TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Memory Allocation Strategies for Large Volumetric Data-Sets

ausgeführt am

Institut für Computergrafik und Algorithmen
der Technischen Universität Wien

unter Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Dipl.-Inform. Sören Grimm

durch

Michael Knapp
Allhangstrasse 26
3001 Mauerbach

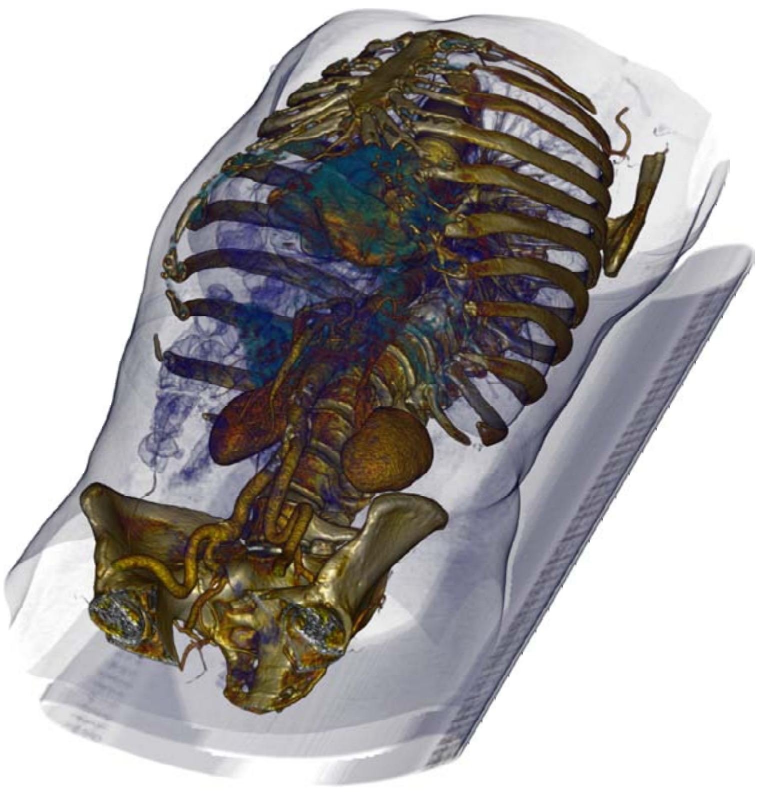
7. Dezember 2004
Datum

Unterschrift

Michael Knapp

Memory Allocation Strategies for Large Volumetric Data-Sets

Diploma Thesis



supervised by

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller

Dipl.-Inform. Sören Grimm

Institute of Computer Graphics and Algorithms
Vienna University of Technology

Abstract

Since the development of medical three dimensional imaging devices in the 1970s, volumetric data processing has tremendously gained in importance. With the growing size of the data-sets, exhausting the capabilities of the hardware of its time, methods for efficient volumetric data processing have been always a hot topic. In this diploma thesis two approaches for processing large volumetric data-sets are presented. Both approaches utilize a block volume for storing the data. Further data compression and out-of-core processing are incorporated. Efficiency is achieved by processing only the required portion of data while omitting the non-related data having no effect on the intended result of the algorithm. This is supported by utilization of the knowledge about access patterns of the algorithms. Also methods for optimizing the efficiency by exploiting architectural properties of the computer hardware are presented.

Kurzfassung

Seit der Einführung von medizinischen dreidimensionalen bildgebenden Verfahren in den 1970er Jahren gewann die Verarbeitung von volumetrischen Daten massiv an Bedeutung. Mit der wachsenden Größe der Daten, die die Kapazitäten der Hardware ihrer Zeit laufend auslastete, waren Methoden für die effiziente Verarbeitung von volumetrischen Daten immer wichtig. In dieser Diplomarbeit werden zwei Methoden für die Verarbeitung großer volumetrischer Daten vergestellt. Beide Methoden verwenden eine geblockte Datenstruktur für die Verwaltung der Daten. Weiters werden Datenkompression und *Out-of-Core*-Methoden integriert. Effizenz wird durch das Verarbeiten von ausschließlich für das Ergebnis eines Algorithmus relevanten Daten erreicht, wobei nicht relevante Daten, die keinen Effekt aus das zu erzielende Ergbenis haben, von vornherein ausgeschlossen werden. Die Effizienz dieser Methoden wird unter Beachtung der Zugriffsmuster von Algorithmen verbessert. Weiters werden Methoden zur weiteren Effizienzsteigerung vorgestellt, die die architekturiellen Eigenheiten der Computer Hardware ausnützen.

Contents

Chapter 1

Introduction

Since the introduction of the computer tomography scanner by G.N. Hounsfield in the early 1970s processing of volumetric data tremendously gained in importance. Computer tomography is the reconstruction of cross-sectional images from multiple one dimensional scans along lines from different angles on a plane. The reconstruction of a series of such cross-sectional dimensional images results in a three dimensional, i.e. volumetric, data-set. The essential steps contributing to the computer tomography technology as it is used today had happened already in the decades before.
In 1895, the so-called X-rays were discovered by W.C. Röntgen. The X-rays, high energetic electro-magnetic radiation with wave-lengths in the picometer range, are able to permeate compact structures without being dispersed too much. Depending on the material of the structures, the X-rays are absorbed in varying intensity.
In the 1960s A.M. Cormack, physicist at the Groote Schuur Hospital in Cape Town / South Africa, developed a mathematical model for the absorption distribution of X-rays in the human body.
A mathematical model of the reconstruction of cross-sectional images from one dimensional transmission measurements was introduced by J.H. Radon in 1917.
In the early 1970s, these three relevant contributions have been incorporated and realized in hardware by G.N. Hounsfield. In 1972 the first clinical diag-

nostics based on computer tomography images were made. A few years later several thousand computer tomography scanners were in use worldwide. The first scanners had a slice size of 80x80 pixel and a thickness of 13 millimeter. *Slice* is a synonym for the reconstructed cross-sectional image. For example scanning the human head would result in approximately 150 slices.
In the early days of computer tomography, it was sufficient to examine a single slice, in order to conduct a medical diagnosis. At this time, scanning a single slice took 300 seconds. Later, with the reduction of the scanning time down to a few seconds, the acquisition of a series of slices at once became common. It was sufficient to examine the scanned data slice by slice like ordinary x-ray images.
With the up-come of personal workstations in the 1980s, digital processing of the computer tomography data became more and more popular. Over the years, the computer tomography technology improved: 1990 the spiral computer tomography was introduced and the scanner resolution increased. Now, the slice resolution of modern scanners is at 1024x1024 with a slice thickness of 0.5 millimeter. A scan usually results in a series of several hundred slices.
This is the point, where volumetric data processing becomes indispensable: Examining hundreds of slices one by one is not feasible anymore. The introduction of an additional processing step between the data acquisition and the visualization for diagnostic purpose is necessary. This processing step is referred to as *volumetric data processing.*
The past years have shown, that the size of the volumetric data-sets has increased as fast as the CPU processing power and memory size. During this time period the data density, i.e. the portion of data acquired for the same volume, increased by 100, while the CPU processing power increased by a factor of 1000, the memory bandwidth increased by a factor of 500 and the mass storage interface bandwidth increased by a factor of 40. Thus efficient data handling is an important issue of volumetric data processing. Today the common silicon technology reached its physical limits, whereas in the computer tomography area there is still much potential for further increase of the data quantity: the recent introduction of 16-slice scanners, better de-

tectors and the emergence of multi-modal data let expect a further increase of the amount of data to be processed.
Common computer architecture is sufficient for average applications, however its complex architecture cannot optimally serve applications with exceptional resource requirements. Such exceptional requirements are needed in the field of volumetric data processing, where very efficient data transfer capabilities in all components, i.e. CPU, memory and mass storage devices, are required. The memory requirements for this type of applications regularly tend to exceed the available memory resources, so that the data often must be processed directly from mass storage devices, which are very slow compared to the memory or caches.
For optimally utilizing the data transfer capabilities of these components, the development of specific algorithms exploiting the architectural advantages of the common computer hardware is necessary.

Chapter 2

State of the Art

2.1 Introduction

The main focus of this diploma thesis is to present a generic approach to out-of-core volume processing. A data processing approach is referred to as *out-of-core*, if the data cannot completely be held in the available physical memory. Most approaches assume that the data is completely memory-resident and directly accessible without any data initializing processes beforehand. There are several publications which present out-of-core approaches for specific volumetric data processing applications, for example, direct volume rendering or surface extraction algorithms.

2.2 General Volumetric Data Handling Approaches

2.2.1 Virtual Memory

A generic out-of-core approach, which is widely used by current operating systems is the concept of *virtual memory*. *virtual memory* is a common approach for providing a significantly larger memory space than physical memory is available. This is achieved by utilizing the paging mechanism of the CPU and the operating system.

The virtual memory is split into pages, memory areas of the common size of 4 kilobytes. In physical memory only the currently processed pages are kept the others are stored on the hard-disk. The paging mechanism of the CPU and additional processes in operating system load on store the pages on the hard-disk when required. This mechanism is completely invisible to the applications, so no special code for using virtual memory is required.
The virtual memory is more or less a generic out-of-core processing approach. It is sufficient for common applications on common computer systems, but its efficiency dramatically degrades when processing large amounts of data. This is due to the relatively small pages an high latency of the paging mechanism. Therefore, for large data processing more efficient approaches are required.

2.2.2 Application Controlled Paging Mechanism

There are only very few approaches for general volumetric data handling. For example, Cox et al. [5] describe such an approach: It is stated that a complete reliance on operating system virtual memory for out-of-core visualization leads to poor performance due to the complexity of the paging mechanism of the CPU and operating system as stated in the previous section. To overcome this issue they developed a paged segment system, where application control over several principles of memory management can significantly improve the performance. This is achieved by sparse traversal, i.e. loading only required data, usage of a blocked data storage layout and controlling the page size.

2.3 Specific Volumetric Data Handling Approaches

2.3.1 Iso-Surface Extraction Methods

Many approaches for iso-surface extraction algorithms have been developed. Some of them also support out-of-core. For surface extraction methods, auxiliary structures like tree-based search-structures are employed for sparse traversal, i.e. processing of only required data while excluding non-relevant data. For example, the interval tree is an efficient search structure to retrieve intervals (i.e. minimum and maximum data value of a 2x2x2 cube) containing a given query data value. Cignoni et al. [4] employ interval trees for fast location of relevant cells, i.e. a cube of 8 adjacent voxels intersected by an iso-surface. Further issues about storage requirements and other operations affecting the iso-surface extraction are addressed.

It has been shown that the search space can be considerably reduced through tree-based search-structures. They are very efficient for relatively small data-sets, which fit easily into the memory of common computer systems. However, due to their additional memory requirements, their application for large data-sets is difficult. Saupe et al. [14] states, that such search-structures can use up more memory than the original data itself. An approach trading off memory usage for extraction speed is presented: A hybrid algorithm combining binary space partition (BSP) trees with fast search methods at leaf nodes of the BSP tree and memory-free linear search at the remaining leaf nodes.

Chiang et al. [3] extend the interval tree for out-of-core iso-surface extraction: A novel application for the extraction of iso-surfaces from volumetric data employing an I/O-optimal interval tree is presented. A search structure is generated from the data-set in a preprocessing step and stored on disk beside the volumetric data. The extraction algorithm efficiently processes the data directly from disk and has very low memory requirements leaving most of the memory for storing the iso-surface.

2.3.2 Volume Rendering

Yang et al. [17] have developed a data-driven execution model for ray-casting that achieves a maximum overlap between rendering computation and disk I/O. Modern hard-disk controllers are able to autonomously transfer data to and from the memory without utilizing CPU resources. During the transfer, the CPU can be used for data processing. Further an application-specific file system maximizing the overlap between disk I/O and computation is introduced.

Bajaj et al. [1] propose a parallel/distributed ray-casting scheme for very large volume data. This method, based on data compression, attempts to enhance the rendering speedups by quickly reconstructing voxel data from local memory rather than expensively fetching them from remote memory spaces. This scheme minimizes the communication between the distributed processing elements during rendering computation.
Guthe et al. [7] present an interactive exploration method for animated volumetric data. The method employs video encoding methods, i.e. wavelet transformation, motion compensation, quantization and various encoding schemes for the resulting wavelet coefficients, which is optimized for good visual impression of the reconstructed volume. Further the temporal coherency is exploited. It was possible to achieve interactive frame rates for images with a resolution of 256x256 pixel.
In another work Guthe et al. [8] present an approach to interactive rendering of large volume data at interactive frame rates. The volumetric data, stored in wavelet representation, is decoded and rendered on the fly and rendered utilizing graphics hardware.
These four previously presented rendering approaches [17, 1, 7, 8] employ a blocked data structure for handling the volumetric data-sets. The volume data-sets are divided into cubes with a width of 2^n voxels. These cubes, called *cells* [1], *blocks* [8] or *macro-voxels* [17], are the *smallest data entities* which the presented compression and rendering algorithms are applied on. The methods presented in the papers are exclusively used in combination with rendering using ray-casting.
An approach exploiting the memory architecture of an Intel Pentium III system is presented by Knittel [9]. The data-set is stored and replicated in such a way, that during accesses to the data-set cache misses are reduced to a minimum. However, due to the specific storage method, the memory requirements are four times the size of the data-set. For large data-sets this method would be inadequate.

2.4 Data Reduction by Wavelet-Transformation and Compression

Rodler et al. [12] propose a wavelet based method for compressing volumetric data with little loss in quality allowing fast random access to individual voxels within the volume. It was possible to achieve very high compression rates with fairly fast random access.
Bajaj et al. [1] use wavelet transformation, for entropy encoding they present their own method. Guthe et al. [7, 8] use also wavelet transformation and various entropy coding methods: LZ77 [18], arithmetic coding and zero tree coding [15].
Many approaches [1, 12, 7, 8] employ a two stage compression method consisting of signal transformation and an entropy compression step. Additionally a quantization step is inserted between the two steps: For the transformation step, the wavelet transformation is widely used. For the entropy compression step, code-book based or tree based compression algorithms are used. Examples for code-book based algorithms are: LZW [16] and LZ77 [18]. Today LZ77 descendants are widely used in many compression tools (for example ZIP) and image formats (for example PNG). LZO [11], the compression algorithm used in this diploma thesis, is also based on LZ77. Tree-based compression algorithms are: zero trees [15] and its descendants like SPIHT [13].
The signal processing step combined with the quantization step is used for controlling the quality of the compression. It is possible to reduce the amount of required space for storing the data 1/100 with only a little loss of information.

Chapter 3

Computer System Issues

3.1 Introduction

In the past years the discrepancy between processor and memory performance has significantly increased, even more, the discrepancy between CPU performance and mass storage access speed has dramatically diverged. This issue turns out to be a potential bottleneck for applications which require very fast access to large amounts of data. Volumetric data processing is prone to cause problems, since several algorithms require fast processing of large amounts of data.

Besides the discrepancy between the CPU and mass storage devices, another important issue arises with the complex design of current computer hardware: The hardware is highly optimized for sequential contiguous accesses to large chunks of data, which does not fit the way of processing of volumetric data requiring arbitrary accesses to data.

For many algorithms in the field of computer science, the device whereon the algorithm is implemented is not important. In contrast, since volumetric data processing requires efficient data transfer capabilities of all hardware components, it is of particular importance to investigate the properties of the computer hardware. In this chapter, each hardware component in the data processing path affecting the efficiency of it in some way, is surveyed.

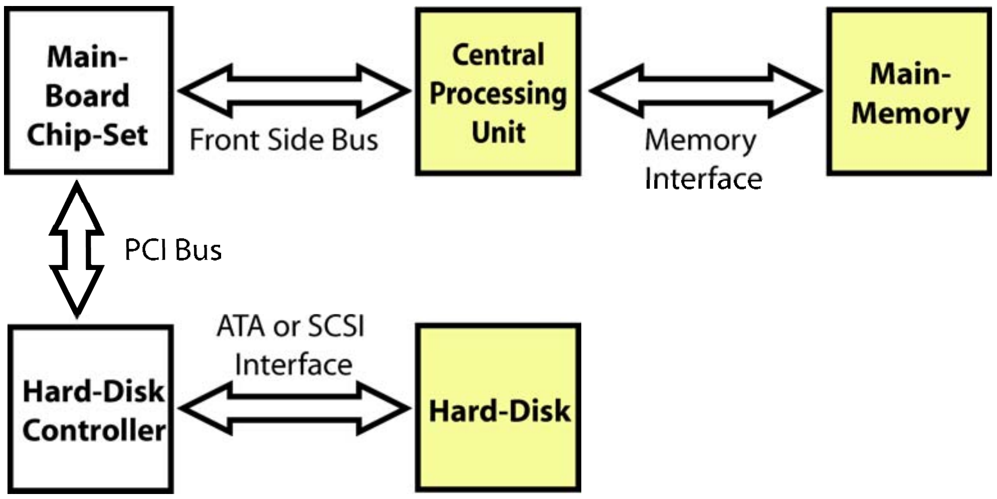


Figure 3.1: A computer system consists of three major components dealing with data: The *CPU* as data processing unit, the *Main-Memory* for volatile temporary data storage and the *Hard-Disk* for permanent data storage. Besides that, there is also some *interconnect logic* required, i.e. the *Main-Board Chip-Set* and the Hard-disk Controller, interconnecting these main components. The arrows denote a bidirectional communication interface between these components.

3.2 Data Processing Components

A computer system consists of three major components inherently dealing with data: The CPU as data processing unit, the main-memory for volatile temporary data storage and the hard-disk for permanent data storage. These three components are depicted in Figure 3.1

1. CPU: The CPU is the *Central Processing Unit* of a computer, consisting of the processing units, the level 1 and level 2 cache and external interfaces (see Figure 3.2). The processing units perform various operations on a set of registers, controlled by instructions. The level 1 cache, separated for data and instructions holds the data respective instructions which are actually processed. The level 2 cache holds copies of recently processed data of the main-memory.

