparing the data for these operations which seems to affect performance significantly.
As expected, just like the CPU version the GPU version of the binary threshold as well as the Laplace filter perform significantly worse in the compression domain compared to the operators in the spatial domain. The binary threshold performs about 30 times worse in the compression domain at lossless compression, the Laplace filter about 25 times worse. This is in the same range as the CPU versions. However, on the GPU the speedup due to higher compression rates is not as big as on the CPU.

Tables 6.7 and 6.8 show the performance of the same operations on the orange dataset. The orange dataset is about twice the size of the cutout from the tooth dataset. The execution of all operations in the spatial domain as well as the invert and histogram spread operations in the compression domain takes about twice the time as on the tooth dataset. This does not transfer to the binary threshold and Laplace operators because both operations recompress the data after the the application of the operation itself. Thresholding and the Laplace filter significantly alter the structure of the volumes and therefore the composition of the coefficients. As the performance of the compression step heavily depends on the number of coefficients which is altered by applying the operations, the results can not directly be compared between different volumes.
Other than this, all conclusion drawn for the tooth dataset are valid for the orange dataset as well. With higher compression rates the invert and histogram spread operations perform significantly better and the slower GPU versions indicate problems with the data handling. The speedup, due to higher compression rates for the thresholding operation and the Laplace filter, is not as great as for the two other operatons. The multiprocessor speedup is with approximately $8\times$ for the invert and histogram spread operations and $4\times$ for the binary threshold on the same level as for the tooth dataset.

Table 6.7: Processing Times for the Orange Dataset (CPU)

Mode	Inv [ms]	Hist [ms]	Thresh [ms]	Lap [ms]
Singlethreaded				
Uncompressed	100.2	164.1	67.7	302.2
Lossless Compressed	32.0	37.6	2035.5	n/a
Compressed @ $1:2.5$	13.6	19.8	854.5	n/a
Compressed @ $1:3$	10.5	16.4	673.9	n/a
Compressed @ $1:4$	8.3	14.4	570.9	n/a
Multithreaded				
Lossless Compressed	4.1	4.5	552.5	n/a
Compressed @ $1:2.5$	1.7	2.2	263.6	n/a
Compressed @ $1:3$	1.3	1.9	224.6	n/a
Compressed @ $1:4$	0.9	1.6	198.1	n/a

Table 6.8: Processing Times for the Orange Dataset (GPU)

Mode	Inv [ms]	Hist [ms]	Thresh [ms]	Lap [ms]
Uncompressed	4.4	5.2	2.81	73.6
Lossless Compressed	28.6	28.9	172.2	910.2
Compressed @ $1:2.5$	10.8	11.5	140.4	630.3
Compressed @ $1:3$	8.6	9.2	121.4	517.6
Compressed @ $1:4$	6.2	6.9	114.2	446.2

Chapter 7

Summary and Conclusion

This chapter summarizes the results of this thesis and gives an outlook for future work including enhancement of compression rates, linear filtering in the compression domain and hierarchical processing.

Within the scope of this thesis solutions to directly process compressed volume data were developed. Therefore, a wavelet transform based compression technique for volume data was designed which allows comparatively fast random access to single voxels at acceptable compression rates. Based on this compression technique direct volume visualization via multi-planar reconstruction was implemented. Moreover, image processing operations were classified by their applicability in the wavelet domain and some operations were implemented both on the CPU and GPU exemplarily.
In a nutshell, this thesis presents the following achievements:

- A wavelet compression scheme for volume data
- Visualization directly from wavelet compressed volume data
- Processing of compressed volume data directly in the compression domain (for some operations)
- Processing of compressed volume data with local decompression for operations not possible directly in the compression domain with minimal expansion of the memory footprint

7.1 Compression

As shown in section 6.1, the implemented wavelet compression technique delivers very good visual quality. With CT datasets hardly any differences between renderings of non compressed and compressed dataset can be found by visual examination. The higher noise level of typical MRI datasets makes it necessary to use larger thresholds to achieve compression rates similar to the CT datasets. This, however, leads to a loss in image quality. Yet, at the presented compression rates only slight deviations are notable.