2. Main-Memory: The main-memory is a volatile temporary data storage holding the data which is processed during the runtime of the computer system. The main-memory is accessed by the CPU through its caches.
3. Mass storage device: A common mass storage device is the so-called *hard-disk drive*. All storage devices have in common, that data can be permanently stored.

Beside these three main components, there is also some interconnecting logic, i.e. the main-board chip-set containing auxiliary system components like a real time clock and interfaces implementing several communication bus standards. Further additional components, like a hard-disk controller for controlling the communication between a device interconnect bus and the hard-disk drive, are required.

3.3 Memory Hierarchy

The memory of contemporary computers is structured in a hierarchy of successively larger, slower, and also cheaper memory levels (see Figure 3.3). The complexity of the hardware architecture incurs a penalty for programs which do not take optimal advantage of this hardware architecture. Disregarding the architectural peculiarities of the hardware results in an increased latency and reduced efficiency of data transfers. When developing efficient data processing algorithms it is very important to take a close look at the memory hierarchy. The hierarchy consists of successively larger but slower memory technology. Table 3.1 shows the memory specifications of a commodity computer.

3.3.1 Hierarchy Properties

The registers are the topmost level of the memory hierarchy which can hold data (see Figure 3.3). The data for the registers is transferred from the level 1 cache. The level 1 cache is used as temporary storage for instructions and data, making sure the processor has a steady supply of instructions and data

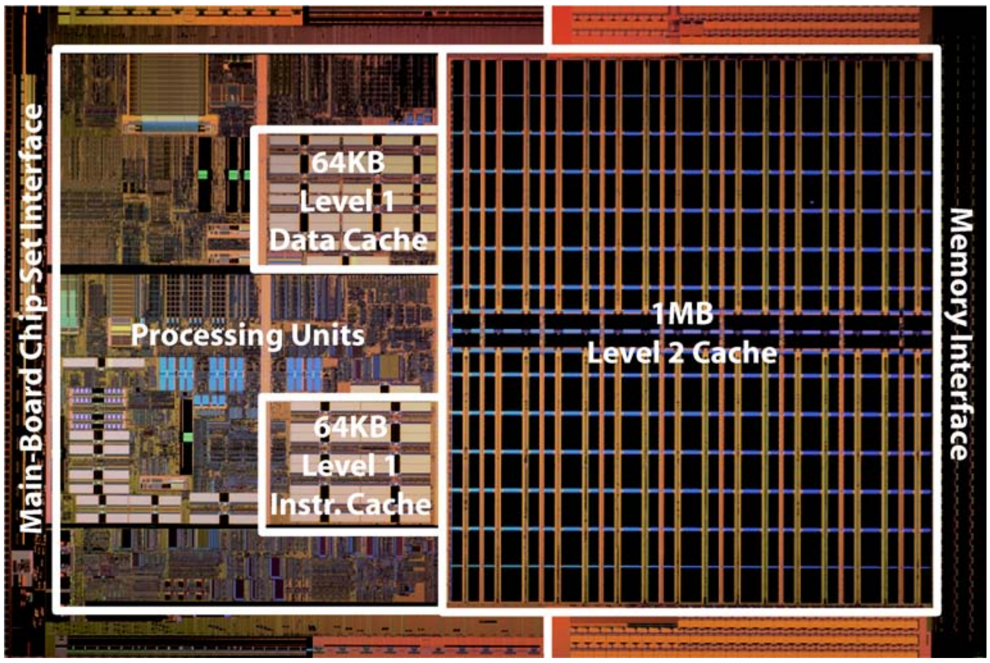


Figure 3.2: Structure of a modern CPU (AMD Opteron) : *Level 1 Cache (Data and Instructions)*, *Level 2 Cache*, *Processing Units* and external interfaces, i.e. the *Memory Interface* and the *Main-Board Chip-Set Interface*, are integrated on a single die. (The *CPU* image is taken from the *AMD Digital Media Library*)

Level	Latency	Size	Bandwidth
Register	1 - 3 ns	1KB	
Level 1 cache	2 - 8 ns	8 - 128KB	20GB/sec
Level 2 cache	5 - 12 ns	0.5 - 8MB	10GB/sec
Main-memory	10 - 60 ns	256 - 2GB	2GB/sec
Hard-disk cache	50 ns	2 - 8MB	200MB/sec
Hard-disk	8 - 20 ms	100 - 300GB	50MB/sec

Table 3.1: Memory hierarchy of modern computer systems. The memory hierarchy is structured top-down in successively larger but slower storage technology.

to process while new data is transferred from the main memory. As listed in Table 3.1 the common size of the level 1 cache is in the kilobyte range

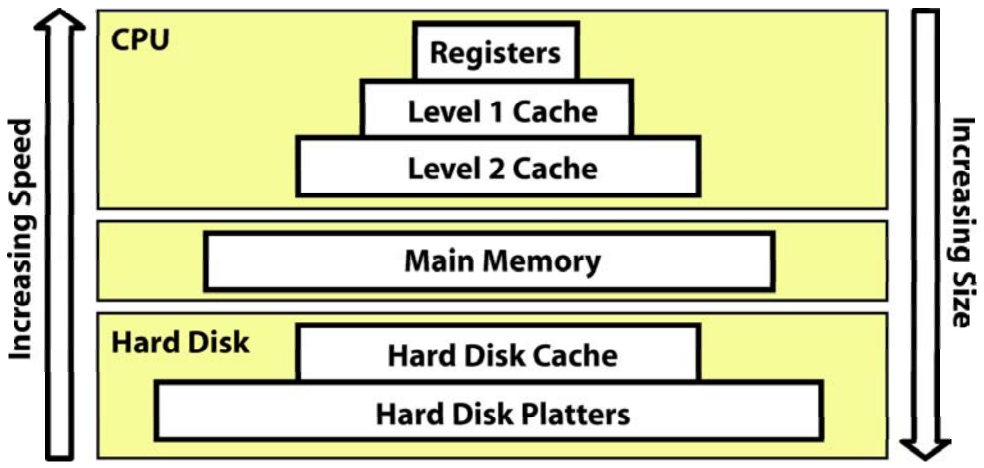


Figure 3.3: The memory hierarchy is mapped onto the three data processing components (yellow boxes). The *CPU* contains the upmost three levels: *Registers*, *Level 1 Cache* and *Level 2 Cache*. The next level below is the *Main-Memory*, placed physically close to the CPU. The *Hard-Disk* is at the bottom level. It communicates through interconnect logic with the CPU. It can also have a cache.

with an access-time of few nanoseconds. The level 2 cache is the high speed memory caching the data of the main-memory. The size of the level 2 cache is several megabytes with an access-time around 10ns. The next level is the main-memory holding the actual data being processed. Its size is roughly 1000 times the size of the level 2 cache and is usually in the gigabyte range. The next level below is the mass storage device, usually a hard-disk drive. It also has a cache with a size in the megabyte range for faster access. The size of the hard-disk can be several hundred gigabytes.

Current operating systems support a virtual address space with paging which allows to extend the memory space onto the hard-disk. The file on the hard-disk holding the pages is called *swap file*.

Going up the cache hierarchy towards the CPU, caches get smaller and faster. In general, if the CPU issues an operation on a data item, the request is propagated down the memory hierarchy until the requested data is found. It is very time consuming if the data is often found in lower levels. This is due to the propagation delay itself as well as to the back propagation of data through the complete hierarchy. For good performance, the number of accesses to the lower levels has to be reduced to an inevitable minimum. Therefore emphasis is put on efficient accessing strategies to the lower levels, like hard-disk and main-memory. Thus, the main focus lies in optimizing main-memory and hard-disk accesses in the following way:

- Loading data from lower levels only if required.
- Sequentially accessing data in memory and on hard-disk.
- Processing of data blocks smaller than the cache size.

3.3.2 CPU

The memory of the CPU consists of: registers, level 1 cache and level 2 cache. In recent years, the level 1 cache and the level 2 cache are integrated together with the CPU on a single die as depicted in Figure 3.2. The short connections between the processing units and the caches dramatically reduces the propagation delay (the speed of signal propagation in silicon is around 20 centimeter per nanosecond), which results in very fast access times in the nanosecond range (see table 3.1.) to the level 1 cache and level 2 cache.

Registers

The registers hold single data words for immediate processing and are addressed directly by the CPU. The registers being accessed are encoded in the instructions. Registers are the fastest cache.

Level 1 Cache and Level 2 Cache

A cache is a highly efficient memory storage device. It holds copies of data blocks stored on its associated storage device in a lower memory hierarchy level.
In general, caches are hidden structures and therefore not directly controllable by software. Caches serve as a temporary storage for fast data accesses.

Two issues are addressed with a cache at once: First, repeated direct accesses to its relatively slow associated storage device holding the actual data are avoided. Second, the efficiency of repeated access to the data is increased because of the short access latency of the cache. It is ensured that the data in the caches is always consistent with the data in the storage device. The only way to optimally utilize caches is accessing them in a pattern that the number of updates of the data in the cache from the actual storage device is reduced to a minimum. A situation, wherein a significant fraction of time is constantly used up by data replacements in the cache, is referred to as *cache thrashing*.

A cache is a relatively small memory holding temporary data. The memory is split up into small units, called *cache lines*, usually 32 or 64 bytes long. A cache line is synchronized only as a whole with the main-memory. Thus a cache consists usually of 2^m cache lines with a cache line length of 2^n. In *direct mapped* caches a location in main-memory of the size of the cache line is mapped exactly to a single cache line. The bits 0 to $n-1$ of the memory address are used as offset into the cache line, the bits n to $n+m-1$ are used to address the cache line itself. The remaining bits are stored as so-called *tag bits* in the cache. In a *fully associative cache* a memory location can be mapped to any cache line. This type of cache is seldom used because it is very complex to implement in hardware. A good compromise is a *n-way set associative cache*. The cache is organized in s sets of n cache lines. A memory location is mapped to one set. A direct mapped cache is an one-way set associative cache.

Since a cache line is updated only as a whole, even when one single byte is accessed, the whole cache line with a length of 32 or 64 bytes is updated. This is the reason for the degrading efficiency during cache thrashing.

The simple test code in Figure 3.4 shows, that the performance dramatically decreases if the memory is not sequentially accessed. Both triple nested loops do exactly the same: incrementing each field of the array by one. Theoretically the speed must be the same, but practically it is not. Experiments on a AMD 1.2 GHz CPU with 133 megahertz SDRAM with 64bit interface width (resulting in a theoretical bandwidth of 1 gigabyte per second) have shown, that the *sequential access* is roughly 50 times faster than the *non-sequential access*.

```
int x,y,z;

// array size 16MB
char array[256][256][256];

// traversal in x,y,z order = total sequential access
for (z=0; z<256; z++)
  for (y=0; y<256; y++)
    for (x=0; x<256; x++)
      array[z][y][x]++;

// traversal in z,y,x order = non-sequential access
for (x=0; x<256; x++)
  for (y=0; y<256; y++)
    for (z=0; z<256; z++)
      array[z][y][x]++;
```

Figure 3.4: This simple code shows the runtime behavior of a cache by accessing a three dimensional array in two different ways.

Figure 3.5 shows the results of the following test: A memory block of a certain size (*block size in kilobytes*) is allocated. Then single bytes are subsequently read from the memory block. After each byte-read an offset is added (*distance*) to the address. This scheme is depicted in Figure 3.6 It can be concluded, that as long the processed data fits into the L1 cache, the performance is not degraded regardless of the offset. The results for 64, 128 and 256 bytes distance are identical because the transfer capacity of the main-memory is exceeded: At each access a cache line of 64 bytes is updated. Since for each byte access a cache line of 64 bytes is updated, theoretically a maximum speed of 17 megabytes per second is possible, practically it was around 12 megabytes per second.

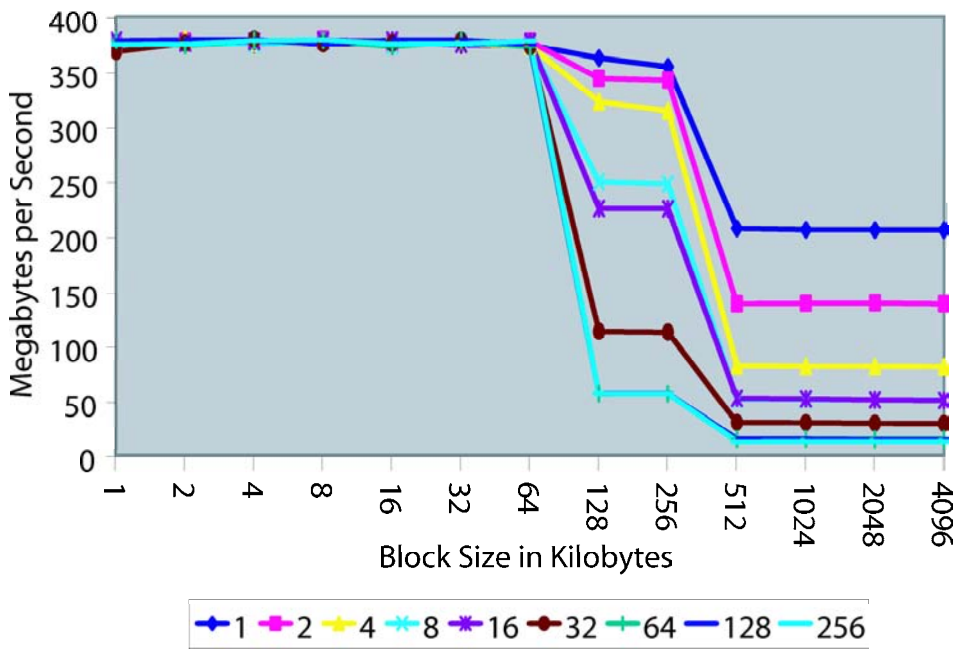


Figure 3.5: Sequential non-contiguous read access of single bytes with varying distance. The distances (offset added to the address after each byte read) are shown in the box at the bottom. The distance of 1 corresponds to a sequential contiguous access. This Figure clearly shows that the efficiency of accesses within the L1 cache, i.e. block size 64KB or smaller, is not degraded in any case. Results of an AMD CPU with 64KB level 1 cache and 256KB level 2 cache. The cache line length is 64 bytes. A graphical scheme of this test is depicted in Figure 3.6

3.3.3 Main-Memory

The history of memory is closely related to the history of the computer. From the beginning on a device storing bits over a longer period of time was required. Early random access memory (RAM) consisted of electron tubes or a wire grid with ferrite rings at the cross-overs. The development of modern memory started with the introduction of integrated circuits. The most common used memory type is the *Dynamic Random Access Memory* (DRAM). A simplified model of the DRAM architecture is depicted in Figure 3.7.

DRAM is organized in the following way: A bit is stored in a memory cell.

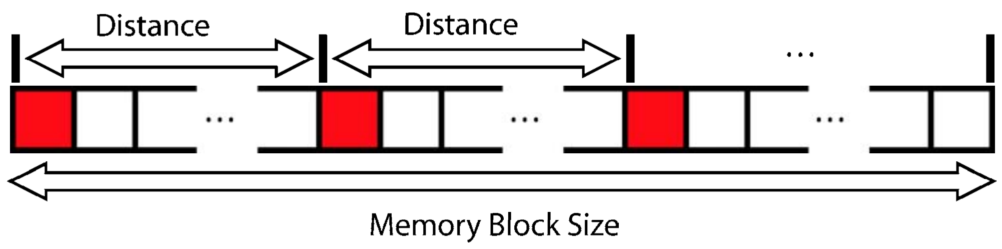


Figure 3.6: Read test scheme. A memory block of a specific size is allocated and single bytes are read. After each read a specific distance is added to the address.

A memory cell consists of a small capacitor connected to a transistor. The memory cells are organized on a grid, also called storage array. The array is structured in rows and columns. The address supplied to the array by the memory controller is split up into two parts. the one part is used for row addressing and the other part for column addressing. A data word has a length of one bit. First the *Row Address* is send to the *Row Decoder*. It selects the desired row. Then each bit in the row is send to the *Column Decoder*. After that, the *Column Address* is supplied to the *Column Decoder* to select the desired bit, which can be read from the *Data I/O* interface (see Figure 3.8, *Random Access*).

For word lengths greater than one, multiple memory arrays are operating simultaneously.

Beside the structures storing the actual bits, additionally a refresh circuit, control and addressing unit is integrated on a single chip.

Since the charges volatilize over time, the charges have to be refreshed every few milliseconds. This is autonomously done by the memory controller. Also after reading, which destroys the contents of the memory cell, the affected charges are restored.

In the following paragraphs, several DRAM types are described in a chronological order, which is also an order with increasing performance.

Ordinary DRAM is the simplest and also one of the oldest types of DRAM. Sequential accesses are supported by keeping the currently addressed row,

while supplying only different column addresses to the column decoder, which avoids extra row addressing (see Figure 3.8, *Row Address Re-Use*). This can be used to access a complete row by increasing the column address after each access.
Increasing the column address is automatically performed by the FPM-DRAM (Fast Page Mode DRAM). The DRAM automatically increments the column address, allowing the controller to access the next location without having to supply a new address. This allows a very efficient sequential access of a complete row, also called *page*. In consumer hardware the typical clock rates are 16 to 50 megahertz, with a word length of 16 or 32 bits yielding into a theoretical transfer bandwidth of 32 to 200 megabytes per second. The access latency time is around 60 to 100 nanoseconds.
EDO-DRAM (Extended Data Out DRAM) is similar to FPM-DRAM with the additional feature that a new access cycle can be started while keeping the data output of the previous cycle active. This allows a certain amount of overlap in operation, which improves the speed by roughly 5% (see Figure 3.8, *Overlapping Addressing*). Typical ratings are 50 to 66 megahertz with a word length of 32 bits yielding in a theoretical bandwidth of up-to 266 megabytes per second. The access latency time is around 40 to 60 nanoseconds.
Synchronous DRAM (SDRAM) is an improved type of DRAM. While DRAM reacts immediately to changes in its control inputs, SDRAM has a synchronous interface, meaning that its signal I/O is synchronized with a clock signal. This allows the SDRAM to have a more complex pattern of operation than plain DRAM. Accesses to the SDRAM are controlled by sequences of commands allowing pipelining, i.e. overlapping of read or write sequences. Typical ratings are 66 to 133 megahertz with a word length of 64 bits yielding in a theoretical bandwidth of up-to 1 gigabyte per second. The access latency time is 10 nanoseconds.
Doubled Data Rate SDRAM (DDR-SDRAM) is a later development of SDRAM. Plain SDRAM acts on rising edge of the clock signal. DDR-SDRAM acts on the rising and the falling edge thereby halving the required clock rate for a given data transfer. Common configurations support a 128bit data word width with 266 to 400 million words per second, resulting in a theoretical

bandwidth up to 6.4 gigabytes per second.

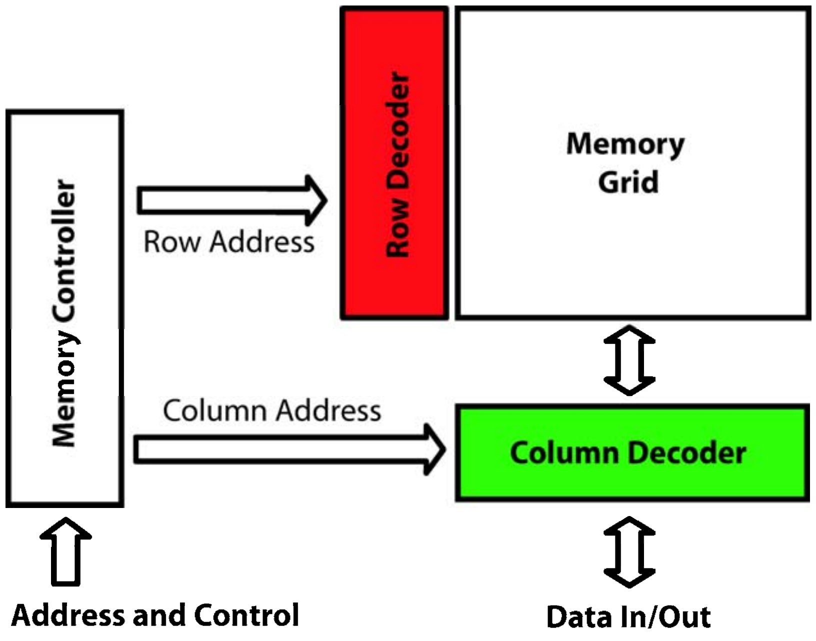


Figure 3.7: Basic memory architecture: The input data is the address and control (read or write access) and the output (read access) respective input (write access) is the data for the specified address. The bits are stored in memory cells organized in a *Memory Grid*. The *Address* is split into a *Row Address* and a *Column Address*. First the *Row Address* is sent to the *Memory Grid*, then the whole row, also called page, is sent to the *Column Decoder* which selects the desired column.

3.3.4 Mass Storage Device: Hard-Disk Drive

The most commonly used mass storage device is the so-called *hard-disk drive*. It consists of a rotating stack of platters made of aluminium or ceramics coated with a magnetic material. A moveable lever called *actuator arm* holds the *heads* which perform write and read actions on the platters (see Figure 3.9). The rotation speed was initially 3600 rpm in the early days, later 4500 rpm and 5400 rpm, and nowadays consumer hard-disk drives have a rotation speed of 7200 rpm. High-end hard-disk drives have a speed from

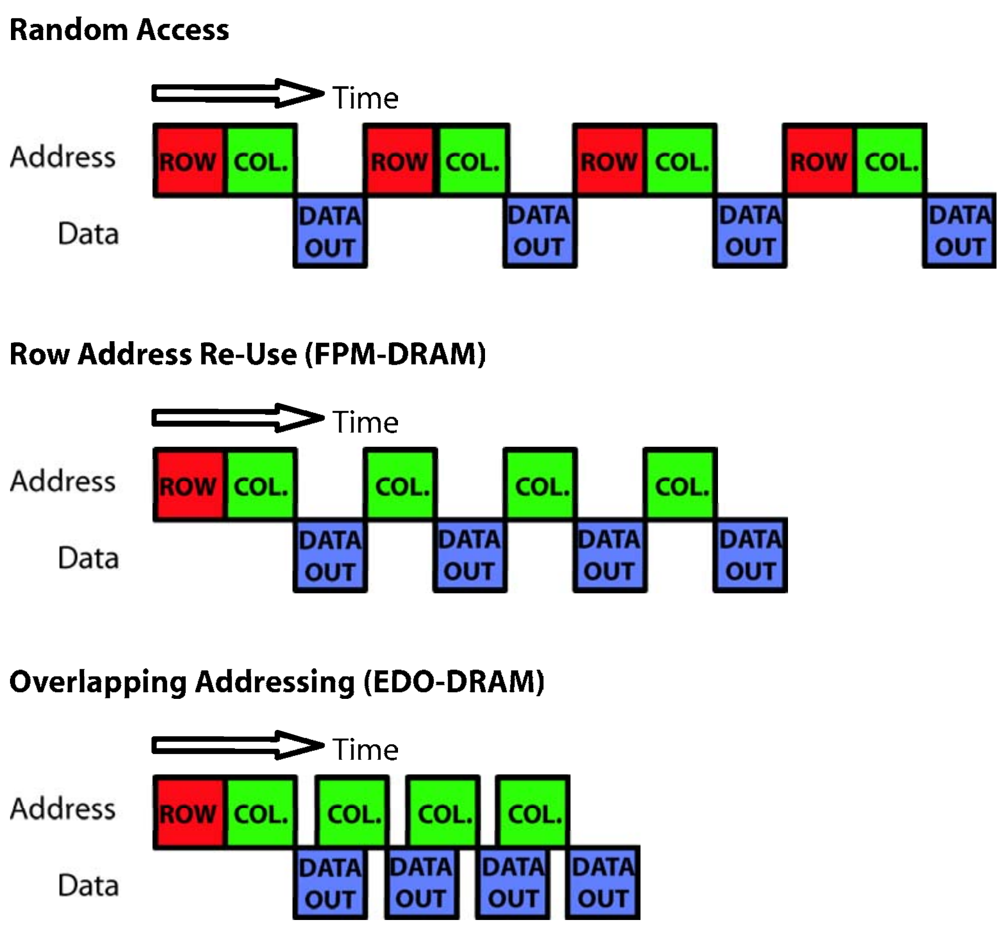

Random Access
Time
Address
Data
ROW
COL.
DATA OUT
Row Address Re-Use (FPM-DRAM)
Overlapping Addressing (EDO-DRAM)

Figure 3.8: Memory read sequences: For reading data words randomly, first the row and subsequently only column addresses have to be send for each word (first sequence). Sequential reads: With FPM-DRAM the sequential reads have been optimized by reusing the column address, for example automatic column address increment (second sequence). Later, overlapping the requests (third sequence) was introduced with EDO-DRAM.

10000 rpm up-to 15000 rpm. At the time of writing this diploma thesis the common size is around 200 gigabytes. The data density is around 600000 bits per inch on a track, and 100000 tracks per inch on a platter. this results roughly in a data density of 60 gigabits per square inch: In a 200 gigabyte disk drive, the stack holds two platters (in total 4 sides), each of them holding 50 gigabytes. Reading and writing is performed by accessing the bits on a track surpassing the head. The theoretical maximum transfer speed is calculated as follows: A track has an average length of 16 cm (depending on its radius the length ranges from 7 to 25 cm). Therefore it holds roughly 4 megabits of data. The maximum transfer speed at 7200 rpm (120 rounds per second) is about 480 megabits per second or 60 megabytes per second. The track to track seek time is around 1ms, a random track seek is 9ms average. These properties of a hard-disk lead to the conclusion, that only sequential contiguous accesses can utilize the hard-disk architecture optimally. Therefore, for out-of-core approaches it is desirable to sequentially access the hard-disk by the way of contiguous data blocks.

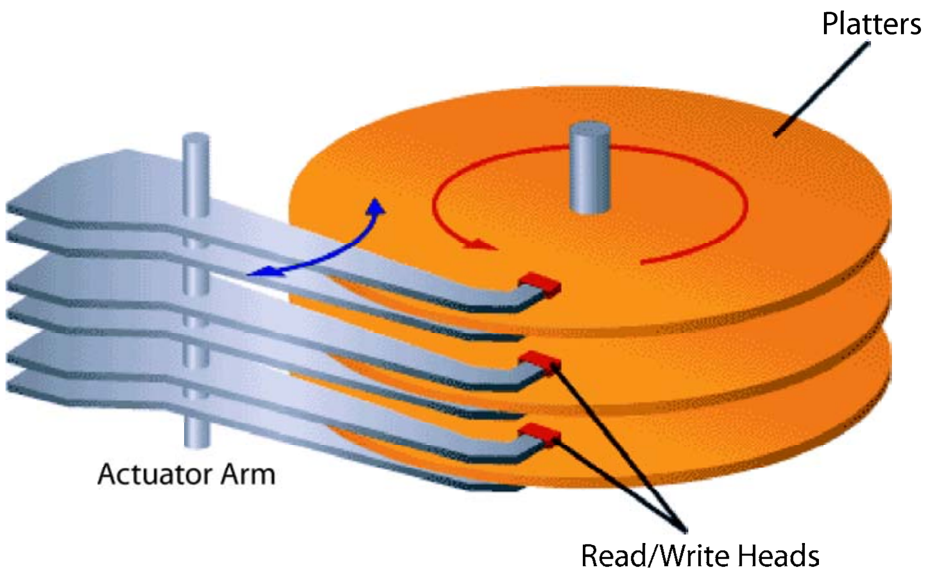


Figure 3.9: Hard-disk drive scheme: A moveable lever called *Actuator Arm* holds the *Read/Write Heads* which perform write and read actions on the *Platters*. The *Platters* are made of aluminium or ceramics coated with a magnetic material. Several *Platters* are mounted on a rotating spindle.

3.4 Summary

Many common memory architectures have one feature in common: they are optimized for sequential contiguous accesses. Each memory level, starting at the top with the level 1 cache down to the hard-disk is designed so that sequential accesses are the most optimal way to access the data. A sequential access is simply an access to a contiguous block of data with specific starting address and length. A reading access results in a data transfer from one memory hierarchy level to the levels above. The propagation of the data bottom up through the hierarchy is crucial. In volumetric data processing fast read accesses are very important to be efficient. Most memory hardware support sequential contiguous accesses by so-called *burst transfers*. After the starting address, and optionally if supported by the architecture the length of the desired block, has been sent to the memory, a whole sequence of words is transferred at once instead of a single word. A word is the smallest transferable element in a memory. Thus it is important to exploit this behavior when accessing the main-memory or any mass storage device. This issue is discussed later in detail in the section about the *optimal brick size*.

Irregular accesses cause a significant degradation of performance if memory hierarchy levels are affected which are sensitive to non sequential accesses. The affected levels are usually the main-memory and the mass storage devices. As long as no update of the caches through those levels is constantly required, no significant performance decrease occurs.

Chapter 4

Block Volume Memory Layout

This chapter addresses the issue, that if an algorithm accesses a memory or mass storage device in an irregular pattern or, more simply speaking, not in a sequential contiguous fashion, the performance drastically degrades. Methods overcoming this issue by exploiting the advantages of the current computer architecture hierarchy are presented. The intention of these methods is to allow irregular accesses while maintaining sequential accesses for performance reasons.
First, volumetric data in general and its properties are reviewed. Second, the sequential volume, a commonly used layout for volumetric data, is introduced and discussed. Further, the block volume concept remedying several deficiencies of the sequential volume and taking concepts from the previous section into account is presented. Finally, the volume processing hierarchy, a high-level concept for accessing the block volume is introduced.

4.1 Volumetric Data Properties

In this section, several characteristics of volumetric data, which can be exploited by various algorithms, are discussed. The focus is put on computer tomography data, but it does not preclude that these characteristics apply also to other types of volumetric data, such as magnetic resonance imaging data, simulation data, etc.

Many algorithms access only a fraction of the volumetric data-set. This fraction can be bounded in the spacial domain or in value domain. For example such a fraction in spacial domain is a block shaped sub-set of the whole volume data-set. In value domain such a fraction consists of all voxels in a certain value range.

4.1.1 Non-Relevant Data

Non-relevant data is, as its term denotes, not relevant for the result of an algorithm. It often degrades the performance of an algorithm due to unnecessary processing. Additional data structures to reduce accesses to non-relevant data can considerably increase the performance of an algorithm.
An example of an algorithm requiring just a fraction of data in value domain is the marching cubes [10] surface extraction method. It only requires the voxels close to the surface, represented by a fixed predefined value, to construct the polygonal mesh of the surface. Figure 4.1 shows the classification of blocks into relevant and non-relevant blocks for surface extraction. If an implementation of this algorithm uses a sequential layout in a slice-by-slice manner, many voxels would be read from the mass-storage device, which do not contribute to the surface construction.
Typical data-structures to determine relevant voxels are tree-based search structures forming a hierarchical search space with increasing granularity. Large partitions of the search space can be omitted in the higher levels which significantly reduces the size of the search space. With these structures it is possible to efficiently exclude non-relevant data. Time-expensive memory or mass-storage accesses are avoided and the efficiency of the algorithms significantly improves.
As shown in Figure 4.2 volume data-sets acquired from computer tomography imaging devices contain large regions of voxels with similar values, representing homogeneous material in the scanned objects. For example, soft tissue or air. In many cases these regions are likely to be non-relevant data. Therefore search structures mentioned above can be used to identify these regions. For example, a simple octree structure can be used: Each

node contains the minimum and maximum density value of its sub-nodes. Assume an algorithm processing density values in a certain predefined range. If the algorithm encounters a node whose minimum-maximum range does not overlap with the range of the algorithm then the whole sub-tree can be omitted.

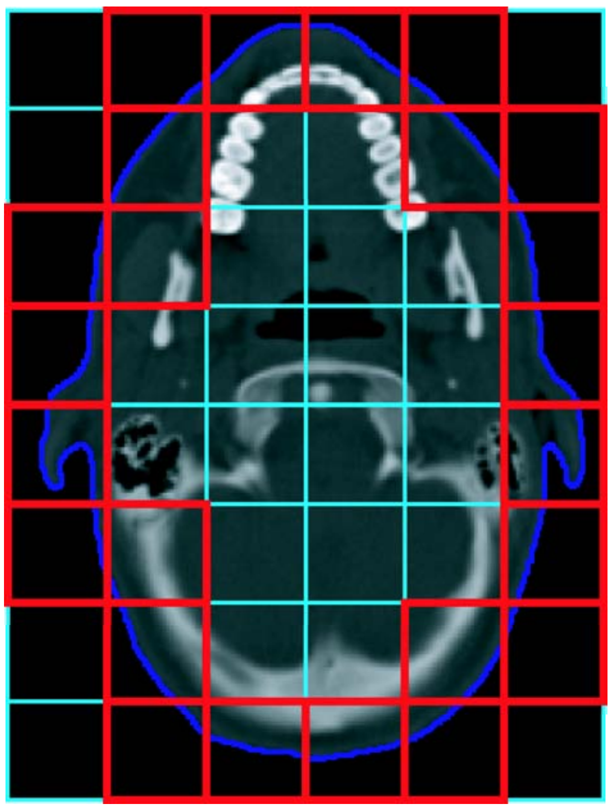


Figure 4.1: Classification of blocks into relevant (red) and non-relevant (cyan) blocks for surface extraction. The intended surface is highlighted (blue).

4.1.2 Memory Access

All memory architectures have one feature in common: they are optimized for sequential contiguous access. Each memory level, starting at the top with the level 1 cache down to the hard-disk is designed so that sequential accesses

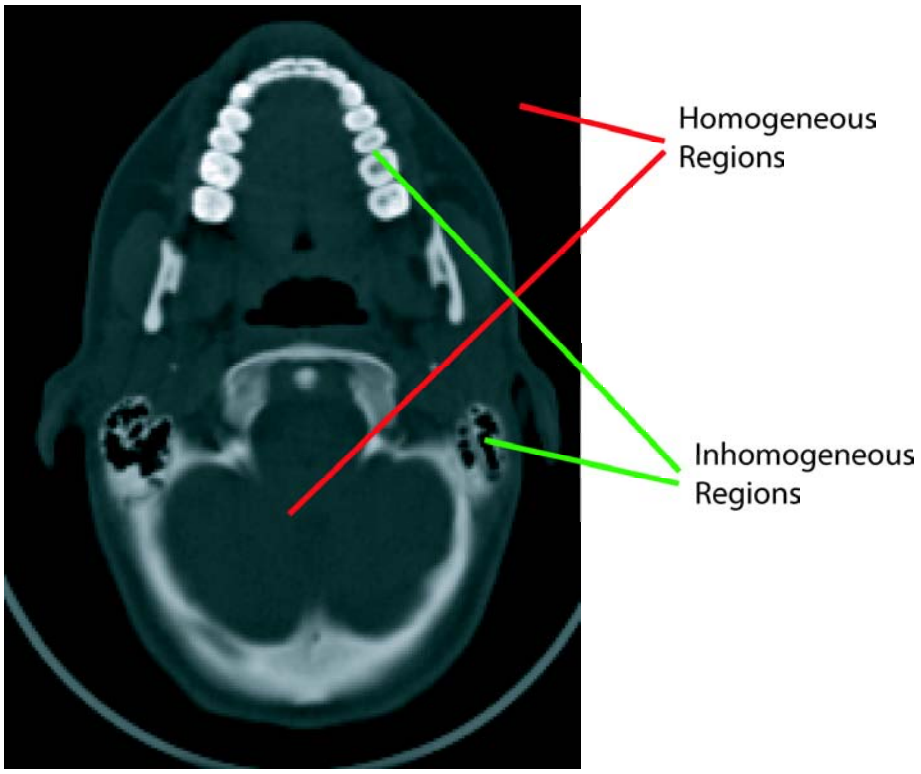


Figure 4.2: Types of regions in a data-set. *Homogeneous Regions* contain less information and *Inhomogeneous Regions* contain more information.

are the most optimal pattern to access the data. A sequential access is simply an access to a contiguous block of data with specific starting address and length. A read-access results in a data transfer from one memory hierarchy level to the levels above. The propagation of the data bottom up through the hierarchy is crucial. In volumetric data processing fast read access is very important in order to achieve high performance.