The current implementation is limited to single level decomposition and thus a theoretical maximum compression rate of $1 : 8$ only. A compression rate of $1 : 8$, however, woud mean that all high pass filtered coefficients are set to zero. This would basically result in a downscaled version of the original volume. As shown in section 6.1, at compression rates of $1 : 4$ excellent visual quality can be reached. Depending on the dataset compression up to $1 : 6$ has proven feasible.

For datasets within the range of several hundred million voxels compression takes several seconds. Thus, it does not make sense to compress a volume only to benefit from the performance increase of the operations applicable in the compression domain. The time, however, is acceptable when it is essential to reduce the memory footprint of the volume.

7.2 Processing

The goal of directly processing compressed volume data was reached. Some operations were transferred directly into the compression domain, whereas for others the detour of block- or voxel-wise decompression had to be used.

The operations which could be ported into the compression domain, especially on the CPU, perform really well. It is not only possible to keep the memory footprint of the datasets small during processing, the operations actually perform better than in the spatial domain. On the GPU, however, the operations do not perform as well as on the CPU. It should be possible to improve performance significantly by optimizing the data handling for these operations and bringing the data structures for all operations and visualization in line.
Whenever decompression is needed performance drops significantly compared to processing in the spatial domain. Decompression and recompression constitute a significant overhead. Besides porting operations directly into the compression domain, whenever possible, this overhead can only be reduced by

optimizing the data structure to better exploit caching, etc.

Additionally, simple visualization based on multi-planar reconstruction directly from the wavelet compressed data was implemented allowing real time to interactive rendering speeds, depending on the size of the dataset. Visual quality, however, is not as good as for uncompressed data as fast rendering only works with nearest neighbor interpolation, while for uncompressed data hardware native trilinear interpolation is used. Even though trilinear interpolation for the compressed visualization is implemented framerates drop significantly when activated.

7.3 Future Work

There are several possibilites to improve the compression rates of the presented wavelet compression scheme. At this point there is no quantization applied to the coefficients at all. By scaling the coefficients to fit into a smaller datatype the size of the coefficient map could be reduced considerably. In the current implementation the coefficients are stored in the same 16bit datatype as the input volume, however, typically only a small part of the available range is used and the major part of the coefficients is very small. Scaling the coefficients to fit into an eight bit datatype would cut the size of the coefficient map by half. Yet the maximum possible compression rate would remain $1 : 8$. Additionally, one has to be very cautious with the larger coefficients, as these provide important detail to the image.
The fixed maximum compression rate of $1 : 8$ can only be raised by decreasing the size of the average volume. This can be done by applying the wavelet transform onto the average volume and compressing the resulting subbands (this is described in section 2.2.1 as multiple level wavelet transform). For every level the maximum possible compression rate will be increased by the factor eight. However, raising the level also increases the number of computations necessary to reconstruct a voxel and such would have negative impact on the rendering and processing speed when reconstruction is needed. However, the performance of class one operations like invert and histogram spread, which work directly on the coefficients, would benefit from higher compression levels.

As shown in section 6.2, operations demanding decompression perform quite poorly. For linear filters decompression could be made unnecessary if translation were possible in the wavelet domain. In section 3.2 it is shown how to apply linear filters as a combination of homogeneous multiplication, image addition and a shift operation. In the currently implemented wavelet transform, however, shifting or translating the image is not possible directly in the transformation domain. Yet, as presented in section 3.2, several shift invariant wavelet transforms exist. Extending the current framework by a shift invariant wavelet transform would make linear filtering directly in the compression domain possible, making decompression only necessary for very few operations. Applying linear filters in such way is expected to increase performance markedly. Consequentially, when extending the work of this thesis a shift invariant wavelet transform should be one of the top priority tasks.