Contiguous sequential accesses are supported by most of the memory hardware by so-called *burst transfers*. After the starting address, and optionally, if supported by the architecture, the length of the desired block in words, has been sent to the memory controller, a whole sequence of words is transferred at once instead of a single word per addressing action.

Irregular accesses constantly trigger cache updates. This leads to a degradation of performance if memory hierarchy levels are affected which are sensitive to non-sequential accesses. The affected levels are usually the main-memory and the mass storage devices. As long no update of the caches through those levels is constantly required, no significant performance decrease occurs. Thus it is important to exploit this sequential access behavior when accessing the main-memory or any mass storage device.

4.1.3 Out-of-Core Processing

Assume a 4 gigabyte data-set to be processed and the following system available: 1 gigabyte of physical memory available for storing volumetric data, and a hard-disk holding the data with an interface speed of 100 megabytes per second. Other limiting factors (i.e. memory speed, CPU speed) are neglected. In this case it is impossible to keep the complete data-set in memory, therefore the complete data-set has to be loaded piecewise from the hard-disk during processing. In the theoretically best case the data-set can be processed in 40 seconds (neglecting all latencies and limiting factors, except the hard-disk interface speed). The processing time and memory requirements can be reduced.

Processing of relevant data. Most algorithms require only a fraction of the volumetric data-set. It is favorable to retrieve only the required data from the storage device which is relevant for the result of the algorithm. Auxiliary search structures are used to efficiently locate the needed data in spacial and value domain. This approach is similar to the sparse traversal in [7].

Compressed data. Effectively more information can be transferred over an interface in compressed form in the same time compared to uncompressed data. Ideally the sum of the transfer time of the compressed data and its decompression time should be smaller than the transfer time of uncompressed data to achieve a speed increase. Preliminary tests of compressing computer tomography data with the LZO-compression [11] showed that the data can be reduced by 30 to 50 percent and decompressed with a speed roughly twice the hard-disk interface speed.

Efficient resource usage. If a data-set does not fit into the main-memory as a whole, the memory must be reused for other portions of the data-set. An efficient way to achieve that is to partition the data-set into small entities which can be retrieved from the storage device as contiguous chunks of data. Ideally such a partition should consist of spatially bounded data. Such partitions can be slices (i.e. data in a plane) or small sub-volumes (i.e. data in cubically regions). In the following such a partition will be referred to as *smallest available entity.* This approach is similar to the *segments*, *pages* and *replacement policy* in [5]

4.1.4 Application Examples

The sequential volume allows an effective sequential line-by-line respective slice-by-slice processing of the data. Therefore this layout is very efficient for algorithms which process volumetric data in this manner.
An example for an algorithm processing volumetric data slice by slice is the well-known surface extraction method called *Marching Cubes* [10]. Algorithms processing single voxels like thresholding or windowing-functions also benefit from the sequential layout, because thy process the data sequentially. The most profoundly disadvantage of this layout is, that the effectiveness degrades dramatically when an algorithm requires a random access pattern to the data or, more simply, requires access in a different order than in a slice-by-slice or line-by-line manner. For example, accessing slices in the xz-plane requires processing of single lines across the entire data-set. Even worse, accesses to slices in the yz-plane require processing of single voxels, which totally compromise the caching mechanisms of the hardware, resulting in massive *cache thrashing.*
Many algorithms require accesses not only to single voxels but additionally to their neighborhood, i.e. a three-dimensional region around a voxel. These algorithms are also affected by cache thrashing. For example, such algorithms are gradient-calculations and general three dimensional signal processing operations, like filters and transformations.

4.2 Block Volume

4.2.1 Sequential Volume

The sequential volume has arisen in the 1970s with the upcoming of three dimensional medical imaging systems. The data acquired by such an imaging device is stored as a sequence of two dimensional images. These images itself consist of a sequence of lines which are itself sequences of single data values. The sequence of images can be interpreted as a three dimensional volume block where the values are stored sequentially starting with the first line of the first slice going to the last line of the last slice.
The sequential volume is structured as follows: The smallest data element in a sequential volume is a *voxel.* A sequence of voxels forms a *line.* A sequence of lines constitute a *slice.* A stack of slices constitute a *volume.* These structures and its relations are shown in Figure 4.3:

Grid. A volume grid is a discrete three dimensional structure of a predefined size in a three dimensional regular orthogonal coordinate system. For each voxel it establishes a relation to a position in space represented by the a grid point.

Voxel. A voxel consists of a vector, a sequence of values assigned to a position in a volume grid. For example, in CT data-sets this vector contains a single unsigned 12 bit value.

Line. A line is a sequence of voxels along an axis of the volume grid. If not identified otherwise a line is parallel to x-axis of a slice parallel to the xy-plane.

Slice. A slice consists of a number of lines. If not identified otherwise a slice is parallel to the xy-plane.

Block. A volume block consists of a stack of slices along the z-axis of the volume grid.

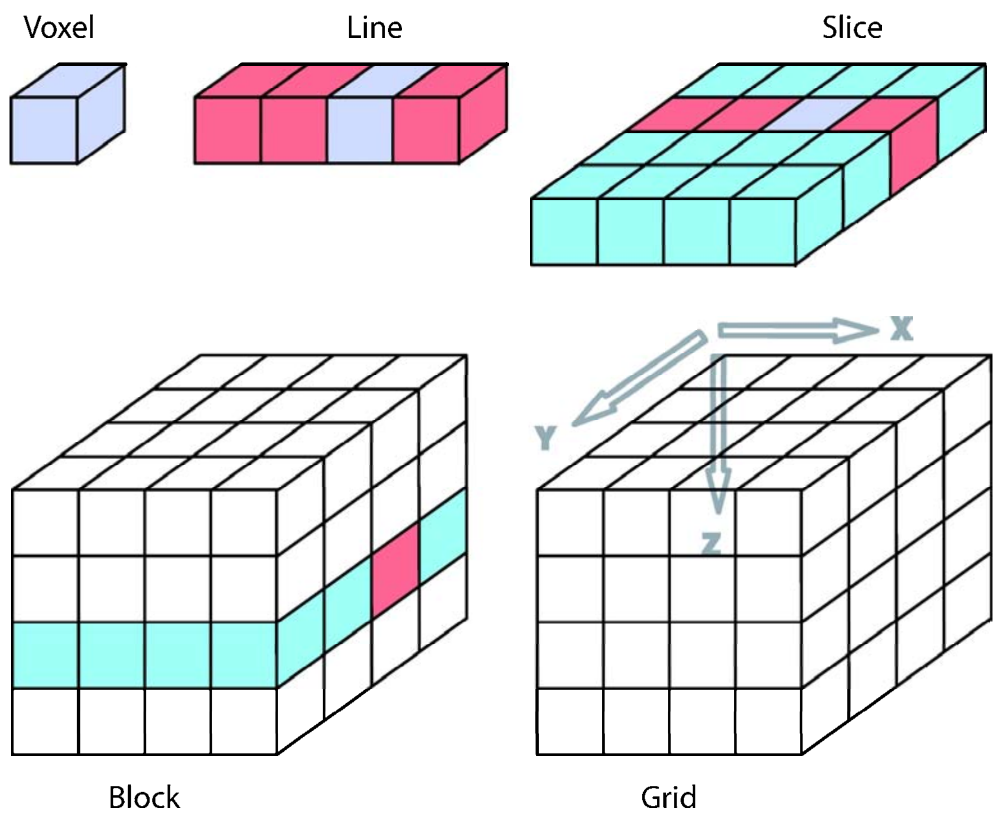


Figure 4.3: The sequential volume layout. A *Block* is a set of *Voxels* stored according to a Grid. A *Block* consists of a stack of *Slices*. A *Slice* consists of a *Line* of *Voxels*.

4.2.2 Motivation of the Block Volume

In Figure 4.4 images of a large volumetric data-set (1.6 gigabyte *Visible Male* data-set) are shown. On common computers, data-sets of this size can be easily stored on the hard-disk, but they often exceed their physical memory resources. Even larger data-sets can exceed the capabilities of all components of the computer system. For example, the address space of a CPU can be easily exceeded: If a CPU has a 32 bit address space it is not possible to directly address more than 4 gigabytes of memory.
Simply dividing the data-set into smaller manageable independent sub-sets is the most straight-forward way to circumvent this issue. Only one sub-set

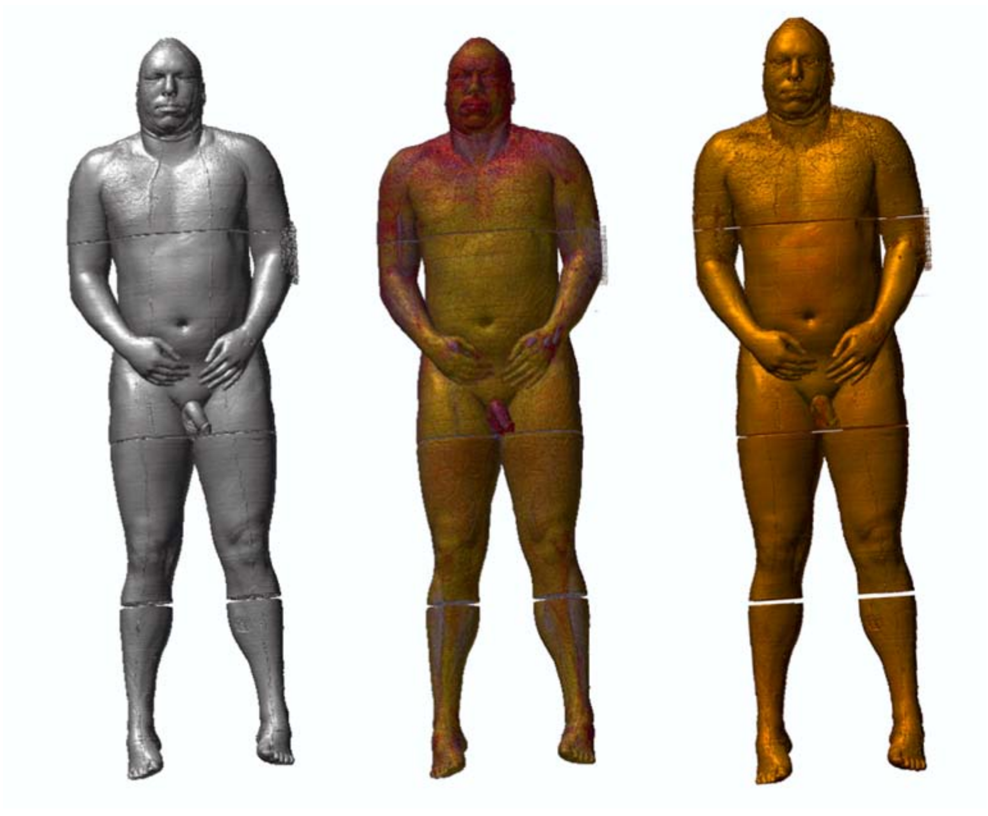


Figure 4.4: Three images of the *Visible Male.* A 1.6 gigabytes volumetric data-set.

is held in memory while the others are remaining on the hard-disk. This would be a reasonable solution if the algorithm can be applied on manageable sub-sets such that the results of each processed sub-set can be merged together easily to get the same result as processing the data-set as a whole. Especially for algorithms relying on common global properties, for example a minimum cost for the region growing algorithm, this is not easily achievable. Another way would be the compression of the data-set to reduce its memory requirements. This requires methods to directly operate on compressed data. However, the additional processing overhead, especially if random access is required, can be considerably high.

A good method would be a trade-off between speed and memory require-

ments. The data-set is divided into sub-sets that can be easily accessed through an interface that hides the actual data management from the algorithm by providing a clear generic interface for data access. The interface should allow random access to the data in a simple and efficient way. Additionally it should support fast compression to reduce memory requirements and better utilization of the hard-disk interface. The subdivision combined with compression allows storing the data-set in compressed form on a hard-disk and in uncompressed form in memory for processing.
Based on this concept an approach for managing large volumetric data-sets is introduced.
Based on these concepts, a data structure for storing volumetric data, the *block volume* is introduced.

4.2.3 Contiguous Block Volume

A block volume is a two level application of the sequential layout. The volumetric data-set is sub-divided in sequentially organized fixed-size data blocks. Also the data blocks are internally organized according to the sequential layout (see Figure 4.5).
A *contiguous block volume* provides a different way to store volumetric data. It has a higher processing overhead than a sequential volume, but its efficient cache usage easily compensates this overhead. Actually according to Grimm et al. [6] the data can be processed multiple times faster than in a sequential volume. The memory requirements are similar to the sequential volume.
The *contiguous block volume* demands a more complex addressing scheme compared to a simple sequential volume. The addressing scheme is described in one of the following sections. The intended purpose of the *contiguous block volume* is to provide an efficient data access. This is achieved by a data block size allowing efficient cache usage and an efficient addressing scheme. The concept of the *contiguous block volume* and its addressing scheme has been introduced by Grimm et al. [6].

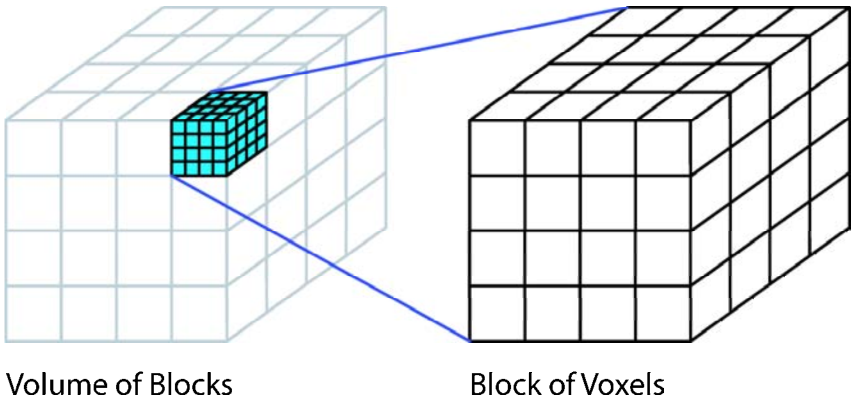


Figure 4.5: The volume data-set is organized in a two level sequential layout: The volume consists of sequentially ordered blocks, which themselves consist of sequentially ordered voxels.

Optimal Data Block Size

The data block size should be selected in favor of fast address calculation for data access and optimal cache usage. If the width of the data block is a power of 2, the address calculation can be efficiently done with bit-manipulating and shifting operations only.

The data block size is determined in such a way, that at least one, or even better, several data blocks should fit into the level 2 cache. Since solely 16 bit data is processed, the formula for the size calculation is: $DataBlockSize := 2 * (2^n)^3 bytes$. Thus meaningful sizes are: 1 kilobyte ($n = 3$), 8 kilobytes ($n = 4$), 64 kilobytes ($n = 5$), 512 kilobytes ($n = 6$) and 4096 kilobytes ($n = 7$). It was found out by Grimm et al. [6], that on a CPU with a level 2 cache size in the 0.5 megabyte to 1 megabyte range a data block size of 64 kilobytes is the most efficient size. Therefore a data block of $32^3 (n = 5)$ voxels is used in the implementation. Each voxel consists of a 16 bit unsigned value. Thus each data block has the size of 64 kilobytes.

Addressing Scheme

For accessing a voxel in a volumetric data-set, its location in memory, the address, has to be calculated. The address is derived from the position of the voxel in the three dimensional space. In the sequential layout the voxel address is simply a linear combination of the components of the position of the voxel. For the block volume a two step addressing scheme is introduced, which is described in the next section.
Since address calculations are performed very frequently, therefore the most important issue is simply *speed*. All steps in the address calculation follow this requirement. The calculations are done in a way that time-consuming CPU instructions like branching instructions (jumps, loops) and instructions associated with time consuming pipeline flushes are avoided. Also complex mathematical calculations (divisions, multiplications) are avoided. The calculations are designed to exploit the fastest instructions like simple arithmetic (addition, subtraction), bit-manipulation and shifting operations. Also no look-up tables are used to minimize the number of relatively slow memory accesses.

Absolute Addressing The absolute address is the address directly derived from the voxel position in the three dimensional space. For the block volume a two step addressing scheme is introduced: In the first step (see Figure 4.6) the block index and the voxel offset are calculated from the position. In the second step (see Figure 4.7) the voxel address is calculated. Since in the *contiguous block volume* the data blocks are sequentially stored in a large contiguous memory area (see Figure 4.8), the data block address can be directly derived from the block index by a simple shift operation.

Relative Addressing Once if the absolute address for accessing a specific voxel, the *active voxel*, is calculated, it is advantageous to have an efficient method to access the neighboring voxels. In this section an approach to efficiently access the 26-neighborhood around a voxel is presented. The neighborhood is the 3x3x3 *cube* of voxels with the *active voxel* in the center. The 3x3x3 cube, where the *active voxel* lies in, is called the *active cube*.

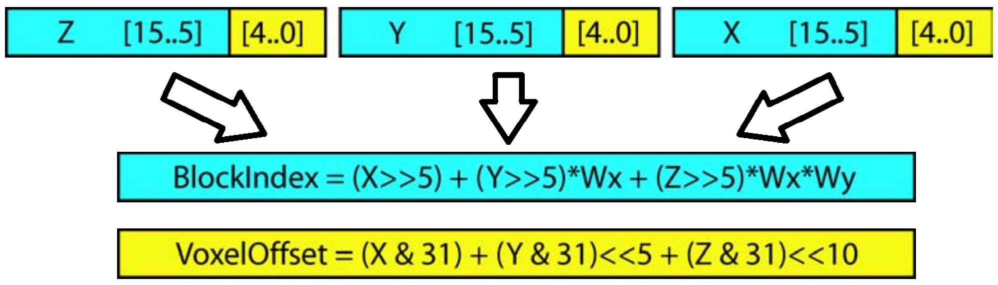

Z [15..5] [4..0]
Y [15..5] [4..0]
X [15..5] [4..0]
BlockIndex = (X>>5) + (Y>>5)*Wx + (Z>>5)*Wx*Wy
VoxelOffset = (X & 31) + (Y & 31)<<5 + (Z & 31)<<10

Figure 4.6: block index and voxel offset calculation. For the voxel offset, only the 5 lowest bits are used, for the block index the bits above the 5 lowest bits are used.

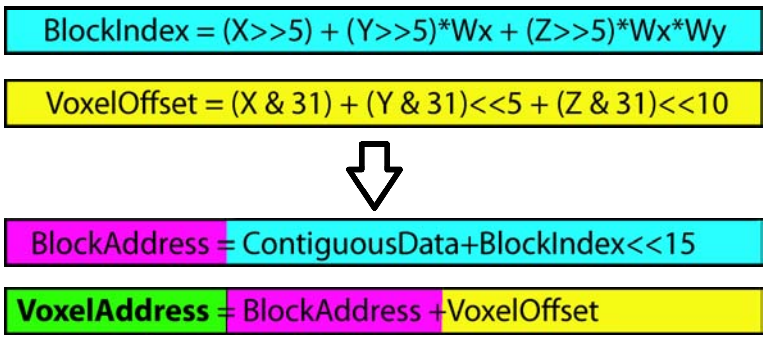

BlockIndex = (X>>5) + (Y>>5)*Wx + (Z>>5)*Wx*Wy
VoxelOffset = (X & 31) + (Y & 31)<<5 + (Z & 31)<<10
BlockAddress = ContiguousData+BlockIndex<<15
VoxelAddress = BlockAddress +VoxelOffset

Figure 4.7: Absolute voxel address calculation for contiguous memory.

For each voxel in a block there are 26 neighbors. The absolute addresses of the neighbors are calculated by simply adding an offset to the address of the active voxel. This offset is called the *relative address*. The relative address is the difference between the the absolute address of the active voxel and one of its 26 neighbors.

Storing a list, the so-called *neighbor-list*, containing 26 relative addresses to the neighboring voxels, for each voxel is not desirable. It would result in a look-up table with $32^3 * 26 = 851968$ entries.

Fortunately the neighbor-lists can be classified into 27 different *neighbor cases*: The trivial case is, that all neighboring voxels are in the same block. This applies for the inner voxels of a block. For the border voxels, the re-

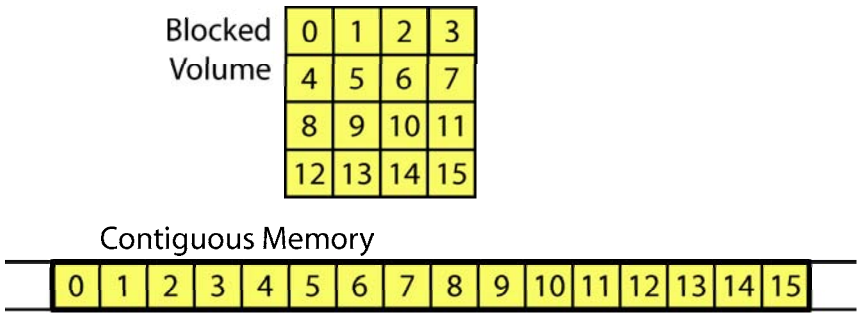


Figure 4.8: Data organization in contiguous memory. The blocks are ordered sequentially. The block index can be directly used for data block address calculation. The thick box denotes the memory areas allocated through the memory manager.

maining 26 cases apply. In each case the 26 relative addresses are identical for each voxel within the case. Ultimately the *neighbor look-up table* consists of 27 lists with 26 entries each, in total 702 entries. Storing the entries as 4 byte integers results in a neighbor look-up table of the size of 2808 bytes. This list fits and remains in the cache all the time due to frequent accesses. Therefore the access is very fast. In the *contiguous block volume* the data blocks are sequentially stored in a large contiguous memory block. Due to this layout the relative distances between a data block and its 26 neighbors are the same for each data block, so only one global *neighbor look-up table* is necessary.

Neighbor Case Determination Each neighbor case is denoted by a *case index*. The case index is derived from the position within the data block. For the one dimensional space there are only 3 cases: most left voxel, inner voxels and most right voxel. For the inner voxels addressing neighboring voxels is straightforward: each neighbor voxel lies within the same block as the active voxel. But for the left most voxel, its left neighbor lies in a completely different block, i.e. the left neighboring block of the active block. Analogous is this case for the right most voxel: The right neighbor voxel of this voxel lies in the right neighbor block.

Formula (4.1) depicts the *case index function*. In the calculation only simple arithmetic, bit-manipulation and shifting operations are used. All possible function values are shown in Table 4.1

$$CaseIndex_{dim} = ((((((Position_{dim}\&31) - 1)\&63)|1) + 1) >> 5) \tag{4.1}$$

The position value is first masked with 31 to get the position within the data block. Then the index is calculated from this local position. Table 4.1 shows all possible function values.

$Position_{dim}$	$CaseIndex_{dim}$
0	2
1..30	0
31	1

Table 4.1: All possible values of the case index function for a data block width of 32 voxels.

The case determination can be extended for n dimensions, resulting in 3^n cases. The case index for the three dimensional space are calculated as follows: for each axis (i.e. x,y and z) the one dimensional case index is calculated (data block size 32^3) from the absolute position in the volume data-set. Since the range of function values of the one dimensional case index function is limited to 0..2, the three dimensional case index is simply derived as depicted in formula (4.2). Figure 4.9 shows all indices assigned to their position on a cube. In Figure 4.10 the cases and two entries of the neighbor-list are depicted for the two dimensional case.

$$CaseIndex = CaseIndex_z * 9 + CaseIndex_y * 3 + CaseIndex_x \tag{4.2}$$

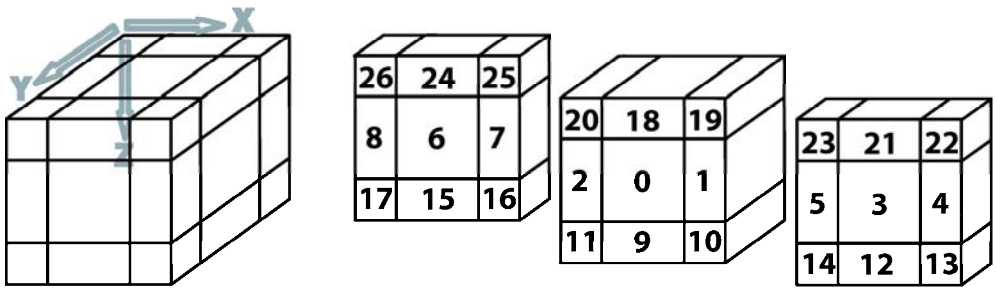


Figure 4.9: The 27 case indices calculated from the case index formula (4.2).

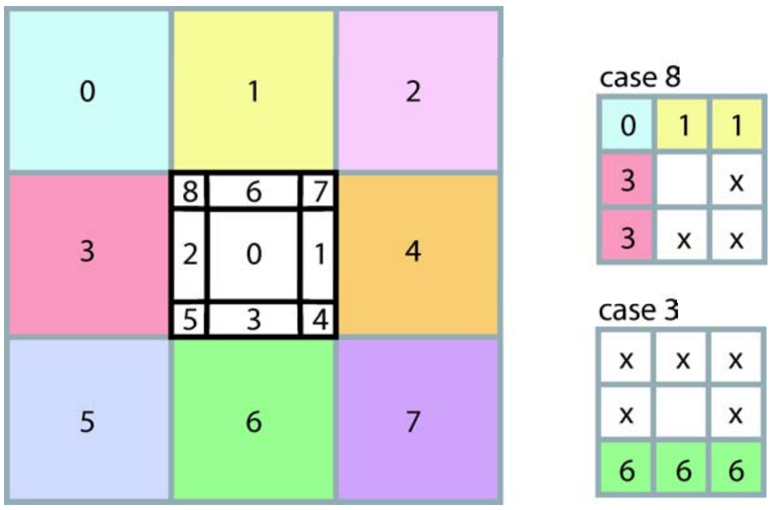


Figure 4.10: 9 different cases in two dimensional space. Each case has 8 entries (one for each neighbor). On the right there are sample tables of cases 8 and 3. The tables show in which neighbor block the neighboring voxel can be found. x denotes that the neighbor is in the same block.

Border Issue

With the relative neighbor addressing scheme a new issue is introduced, which affects the voxels at the volume border (see Figure 4.11). The relative address is calculated based on the addresses of the neighboring blocks, but at the border for some of the cases (see Figure 4.10) some neighboring blocks are missing.
This issue is solved by adding extra border blocks. This increases the size of the volume of the original size of X x Y x Z blocks to a minimum size of (X+1) x (Y+1) x (Z+1) blocks and a maximum size of (X+2) x (Y+2) x (Z+2) blocks as depicted in Figure 4.12 and Figure 4.13.
In general only adding border blocks at three sides of the volume block is sufficient, because the volume data does not fill up all blocks at the border (see Figure 4.12). In certain cases, i.e. the width of the volume is a multiple of 32 voxels (block width), which in the volumetric data-sets completely fills up blocks at the border, border blocks at up to additional 3 sides of the volume block are required as depicted in Figure 4.13.

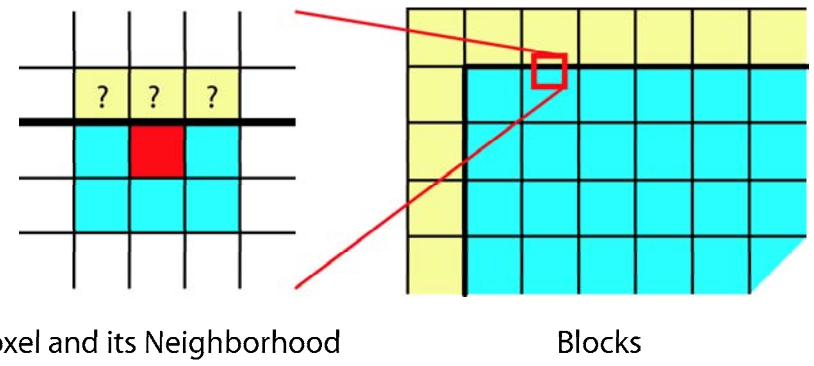


Figure 4.11: The voxel (red) is located at the border of a block which is also at the border of the volume. Also for such border voxels a relative neighbor addressing is required. It this case some of the neighboring voxels (denoted by "?") are located in a non-existing neighbor block.

Impact on the absolute addressing. The calculation of the block address is only affected marginaly by adding border blocks: The width variables (holding the width of the volume in blocks) W_x, W_y, W_z are replaced by the width variables W'_x, W'_y, W'_z which contain the original values W_x, W_y, W_z increased by 1 or 2 (the additional borders). Further to the block index an offset $W'_x * W'_y + W'_x + 1$ (i.e. the relative address of the first block containing volume data) is added. The voxel offset calculation is not modified because the bits which are relevant for the voxel offset are not affected by the border blocks.

Impact on the relative addressing. The adding of the border has no effect on the relative addressing scheme since it operates only locally within a block.

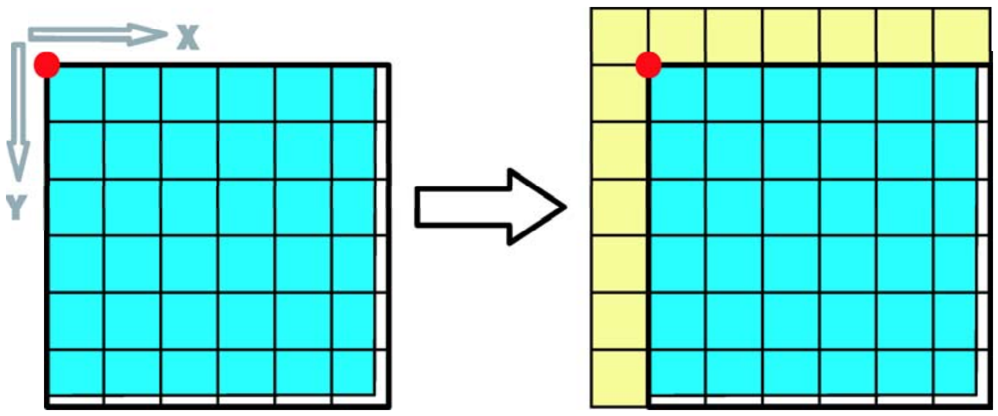


Figure 4.12: If the volume data does not fill up the data blocks completely on the right and/or bottom border, then only border data blocks at the left and top have to be added. The red dot denotes the origin of the coordinate system.

4.2.4 Fragmented Block Volume

In many cases during runtime the memory manager of an operating system is not able to allocate a contiguous memory block for storing the volumetric data, even though enough memory would be available. This is due to the fragmentation of the virtual address space during runtime. A solution to

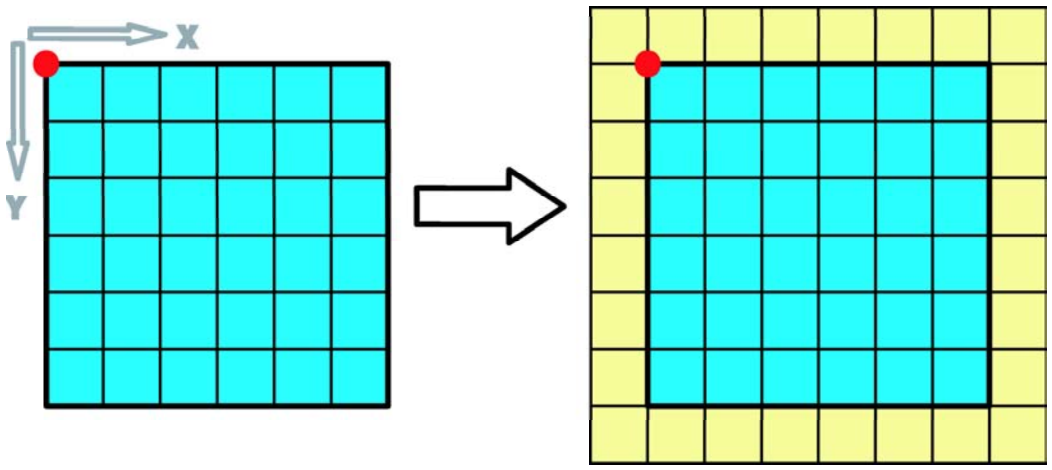


Figure 4.13: If the volume data fills up the data blocks completely on the right and/or bottom border, then border data blocks at all borders have to be added. The red dot denotes the origin of the coordinate system.

that problem is to split up the block volume in smaller pieces. To achieve that the data blocks are allowed to be irregularly ordered in virtual memory. The type of block volume is called *fragmented block volume*.

Absolute Addressing Instead of directly deriving the data block address from the block index, the data block address is looked up in a data block address list, see Figure 4.14 in the second calculation step. The organization of the data blocks in virtual memory is depicted in Figure 4.15.

Relative Addressing Basically the relative addressing scheme stays unmodified. Since in the *fragmented block volume* the data blocks are irregularly ordered, a single global *neighbor look-up table* is not sufficient because the relative position of the neighboring data blocks differs for each data block. Thus for each data block an own *neighbor look-up table* has to be maintained. This increases the overall memory requirement by 4.3% (64 kilobytes data + 2.8 kilobytes look-up table). As discussed later, this additional memory requirement can be quite large for large data-sets.

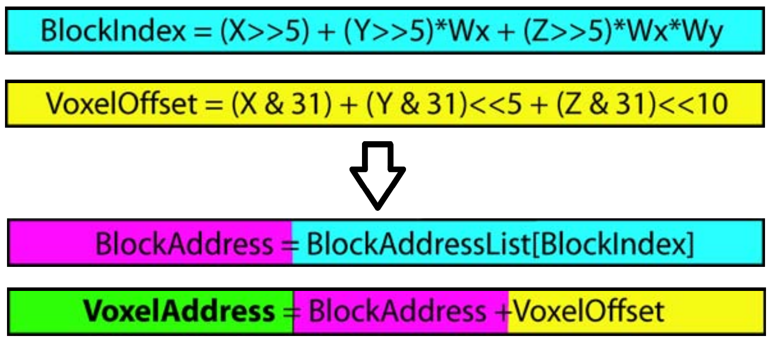

BlockIndex = (X>>5) + (Y>>5)*Wx + (Z>>5)*Wx*Wy
VoxelOffset = (X & 31) + (Y & 31)<<5 + (Z & 31)<<10
BlockAddress = BlockAddressList[BlockIndex]
VoxelAddress = BlockAddress + VoxelOffset

Figure 4.14: Absolute voxel address calculation for fragmented memory.

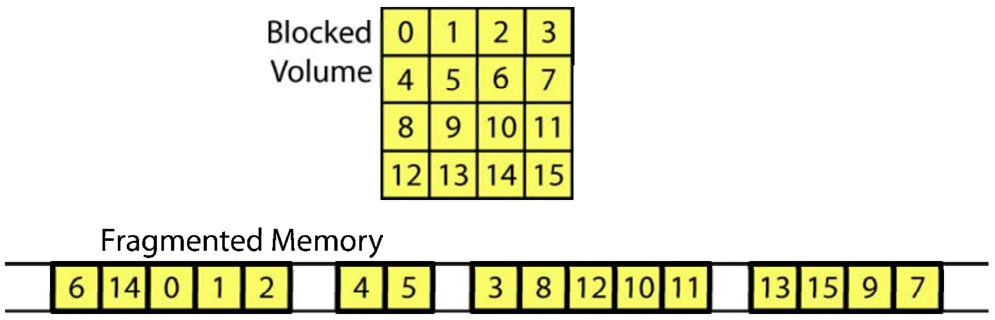


Figure 4.15: Data organization in fragmented memory. The blocks are irregularly ordered. The block index is used as index into a list containing the actual data block addresses. The thick border denotes the memory areas allocated through the memory manager.