A very interesting practical application of the presented work could be so-called hierarchical processing. Many tasks when working with volume data, wether visualization or processing, require user input. For example, in visualization the user might want to modify the transfer function to examine different features of the dataset or when applying the binary threshold operator the user interactively wants to change the threshold and follow the modifications live on screen. For good usability user inputs should at best lead to an instant feedback on the screen. For large datasets, however, not all operations can be applied in real time. When using wavelet compression to store the volume every level in the compression hierarchy implicitly contains a downscaled version of the original dataset, the average volume. Whenever the user is interacting with the system the average volume can be used to generate previews of the results, based on which the user can decide wether he is satisfied with the chosen parameters or not. If yes the system can apply the operation with the chosen parameters to the entire volume.

Acknowledgements

This work has been carried out at the working group Computer Graphics at the Institute for Computational Visualistics of the University of Koblenz.

I would like to express my sincere gratitude and appreciation to all who made this thesis possible. First and foremost to my adviser Matthias Raspe for his expert guidance. Thanks for encouraging me in choosing this challenging topic for my diploma thesis, for all the helpful advices, for the long (GPU) debugging sessions and for proof reading large parts of this thesis. Further thanks goes to Stefan Müller for arising my interest in computer graphics.
Additionally sincere thanks go to Dina and Stephan for proof reading this thesis as well as Christian and Michael for their input regarding illustrations and especially 3D the models used in this thesis.

A special thanks is reserved for my parents for their financial aid, but especially for always believing in me over now nearly six long years of my studying. Thank you!

Finally i have to thank Sarah for her sheer infinite patience with me. It is eventually done!

Bibliography

DEBEBE ASEFA, DINESH MITAL, SYED HAQUE & SHANKAR SRINIVASAN, 2006, *Restoration of fMRI Signal using Wiener Filter in a Wavelet Domain*, in The Internet Journal of Medical Informatics, vol. 3, no. 2.

MICHAEL F. BARNSLEY & LYMAN P. HURD, 1993, *Fractal Image Compression*, A. K. Peters, Ltd., Natick, MA, USA.

ANDREW P. BRADLEY, 2003, *Shift-invariance in the Discrete Wavelet Transform*, in DICTA'03: Digital Image Computing: Techniques and Applications, pp. 29–38.

CCIR, 1990, *Encoding Parameters of Digital Television for Studios*, International Radio Consultative Committee, in Recommendations of the CCIR, vol. 601.

KELBY K. CHAN, CHRISTINA C. LAU, KEH-SHIH CHUANG & CRAIG A. MORIOKA, 1991, *Visualization and Volumetric Compression*, in Medical Imaging V: Image Capture, Formatting, and Display, vol. 1444, pp. 250–255, SPIE.

S. GRACE CHANG, BIN YU & MARTIN VETTERLI, 2000, *Adaptive Wavelet Thresholding for Image Denoising and Compression*, in IEEE Transactions on Image Processing, vol. 9, no. 9, pp. 1532–1546.

TZI-CKER CHIUEH, CHUAN-KAI YANG, TAOSONG HE, HANSPETER PFISTER & ARIE KAUFMAN, 1997, *Integrated Volume Compression and Visualization*, in VIS '97: Proceedings of the 8th conference on Visualization '97, pp. 329–ff., IEEE Computer Society Press, Los Alamitos, CA, USA.

HUNG-KAI CLIFF CHOI & CHOK-KI CHAN, 1996, *Motion Classified 3D Vector Quantization for Sequence Coding*, in Proceedings of International Conference on Image Processing '96, vol. 3, pp. 275–278.

WAYNE O. COCHRAN, JOHN C. HART & PATRICK J. FLYNN, 1996, *Fractal Volume Compression*, in IEEE Transactions on Visualization and Computer Graphics, vol. 2, no. 4, pp. 313–322.