4.2.5 Shared Block Volume

In the previous sections the contiguous and the *fragmented block volume* have been introduced. They provide an efficient access to the data and utilize memory space efficiently, but their memory requirements are not reduced, although it is an important issue for handling large data-sets.
Many data-sets contain large homogeneous regions, i.e. the border blocks or homogeneous regions (for example air) around a scanned object. These blocks contain the same data. Also when creating an large empty block volume filled with zero, all blocks contain the same value. Storing "empty" space

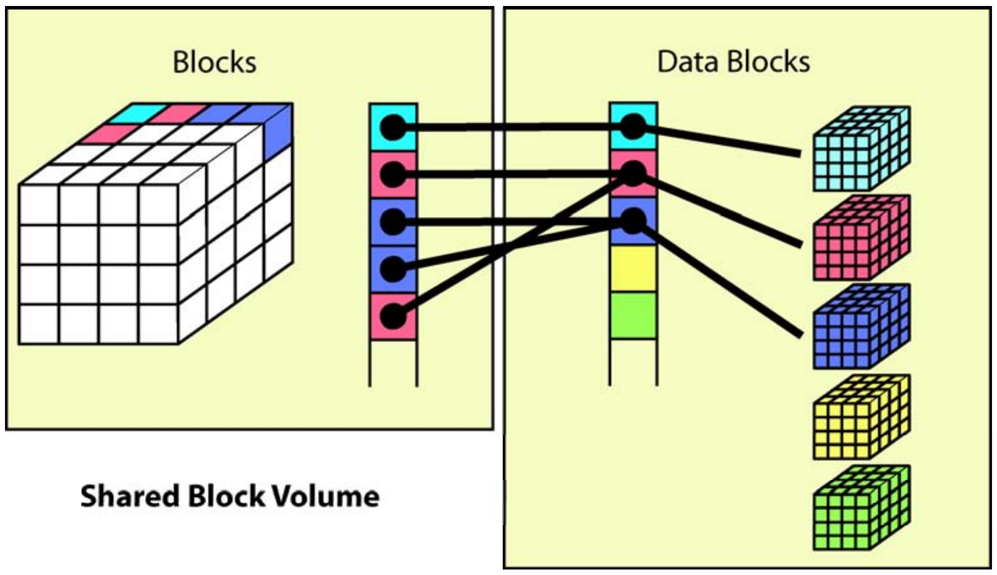


Figure 4.16: The *shared block volume*. In the *shared block volume* the mapping between the *Blocks* and the *Data Blocks* is a many-to-one mapping.

is not feasible, so actually the data of only one block is stored and is shared among the blocks containing the same data.
A step towards lower memory demands is the *shared block volume*. A *shared block volume* is an extension of the *fragmented block volume*.
In this concept a data block is extended in the following way: The block volume consists of blocks, which are references of data blocks: A block references exactly one data block, whereas a data block can be referenced, i.e. *shared*, by several blocks (see Figure 4.16). A *fragmented block volume* can be interpreted as a special case of a *shared block volume*: Each data block is referenced by exactly one block (see Figure 4.17).

Data Handling

At creation time, a *shared block volume* is empty: All voxels contain the density value zero, therefore each block contains the same values, thus only one data block is necessary for representing the data of all blocks so one data block is shared by all blocks (see Figure 4.18).

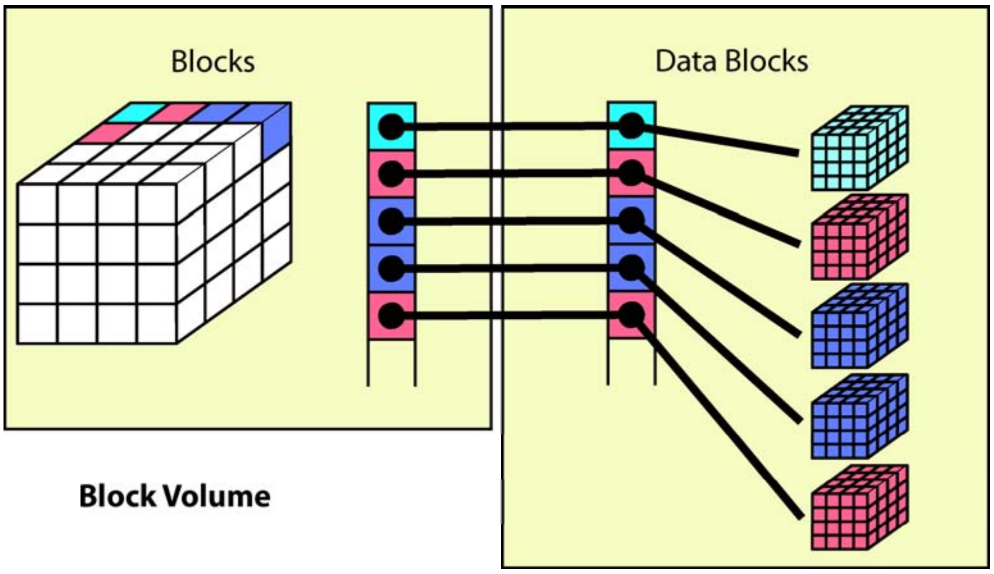


Figure 4.17: The block volume. In the contiguous or fragmented block volume there is a one-to-one mapping between *Blocks* and *Data Blocks*.

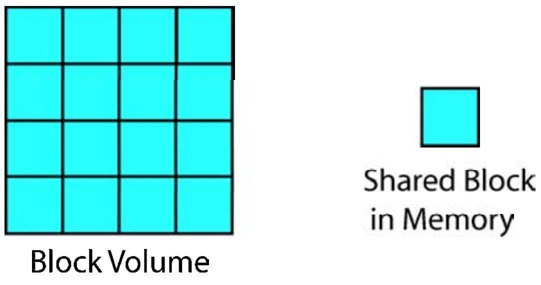


Figure 4.18: A *shared block volume* at initialization time. A single data block is shared by all blocks.

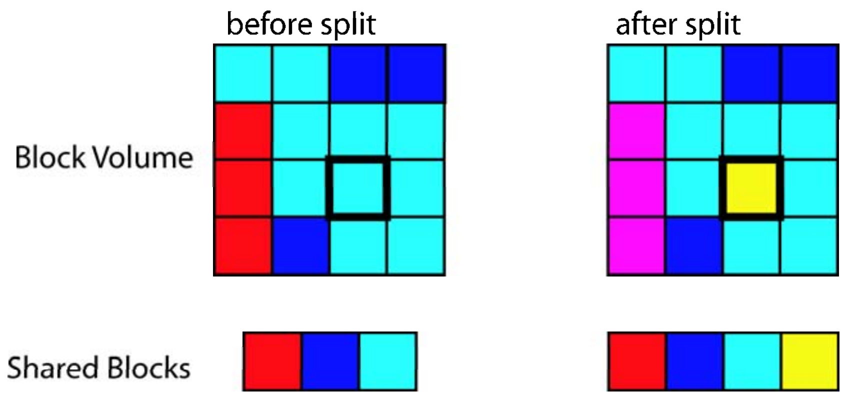


Figure 4.19: A *shared block volume* at split. A block is modified (thick border). A new block is created holding the modified block (yellow block).

Data Block Split If a new value is written to a data block, which differs from the value at the current location, then a replica of this data block is generated. To this replicated data block the new value is written at the desired location. First the data block is shared by n blocks, after this step the data block is shared by $n-1$ blocks, and the modified data held by the replicated data block is shared by one block (see Figure 4.19).

Block Data Merge If a set of m blocks are sharing a data block and a set of n blocks are sharing another data block, both sets are disjunct, but the data blocks contain identical data, both sets can be merged. One memory location containing one data block can be freed and the other data block is shared then by $n+m$ blocks. A merge requires comparing one data block with all other data blocks on voxel basis, which is computationally expensive, therefore it should be applied when no time-critical access to the volume occurs.

Addressing Issues

The *shared block volume* is basically a *fragmented block volume*, therefore each block has a *neighbor look-up table*. In a *shared block volume* where the addresses of the neighboring data blocks change during runtime, the *neighbor*

look-up tables of the affected blocks have to be updated to ensure consistent addressing of the data blocks. This achieved by using an *update look-up table* containing for each neighbor the fields of the *neighbor look-up table* to be updated.
If the address to a data block is changed by a data block split or a data block merge, then the neighbor look-up table of the block (i.e. the center block of a 3x3x3 cube of blocks) itself and its 26 neighbors have to be updated according to the changed address. For the center block the offset between the addresses of the old data block and the new data block is added to each entry in the *neighbor look-up table.* For each neighboring block, only those cases are updated, which in the center block is addressed.
Figure 4.20 shows the relation between the addressing and the *update look-up table* for updating the *neighbor look-up table.* For 2D the *update look-up table* contains $4 * 1 + 4 * 7 = 32$ and for 3D $8 * 1 + 12 * 7 + 6 * 49 = 386$ entries.

Border Handling

In the shared block volume, no additional border blocks are added. Instead a special border data block is provided for simulating the border. Instead of having extra blocks as in Figure 4.10 these data blocks are replaced by this special border data block at the according position as depicted in Figure 4.21.

4.2.6 Compressed Shared Block Volume

For data-sets with large homogeneous regions the *shared block volume* reduces the memory requirements significantly. Also having an extremely large volume initially does not require much memory because one data block is shared among all blocks in the block volume. The memory requirements can be further reduced with the utilization of compression methods.
Up-to now, only data structures storing uncompressed data have been presented. In these data structures the data is completely memory resident and instantly accessible. Thus a new issue comes up with the use of compression: Memory for concurrently storing the compressed *and* uncompressed data is required. The memory requirements of the compressed data is proportional

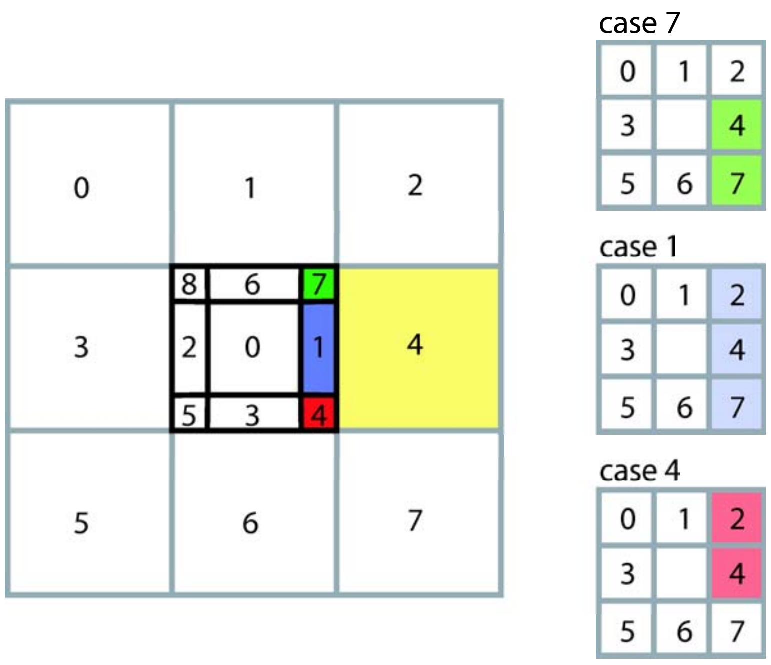


Figure 4.20: This figure shows the affected case (dark color) after a neighbor (light color) has changed its address. Only the affected fields are updated for each case. The following list is stored for neighbor 4 (yellow): (7,4),(7,7),(1,2),(1,4),(1,7),(4,2),(4,4). This list is to be interpreted as follows: *For neighbor 4 the following fields have to be updated: entry 4 of case 7, entry 7 of case 7, entry 2 of case 1, etc.*

with the size of the data-set, whereas the memory requirements for the uncompressed data depends on the way of processing the data. For example, processing the data slice by slice would require only a memory area of the size of a slice for processing. The slices to be processed would be decompressed successively into this memory area. The handling of the compressed data and its decompression is discussed in the following chapter.

Compression and Memory Usage

For the compression of the data blocks, a LZ77 descendent compression algorithm is used. It is the so-called LZO [11], the Lempel-Ziv-Oberhumer compression algorithm, which was developed by Markus Oberhumer in 1994.

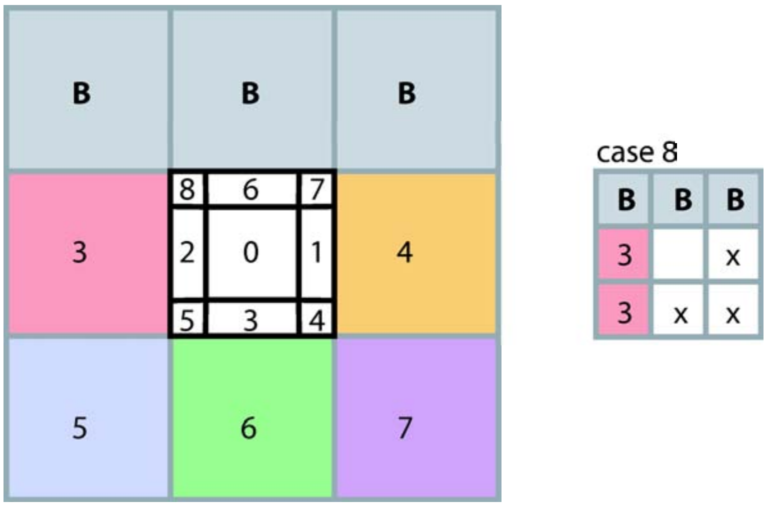


Figure 4.21: Relative addressing scheme at the border in the shared block layout.

It seems not to be very well known, but is used occasionally in the Linux community. The compression and decompression is slightly faster, than of other LZ77 variants, but is not required to compress better, because speed is the more important issue.
Compressed data requires usually less space than uncompressed data. For computed tomography data the compressed data requires around 30 to 80 percent less memory than the uncompressed data according to Table 4.2. The table shows the data reduction (0% no reduction, 100% full reduction) for various volumetric data-sets.

Type	Uncompressed	Compressed	Reduction
Visible Human (CT)	717MB	116MB	84%
Santa Claus (CT)	499MB	247MB	51%
Mouse Embryo (Photo)	592MB	388MB	34%
Visible Human (Photo)	1613MB	357MB	78%
Analytic Function (Synthetic)	2048MB	462MB	77%

Table 4.2: Various types of volumetric data compressed with the LZO algorithm [11].

4.3 Volume Processing Hierarchy

In this section a multi-layer abstraction of volume processing is introduced. This is an abstraction of the interaction between an algorithm processing the actual data and the structures handling the actual volumetric data-set. On the top level there is the algorithm operating on the volumetric data. On the bottom level, there is the volumetric data itself. In between, up to two additional layers are introduced. One layer providing an consistent addressing scheme to the algorithm layer above, and the other layer below for preparing the data for access by communicating with the data layer below.

In Figure 4.22 this multiple layer abstraction is depicted: The *Volume Data* is the actual volumetric data organized in a specific layout. The *Volume Handler* provides access to volumetric data and handles memory allocation and addressing of the data. The *Volume Iterator* translates addressing schemes and implements specific access patterns, for example slice by slice access. It hides the actual volumetric data layout. The *Algorithm* itself operates on the data.

For example, an algorithm can access memory resident data directly without any additional data handling processes in between, whereas compressed data or out-of-core methods require some preprocessing, i.e. data decompression, before the actual data can be accessed.

In the following four examples of volumetric data processing abstractions are explained:

a. Direct Access The algorithm operates directly on the memory resident volumetric data. For example, the data-set is stored in a contiguous memory block using the sequential layout. The algorithm can directly access a voxel by calculating the voxel address from its position.

b. Compression and Out-of-Core . With out-of-core processing an additional layer, the *Volume Handler* is inserted: it prepares the volumetric data in such way, that it can be processed by the algorithm like memory resident data. For example, the Volume Handler holds a buffer for a single slice, so the *Algorithm* triggers the *Volume Handler* to load (and decompress) a specific slice into the buffer.

c. Hidden Data Structures The *Volume Iterator* hides the actual volume organization. It provides access to the data in specific patterns. For example the volumetric data is stored in a memory resident block volume, but provides an interface for slice-wise access. The *Algorithm* requires access to a slice using sequential addressing, so the *Volume Iterator* translates the sequential addressing into the block volume addressing scheme.

d. A combination of b and c Since *Volume Iterator* hides the actual volume layout from the *Algorithm*, the data can be processed transparently regardless of the layout of the data. The *Volume Iterator* and the *Volume Handler* interact in a way that the resources for the volumetric data access are utilized optimally based on the specific access patterns.

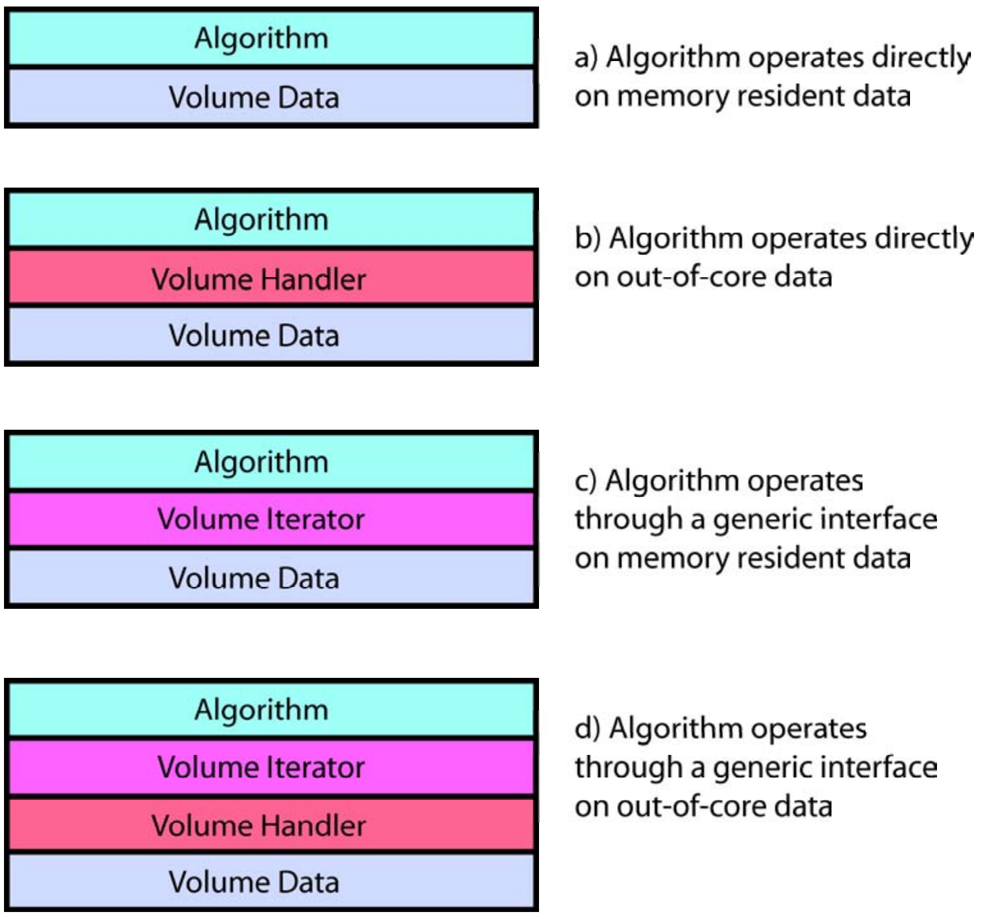

Algorithm
Volume Data
a) Algorithm operates directly on memory resident data
Algorithm
Volume Handler
Volume Data
b) Algorithm operates directly on out-of-core data
Algorithm
Volume Iterator
Volume Data
c) Algorithm operates through a generic interface on memory resident data
Algorithm
Volume Iterator
Volume Handler
Volume Data
d) Algorithm operates through a generic interface on out-of-core data

Figure 4.22: The volume processing hierarchy.

Chapter 5

Memory Allocation Strategies

5.1 Introduction

In this chapter two memory allocation strategies for *compressed shared block volumes* are presented.
In a *compressed shared block volume* it is not possible to directly access the data as opposed to the memory resident sequential volume. Before the data can be accessed it has to be uncompressed. Additional memory has to be allocated for storing the uncompressed data. Allocating memory for the whole block volume is not feasible due to the high memory requirement.
A solution for reducing the memory demands during runtime is the allocation of memory for only those data blocks which are currently processed by an algorithm. For example, for extracting single slices from a block volume only the data blocks intersected by the slice plane are required. Since frequently allocating and freeing memory is a time-consuming process, an efficient memory allocation process for these memory areas is desirable.
The memory management is implemented in the volume iterator and the volume handler layer of the volume processing hierarchy. The volume iterator implements a specific access pattern. An algorithm utilizes the volume iterator which matches the access pattern of the algorithm. For an algorithm processing data slice-wise this would be an iterator implementing a slice-wise access to the data. Since the access pattern is known, the volume handler

is able to prepare the required data blocks while avoiding non-relevant data blocks.

5.2 Strategy 1: Dynamic Block Allocation

This strategy manages the allocation of memory areas to data blocks. Basically a list of all currently uncompressed data blocks is maintained. Each of these blocks point to a memory area containing the uncompressed data. This memory area is referred to as *processing memory block*.
When a compressed data block has to be uncompressed, an uncompressed data block from the list is taken, which is not processed at the time and has a certain minimum age. This means the difference between the actual time and the time when the data block was processed for the last time is above a specified limit, and its *processing memory block* is assigned to the block to uncompress. If such an uncompressed data block cannot be found, a new *processing memory block* for the block to uncompress is allocated and is added to the list.

5.2.1 Components

In this subsection the dynamic block allocation is explained on the basis of the various layers of the volume processing hierarchy.

Volume Data

For storing the volume data, the *compressed shared block volume* model is used. Additionally to each data block the following variable is added: a reference counter holding the number of volume iterators that are using the data block.

Volume Handler

The volume handler maintains a list of all accessible blocks, i.e. all data blocks which are uncompressed and have a pointer to a *processing memory*

block. A data block can have two exclusive states:
1. *Frozen.* In this state, the data block is in a compressed state and the pointer to the *processing memory block* is null.
2. *Defrosted.* In this state the data block is in a uncompressed state, stored in a *processing memory block*, and is directly accessible.

Volume Iterator

A volume iterator contains an additional structure, which holds a list of the data blocks used by the iterator. At initialization time, the number of *defrosted* data blocks in the list is zero. Before the blocks can be accessed, the volume iterator prepares the required blocks for being accessed according to its access pattern.

5.2.2 Internal Operating Sequence

An *algorithm*, for example a slicer or renderer, operates on a specific set of blocks. The blocks are accessed in a predefined order. In many cases the set of blocks can be divided into sub-sets which must be accessible at the same time or the order of processing is not important. The sub-sets are not necessarily required to be disjunct. Processing the blocks in such sub-sets reduces the memory requirements during runtime. Per iteration step such a sub-set is processed. The *volume iterator* operates in the following way:

1. Initialization. The *processed blocks list*, holding the blocks being processed in an iteration step, is emptied.

2. Processing loop. During each iteration step in the processing loop, a different set of blocks is operated on. In many cases the set of blocks used before and the new set of blocks are not disjunct. This accelerates uncompressing of blocks. The following process is depicted in Figure 5.1:

    (a) Initialization. In the initialization step the blocks which should be processed in this iteration step are prepared.

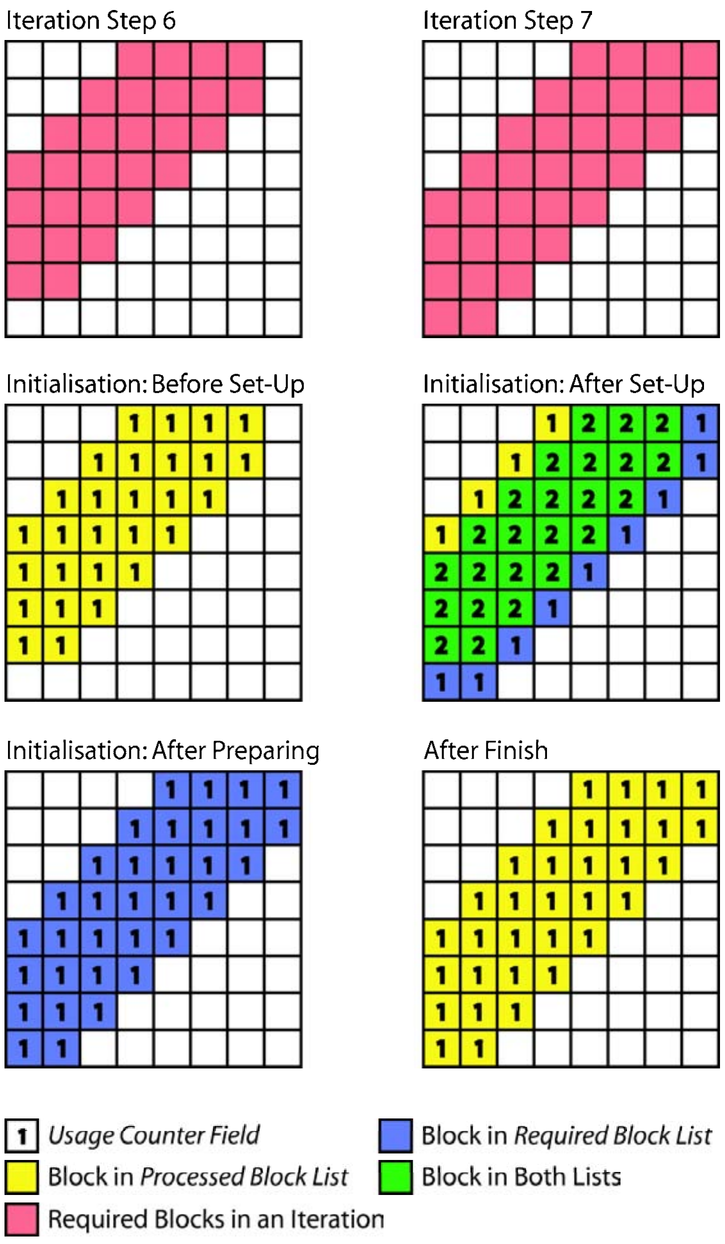

Iteration Step 6
Iteration Step 7
Initialisation: Before Set-Up
Initialisation: After Set-Up
Initialisation: After Preparing
After Finish
Usage Counter Field
Block in Required Block List
Block in Processed Block List
Block in Both Lists
Required Blocks in an Iteration

Figure 5.1: Dynamic Block Allocation. Example of an iteration step (iteration step 7) of the rendering algorithm of Grimm et al. [6].

i. Set-up. The *iteration counter*, is increased by one. The *required blocks list* is emptied, then the blocks which should be processed in this iteration step are added to the list. For each block the following steps are performed:

   A. If the *iteration number field*, holding the number of the iteration when the block was used for the last time, of its data block is not equal to the current *iteration counter* omit this block. That avoids unwanted multiple additions of the same data block to the list.

   B. The *iteration number field* is set to the current *iteration counter* value.

   C. The *usage counter field* is increased by one.

   This ensures that a data block appears in the *required block list* only once.

ii. Preparing. In this step all blocks from the *required blocks list* are prepared for processing. First the *usage counter field* of each block in the *processed block list* which contains the data blocks used in the previous iteration step is decreased by one and after that the list is cleared. Now, for each data block in the *required blocks list* which does not point to a memory block, a memory block is acquired as follows: Basically the memory allocation algorithm maintains a list containing all currently *defrosted data blocks.* These data blocks are stored in a queue. When a free memory block is needed a data block is retrieved from the queue. Then it is checked whether the data block is in use or not. If not then this data block is turned into the *frozen* state and the memory area is reused for the new data block. If the data block is in use, then it is appended at the end of the queue and the next data block is retrieved from the queue. If no free memory area can be found, simply a new memory area is allocated through the system memory manager.

After for a data block a valid memory area is allocated it is turned into the *defrosted* state. The compressed data is decompressed into the memory area. Finally the neighbor look-up tables for the relative neighbor addressing scheme have to be updated accordingly.

(b) Processing. The data blocks in the *required blocks list* can be accessed like a memory resident block volume.

(c) Finish. The *processed blocks list* is filled with the contents of the *required blocks list.*

3. Clean-Up. The *usage counter field* of each block in the *processed blocks list* which contains the blocks used in the previous iteration step is decreased by one and after that the list is cleared.

5.2.3 Conclusion

Initially the memory requirements comprise the compressed data and the neighbor look-up tables of each block in the shared block volume. During runtime, additional memory for storing the uncompressed data is allocated. Since in the shared block volume for each data block a neighbor look-up table for the relative neighbor addressing has to be stored, the space required for the tables linearly increases with the size of the data-set. The neighbor look-up table has the the size of 2.8 kilobytes, therefore the memory requirement for the shared block volume is at least 4.3 percent (2.8 kilobytes of 64 kilobytes) of the original uncompressed size. Thus space required for the neighbor look-up tables is significantly large for large data-sets. It turned out that this issue makes this strategy impractical for large data-sets.
For example, a 2048x2048x2048 volumetric data-set, which has a total size of 16 gigabytes, is divided into 64x64x64 blocks, in total 262144 blocks. This would result into a memory requirement of 716 megabytes for the neighbor look-up tables only. This is not feasible on computers with limited memory resources, where the neighbor look-up tables would use up most of the memory. Since this issue cannot be solved easily, a new strategy circumventing

this problem is presented.

5.3 Strategy 2: Block Mapping

For this strategy the memory requirements for processing is constant and independent from the data-set size. This is achieved by utilizing a block volume of a fixed size. In this approach, the sequential layout is applied in a three level arrangement (see Figure 5.2, compare it with Figure 4.5).
The block volume consists of a block of *sub-volumes*, which itself consist of blocks. There are two types of sub-volumes: *data sub-volumes* holding the *data blocks* in a compressed state and one *processing sub-volume* holding the *data blocks* in an uncompressed state. The blocks in the *processing sub-volume* are referred to as *processing blocks*. To achieve the independency of the memory requirements for processing from the data-set size, a single *processing sub-volume* is shared among all *data sub-volumes*, regardless of the size of the overall block volume. Internally, all data blocks at a specific position in each *data sub-volume* share a single *processing block* of the *processing sub-volume* at the according position. This is referred to as *mapped blocks*, i.e. the data blocks are *mapped* onto a single *processing block* (see Figure 5.3).
The size of a sub-volume is fixed and the whole block volume consists of a fixed number of blocks. The sub-volumes are cube-shaped with a width of 2^n blocks, where n is a fixed number. At the border the address wraps around to simulate a large, theoretically infinite volume data-set. This is achieved without any additional computational costs by simple masking operations. n must have at least the value of 2 to avoid data block conflicts at the wrap-around (see Figure 5.4).
Also, different from the dynamic block allocation strategy, the size of the overall block volume is fixed regardless of the size of the actual volumetric data-set.
In this strategy the previously introduced *compressed shared block volume* layout, which is used in the dynamic block allocation, is not required. Since the processing sub-volume is a block-volume of fixed size for only storing

uncompressed data, the fragmented block volume is used as a basis for the *processing sub-volume*. The contiguous block volume cannot be used because of the cyclic neighbour addressing introduced in the next section.
For example, assume a block volume of the size 2048x2048x2048. The sub-volume size is 4x4x4 blocks, which is 128x128x128 in voxels. Thus the block volume consists of 16x16x16 data sub-volumes. The processing sub-volume has also the size of 4x4x4 blocks. Thus the processing sub-volume is shared among $16^3 = 4096$ data sub-volumes.

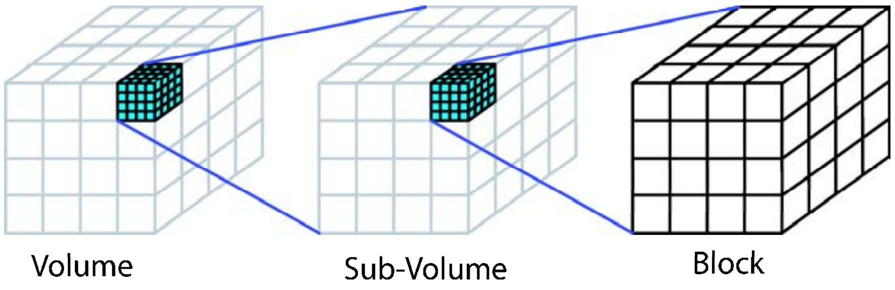


Figure 5.2: Block Mapping: Three-level application of the sequential layout.

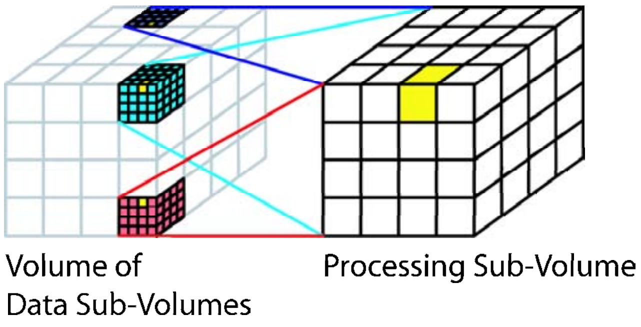


Figure 5.3: Mapping of *Data Sub-Volumes* to a single *Processing Sub-Volume*. A processing block contains a list of all associated data blocks.

5.3.1 Addressing the Sub-Volumes

The addressing in this layout is as easy as for a simple block volume. At the very beginning the higher bits, from which the BlockIndex is calculated

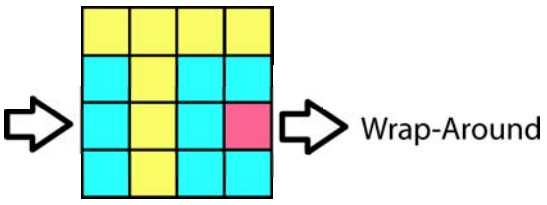


Figure 5.4: In the processing sub-volume with wrap-around the width of the sub-volume must be at least 4 blocks to avoid a self-overlapp of the neighboring blocks (cyan) of a block (red).

are split up into two parts. The higher part is used for calculating the sub-volume index. The lower bits are used for the BlockIndex to address the block within the processing sub-volume. This is depicted in Figure 5.5

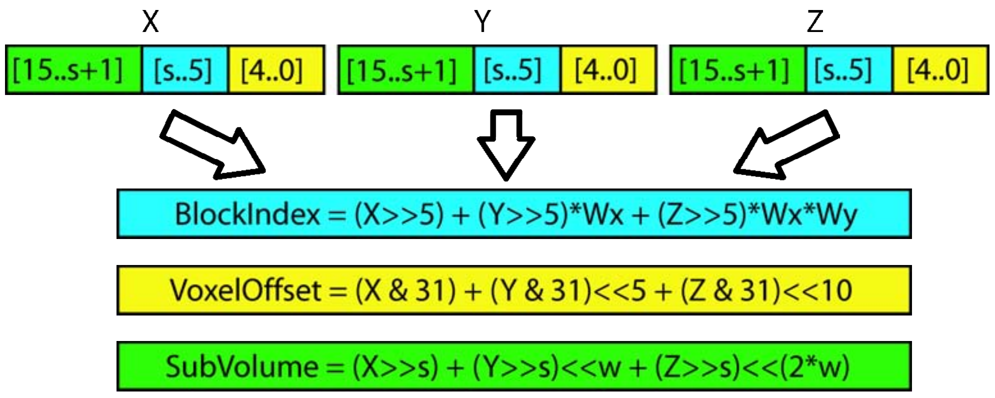

X
Y
Z
[15..s+1]
[s..5]
[4..0]
[15..s+1]
[s..5]
[4..0]
[15..s+1]
[s..5]
[4..0]
BlockIndex = (X>>5) + (Y>>5)*Wx + (Z>>5)*Wx*Wy
VoxelOffset = (X & 31) + (Y & 31)<<5 + (Z & 31)<<10
SubVolume = (X>>s) + (Y>>s)<<w + (Z>>s)<<(2*w)

Figure 5.5: Block index and voxel address calculation in a mapped block volume. For the voxel address, only the 5 lowest bits are used, for the block index the bits above the 5 lowest are used. The higher bits are used for the sub-volume index. The variable s denotes the logarithmic width in blocks of a sub-volume. 2^w is the actual width of a sub-volume. w is the logarithmic width of the overall block volume in sub-volumes. The width in blocks of the overall block volume would be 2^{s+w}.