ISRAEL COHEN, SHALOM RAZ & DAVID MALAH, 1997, *Orthonormal Shift-invariant Wavelet Packet Decomposition and Representation*, in Signal Process., vol. 57, no. 3, pp. 251–270.

JAMES W. COOLEY & JOHN W. TUKEY, 1965, *An Algorithm for the Machine Calculation of Complex Fourier Series*, in Mathematics of Computation, vol. 19, no. 90, pp. 297–301.

PAMELA C. COSMAN, KAREN L. OEHLER, EVE A. RISKIN & ROBERT M. GRAY, 1993, *Using Vector Quantization for Image Processing*, in Proceedings of the IEEE '93, vol. 81, pp. 31326–13419, IEEE Computer Society, Washington, DC, USA.

CUDA 2.0, *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Ver. 2.0*, nvidia, http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf visited August, 10th, 2008.

DAVID L. DONOHO & IAIN M. JOHNSTONE, 1995, *Adapting to Unknown Smoothness via Wavelet Shrinkage*, in Journal of American Statistical Association, vol. 90, no. 432, pp. 1200–1224.

ANDREW DORRELL, 1996, *Image Processing in the Block-DCT Domain: Fast Techniques and Applications*, published through http://www.pnc.com.au/~enviscom/research/dct-processing.pdf (no longer available).

ANDREW DORRELL & DAVID LOWE, 1995, *Fast Image Operations in Wavelet Spaces*, in Digital Image Computing, Techniques and Applications, Brisbane, Australia.

IDDO DRORI & DANI LISCHINSKI, 2003, *Fast Multiresolution Image Operations in the Wavelet Domain*, in IEEE Transactions on Visualization and Computer Graphics, vol. 9, pp. 395–411.

S. DUNNE, S. NAPEL & B. RUTT, 1990, *Fast Reprojection of Volume Data*, in Proceedings of the First Conference on Visualization in Biomedical Computing, 1990., pp. 11–18.

KLAUS ENGEL, MARKUS HADWIGER, CHRISTOF REZK-SALAMA, JOE M. KNISS & DANIEL WEISKOPF, 2006, *Real-time Volume Graphics*, A. K. Peters, Ltd., Natick, MA, USA.

UGO ERRA, 2005, *Toward Real Time Fractal Compression Using Graphics Hardware*, in ISVC '05: Proceedings of the International Symposium on Visual Computing, vol. 3804, pp. 723–728.

ALLEN GERSHO & ROBERT M. GRAY, 1992, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers.

M. GOLDBERG & HUIFANG SUN, 1986, *Image Sequence Coding Using Vector Quantization*, in IEEE Transactions on Communications, vol. 34, no. 7, pp. 703–710.

ROBERTO GROSSO, THOMAS ERTL & JOACHIM ASCHOFF, 1996, *Efficient Data Structures for Volume Rendering of Wavelet-Compressed Data*, in WSCG '96 - The Fourth International Conference in Central Europe on Computer Graphics and Visualization, vol. I, pp. 103–112, University of West Bohemia, Plzen.

HAITAO GUO & C. S. BURRUS, 1996, *Convolution Using the Undecimated Discrete Wavelet Transform*, in ICASSP '96: Proceedings of the Acoustics, Speech, and Signal Processing, 1996. on Conference Proceedings., 1996 IEEE International Conference, pp. 1291–1294, IEEE Computer Society, Washington, DC, USA.

STEFAN GUTHE, MICHAEL WAND, JULIUS GONSER & WOLFGANG STRASSER, 2002, *Interactive Rendering of Large Volume Data Sets*, in VIS '02: Proceedings of the conference on Visualization '02, pp. 53–60, IEEE Computer Society, Washington, DC, USA.

MATTHIAS HOPF & THOMAS ERTL, 1999, *Hardware-Based Wavelet Transformations*, in Workshop of Vision, Modelling, and Visualization (VMV '99), pp. 317–328, infix.

INSUNG IHM & SANGHUN PARK, 1999, *Wavelet-Based 3D Compression Scheme for Interactive Visualization of Very Large Volume Data*, in Computer Graphics Forum, vol. 18, no. 1.

JPEG, *The JPEG Standard*, Joint Photographic Experts Group, `http://jpeg.org/jpeg/index.html` visited June, 11th, 2008.

JPEG 2000, *The JPEG 2000 Standard*, Joint Photographic Experts Group, `http://jpeg.org/jpeg2000/index.html` visited June, 11th, 2008.

LAKHWINDER KAUR, SAVITA GUPTA & R. C. CHAUHAN, 2002, *Image Denoising Using Wavelet Thresholding*, in The 3rd Indian Conference on Computer Vision, Graphics and Image Processing '02.

TAE-YOUNG KIM & YEONG GIL SHIN, 1999, *An Efficient Wavelet-Based Compression Method for Volume Rendering*, in PG '99: Proceedings of the 7th Pacific Conference on Computer Graphics and Applications, p. 147, IEEE Computer Society, Washington, DC, USA.

DONALD E. KNUTH, 1985, *Dynamic Huffman Coding*, in Journal of Algorithms, vol. 6, no. 2, pp. 163 – 180.

THOMAS LEHMANN, WALTER OBERSCHELP, ERICH PELIKAN & RUDOLF REPGES, 1997, *Bildverarbeitung für die Medizin*, Springer-Verlag, Berlin.

L. LIPPERT & MARKUS. H. GROSS, 1995, *Fast Wavelet Based Volume Rendering by Accumulation of Transparent Texture Maps*, in Computer Graphics Forum, vol. 14, pp. 431–444.

TOM MALZBENDER, 1993, *Fourier Volume Rendering*, in ACM Trans. Graph., vol. 12, no. 3, pp. 233–250.

SHIGERU MURAKI, 1993, *Volume Data and Wavelet Transforms*, in Computer Graphics and Applications, vol. 13, no. 4.

MVL, *MeVisLab, Medical Image Processing and Visualization*, meVisLab download at http://www.mevislab.de/index.php?id=4 visited August, 10th, 2008.

PAUL NING & LAMBERTUS HESSELINK, 1992, *Vector Quantization for Volume Rendering*, in VVS '92: Proceedings of the 1992 workshop on Volume visualization, pp. 69–74, ACM, New York, NY, USA.

PAUL NING & LAMBERTUS HESSELINK, 1993, *Fast Volume Rendering of Compressed Data*, in VIS '93: Proceedings of the 4th conference on Visualization '93, pp. 11–18.

OpenMP, *OpenMP Application Program Interface*, OpenMP Architecture Review Board, http://www.openmp.org/mp-documents/spec30.pdf visited July, 17th, 2008.

OsiriX, *DICOM Sample Image Sets*, The OsiriX Foundation, http://pubimage.hcuge.ch:8080/ visited June, 23th, 2008.

JAVIER PORTILLA, VASILY STRELA, MARTIN J. WAINWRIGHT & EERO P. SIMONCELLI, 2002, *Image Denoising Using Gaussian Scale Mixtures in the Wavelet Domain*, Tech. Rep. TR2002-831, Computer Science Technical Report, Courant Inst. of Mathematical Sciences, New York University.

SHEN-EN QIAN, ALLAN B. HOLLINGER, DAN WILLIAMS & DAVINDER MANAK, 1995, *3D Data Compression System Based on Vector Quantization for Reducing the Data Rate of Hyperspectral Imagery*, in Proceedings of International Conference on Applications of Photonics Technology, pp. 641–654.

MATTHIAS RASPE, GUIDO LORENZ & STEFAN MÜLLER, 2008a, *Evaluating the Performance of Processing Medical Volume Data on Graphics Hardware*, in Bildverarbeitung für die Medizin, pp. 427–431.

MATTHIAS RASPE, GUIDO LORENZ & STEPHAN PALMER, 2008b, *Hierarchical and Object-Oriented GPU Programming*, in Computer Graphics International Conference, pp. 333–337.

MATTHIAS RASPE & STEFAN MÜLLER, 2007, *Using a GPU-based Framework for Interactive Tone Mapping of Medical Volume Data*, in SIGRAD 2007. The Annual SIGRAD Conference, Special Theme: Computer Graphics in Healthcare, vol. 28, no. 3, pp. 3–10.

FLEMMING FRICHE RODLER, 1999, *Wavelet Based 3D Compression with Fast Random Access for Very Large Volume Data*, in PG '99: Proceedings of the 7th Pacific Conference on Computer Graphics and Applications, p. 108, IEEE Computer Society, Washington, DC, USA.

J. K. ROGERS, 1998, *Robust Wavelet Zerotree Image Compression with Fixed-Length Packetization*, in DCC '98: Proceedings of the Conference on Data Compression, p. 418, IEEE Computer Society, Washington, DC, USA.

JENS SCHNEIDER & RÜDIGER WESTERMANN, 2003, *Compression Domain Volume Rendering*, in Proceedings of the 14th IEEE Visualization 2003, p. 39, IEEE Computer Society, Washington, DC, USA.

PETER SCHRÖDER, 1996, *Wavelets in Computer Graphics*, in Proceedings of the IEEE, vol. 84, pp. 615–625.

N. SEBE, C. LAMBA & M.S. LEW, 2002, *An Overcomplete Discrete Wavelet Transform for Video Compression*, in ICME '02. Proceedings. 2002 IEEE International Conference on Multimedia and Expo, vol. 1, pp. 641–644.

BO SHEN & ISHWAR K. SETHI, 1996, *Convolution-Based Edge Detection for Image/Video in Block DCT Domain*, in Journal of Visual Communication and Image Representation, vol. 7, no. 4, pp. 411–423.

BO SHEN, ISHWAR K. SETHI & V. BHASKARAN, 1998, *DCT Convolution and its Application in Compressed Video Editing*, in IEEE Transactions on Circuits and Systems for Video Technology, vol. 8, no. 8, pp. 947–952.

MARK J. SHENSA, 1992, *The Discrete Wavelet Transform: Wedding the À Trous and Mallat Algorithms*, in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 40, no. 10, pp. 2464–2482.

BRIAN C. SMITH & LAWRENCE A. ROWE, 1993, *Algorithms for Manipulating Compressed Images*, in Computer Graphics and Applications, vol. 13, no. 5, pp. 34–42.

BRIAN C. SMITH & LAWRENCE A. ROWE, 1996, *Compressed Domain Processing of JPEG-encoded Images*, in Real-Time Imaging, vol. 2, pp. 3–17.

VASILY STRELA, 2000, *Denoising via Block Wiener Filtering in the Wavelet Domain*, in Third European Congress of Mathematics, Progress in Mathematics, Birkhauser Verlag, Barcelona.

OSCAR VALERO, 2005, *On Banach Fixed Point Theorems for Partial Metric Spaces*, in Applied General Topology, vol. 6, pp. 229–240.

GREGORY K. WALLACE, 1991, *The JPEG Still Picture Compression Standard*, in Commun. ACM, vol. 34, no. 4, pp. 30–44.

BEN WEISS, 2006, *Fast Median and Bilateral Filtering*, in ACM Transactions on Graphics, vol. 25, no. 3, pp. 519–526.

STEPHEN WELSTEAD, 1999, *Fractal and Wavelet Image Compression Techniques*, SPIE - The International Society for Optical Engineering.

RÜDIGER WESTERMANN, 1994, *A Multiresolution Framework for Volume Rendering*, in VVS '94: Proceedings of the 1994 symposium on Volume visualization, pp. 51–58, ACM, New York, NY, USA.

NORBERT WIENER, 1949, *Extrapolation, Interpolation, and Smoothing of Stationary Time Series*, Technology Press of MIT, Cambridge, MA, USA.