5.3.2 Components

In this subsection the various layers according to the volume processing hierarchy are explained.

Volume Data

In the data domain the block volume consists of a number of *data sub-volumes.* For processing, all *data sub-volumes* are represented by a single *processing sub-volume* structure. The data blocks of each *data sub-volume* are mapped to the processing block in the *processing sub-volume* at the corresponding location as depicted in Figure 5.3.

Volume Handler

The volume handler is represented by the *processing sub-volume.* It maintains the *processing blocks* which hold the uncompressed data. It also decompresses data blocks when required.

Volume Iterator

The volume iterator accesses the block volume in an order that the number of data block replacements are minimized, as far as it is the inter-block dependencies allow it. For example an iterator extracting slices the order of blocks is not important, whereas for a ray-caster processing blocks front to back the processing order of the blocks is important, because the blocks further behind depend on the blocks in front of them.

5.3.3 Out-of-core Volumetric Data Processing

The block mapping strategy is extended with an out-of-core data handling feature. This out-of-core approach operates only on the compressed data blocks.

For each data block a state is maintained describing whether it is memory resident or not. Thus the volume handler knows when to retrieve a data block from the mass storage device. For each data block the time when it

was accessed the last time is stored, called *last access time*. The time can be the real-time or simply a counter increased over time, not necessary regularly, but in such a way that a temporal order is achieved. In the implementation such an *iteration counter* was used. After a *volume iterator* iteration step has ended, the set of all memory resident data blocks is analyzed for data blocks whose difference between the current time and their *last access time* exceeds a specific *time limit*. This is referred to as *block memory analysis*. If the limit was exceeded, the block is *swapped out*: If it is not yet stored on the mass storage device, it is written to it, then its memory is freed.
Another limit is the *memory usage limit*. This limit is fixed during an iteration step. This memory limit is a soft limit, which means that the algorithm tries to keep the memory usage below this limit, but it is allowed to exceed the limit if necessary. As long as the memory usage stays below this limit, no swapping out is necessary and the time limit is increased by one after each *block memory analysis*. But, if the total memory usage is above the limit, the blocks exceeding the *time limit* are swapped out thus reducing the memory usage and the time limit is decreased by one.
Data blocks are *swapped in* when the volume handler prepares a data block for being accessed by an iterator, but the data block is not memory resident. In that case memory is allocated and the data block is loaded from the mass storage device. Then it is decompressed into its associated *processing block*.

5.3.4 Operation of the Mapped Block Volume

The basic operation order of the mapped block volume is depicted in Figure 5.6. It shows a quadratic (can be also rectangular) area of blocks to be processed. a) processing of blocks 0-2. Data is decompressed (if required) into the memory blocks and can be processed. b) Here the first block during this process has to be overwritten: data block 0 is replaced by data block 4. c-f) During further processing more blocks have to be replaced. g) This is the state of the processing volume after the traversal of the rectangular area.

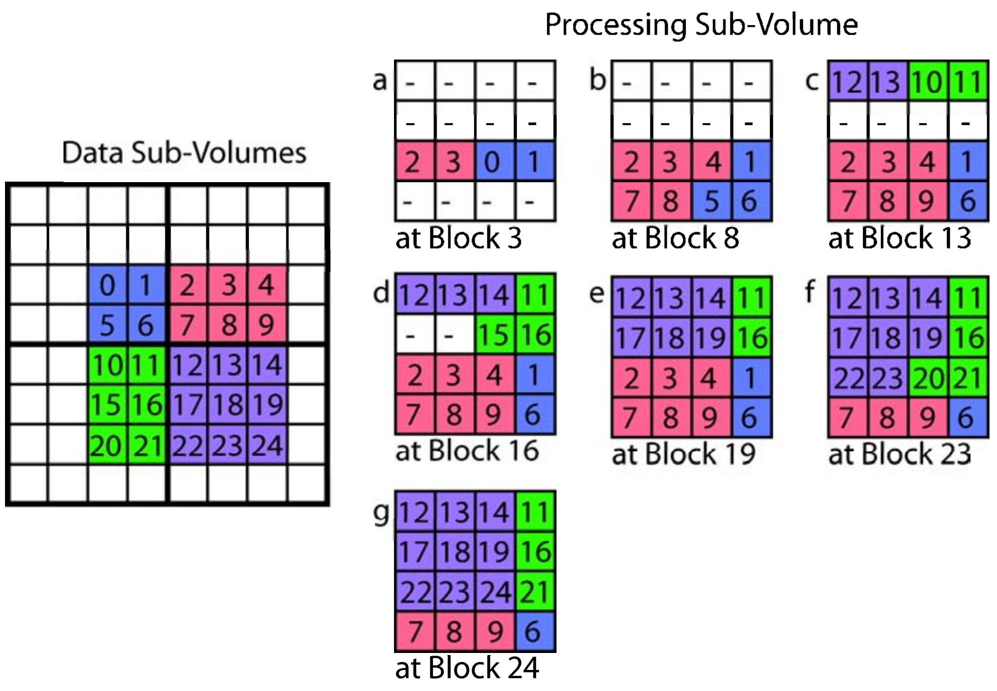


Figure 5.6: The mapped block volume while processing. This Figure shows a rectangular area of blocks to be processed. The numbers show the order of processing. On the left side the rectangular area to be processed is shown, on the right the states of the processing sub-volume while processing.

5.3.5 Conclusion

The advantage of this strategy is, that the resources needed for processing have a fixed size. Also the size of the data-set has no impact on the amount of resources required for processing. For multiprocessing purposes several processing sub-volumes can operate on the same volumetric data-set.

Chapter 6

Implementation

6.1 Introduction

In this section implementation schemes for both strategies are presented. The implementation was integrated into an existing volume renderer [2].

6.2 Dynamic Block Allocation

Algorithm An *Algorithm*, processing the volumetric data, utilizes a *Volume Iterator* for preparing the required data (black line in Figure 6.1 from the *Algorithm* to the *Volume Iterator*)

Volume Iterator It prepares the data for the *Algorithm.* It processes blocks in a specific order: A slicer processes all blocks intersecting the slicer, a renderer processes blocks for example, in a front to back order. It utilizes a *Defroster* for accessing the actual *Block Volume.*

Defroster With this structure an *Volume Iterator* interfaces with the *Block Manager* (black line in Figure 6.1 from the *Defroster* to the *Block Manager*) . It maintains the specific set of blocks required by the *Volume Iterator* (green line in Figure 6.1 from the *Defroster* to the *Blocks*)

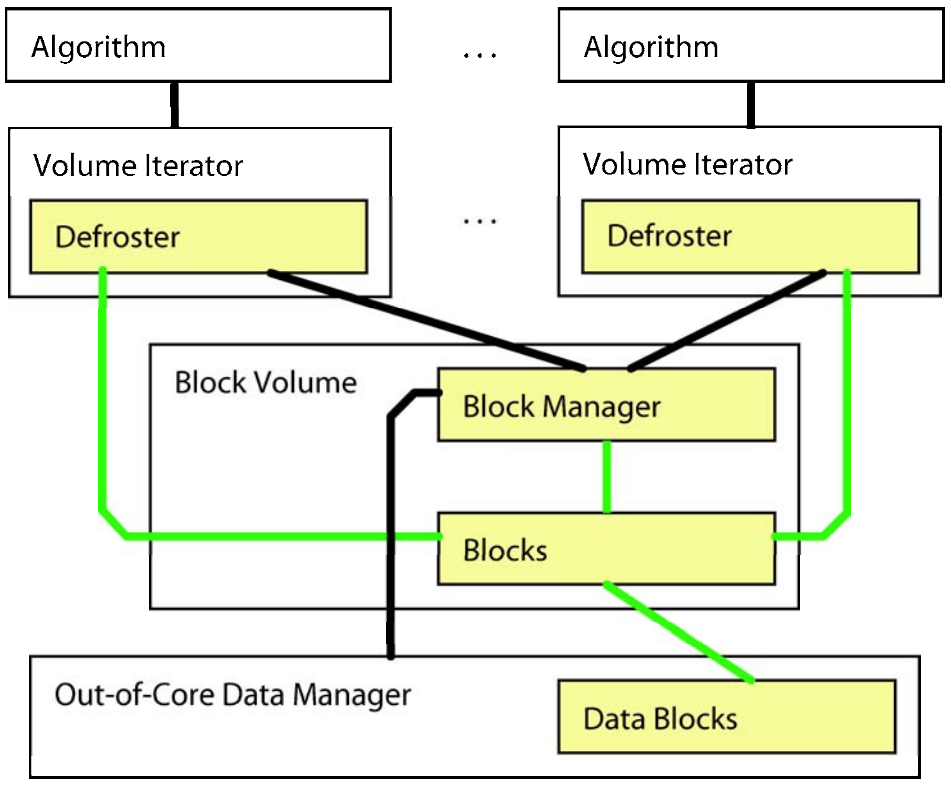


Figure 6.1: *Dynamic Block Allocation* implementation scheme. Boxes denote classes (yellow: inner classes), black lines denote communication paths, green lines denote references.

Block Volume This structure represents the shared block volume. It holds the properties of a volume, a list of *Blocks* and a *Block Manager*.

Block Manager The *Block Volume* utilizes a *Block Manager* for controlling the preparation of the *Blocks* during runtime. It is responsible for the handling of the *Blocks* (green line in Figure 6.1 from the *Block Manager* to the *Blocks*). This is done by interacting with the *Out-of-Core Data Manager* (black line in Figure 6.1 from the *Block Manager* to the *Out-of-Core Data Manager*)

Block A generic structure for a block. It contains the neighbor look-up table for the relative addressing scheme. It holds the uncompressed data from its associated *Data Block* (green line in Figure 6.1 from the *Blocks* to the *Data Blocks*)

Out-of-Core Data Manager This structure holds the actual *Data Blocks*. It is responsible for the compression and the decompression of the data. It also controls the out-of-core process. It maintains a list of data blocks currently hold in memory.

Data Block This structure holds the actual volume data and auxiliary structures related to the data.

6.3 Mapped Block Volume

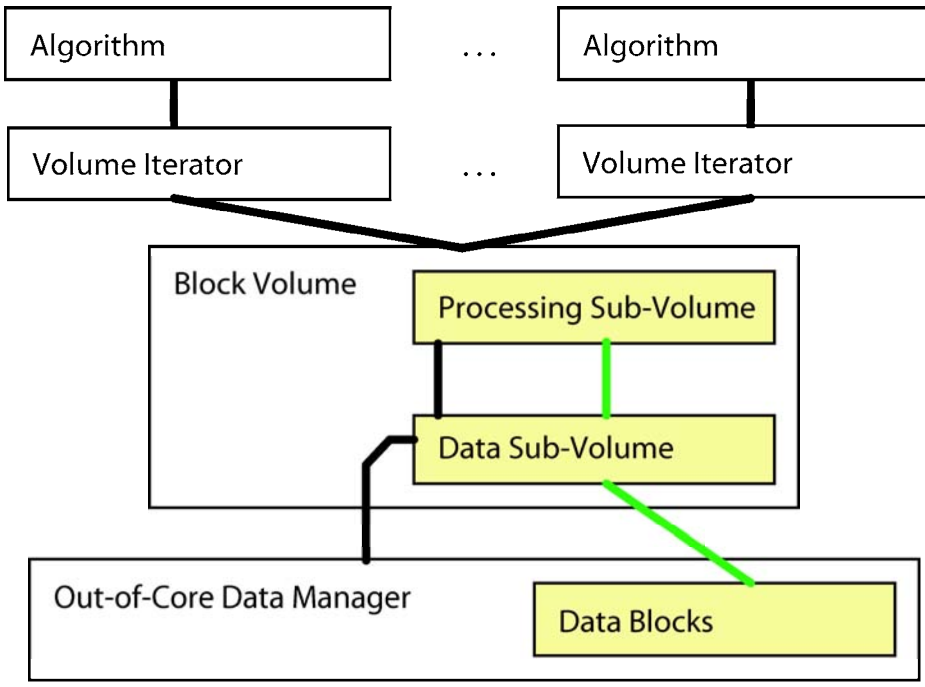


Figure 6.2: *Mapped Block Volume* implementation scheme. Yellow Boxes denote classes (yellow: inner classes), black lines denote communication paths, green lines denote references.

Algorithm An *Algorithm*, processing the volumetric data, utilizes a *Volume Iterator* for preparing the required data (black line in Figure 6.2 from the *Algorithm* to the *Volume Iterator*)

Volume Iterator The volume iterator prepares the data for the *Algorithm*. It processes blocks in a specific order: A slicer processes all blocks intersecting the slicer, a renderer processes blocks for example, in a front to back order. It accesses the actual *Block Volume* (black line in Figure 6.2 from the *Volume Iterator* to the *Block Volume*

Block Volume This structure holds the *Processing Sub-Volume* and the *Data Sub-Volume*. The *Processing Sub-Volume* triggers the *Data Sub-Volume* (black line in Figure 6.2 from the *Processing Sub-Volume* to the *Data Sub-Volume*) to retrieve *Data Blocks* from the *Out-of-Core Data Manager* (black line in Figure 6.2 from the *Data Sub-Volume* to the *Out-of-Core Data Manager*)

Processing Sub-Volume This structure implements a processing sub-volume. It contains a list of its *Data Sub-Volumes* (green line in Figure 6.2 from the *Processing Sub-Volume* to the *Data Sub-Volume*). Preparing specific *Data Blocks* is done by interacting with the *Out-of-Core Data Manager* through the *Data Sub-Volume* (black line in Figure 6.1 from the *Processing Sub-Volume* to the *Data Sub-Volume*)

Data Sub-Volume This structure implements a data sub-volume. It contains a list of its *Data Blocks* (green line in Figure 6.2 from the *Processing Sub-Volume* to the *Data Sub-Volume*). It interacts with the *Out-of-Core Data Manager* (black line in Figure 6.2 from the *Block Manager* to the *Out-of-Core Data Manager*) to load specific data blocks.

Out-of-Core Data Manager This structure holds the actual *Data Blocks*. It is responsible for the compression and the decompression of the data.

It also controls the out-of-core process. It maintains a list of data blocks currently held in memory.

Chapter 7

Results

The implementation was integrated into the rendering framework *Raybooster* developed by Bruckner [2]. A screen shot of the framework is depicted in Figure 7.1. For generating the results mainly the ray-casting algorithm developed by Bruckner [2] is used.
For generating the results various computer tomography data-sets have been used: *Visible Male* (Figure 7.7), *Christmas Tree* (Figure 7.8), *Santa Claus* (Figure 7.9) and *Skewed Head* (Figure 7.10)

7.1 General Remarks

All timings and tests in the results chapter have been performed on an AMD Athlon System with a 1.2 gigahertz CPU and 512 megabytes of memory running Microsoft Windows 2000 Professional.

7.1.1 Data Access

While processing a block, the CPU reads from the following memory locations:

- *The Data Block.* It is a block of 32x32x32 16bit values containing the density information. It has the size of 64 kilobytes.

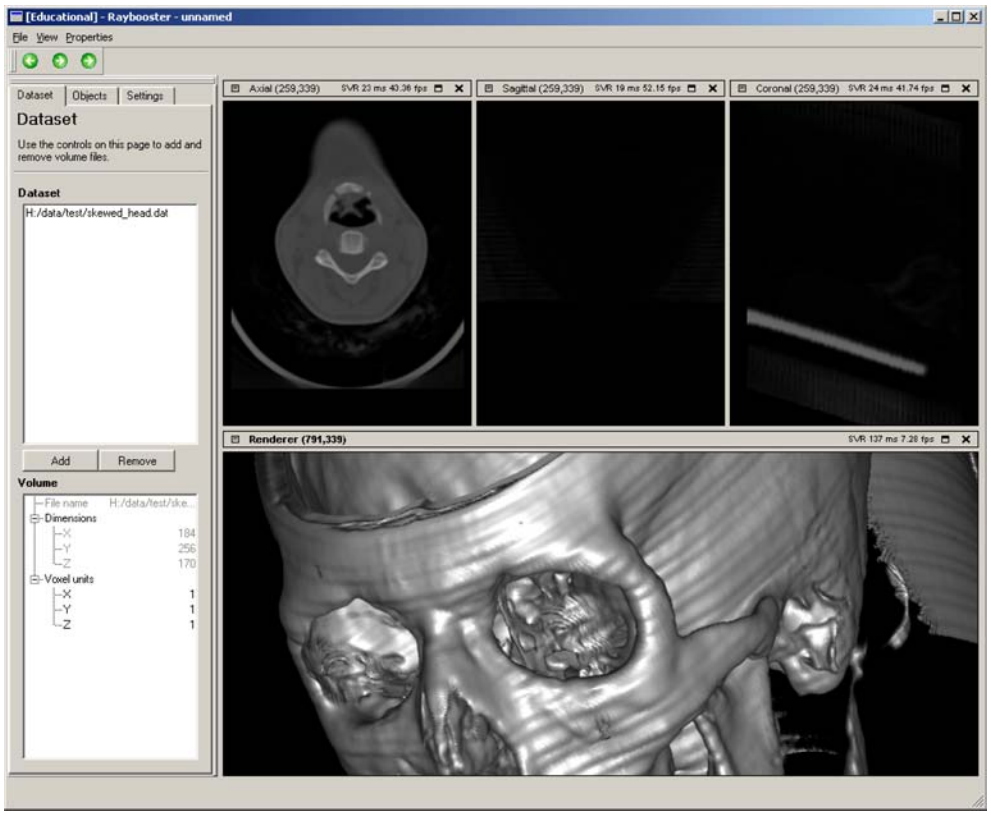

[Educational] - Raybooster - unnamed
File View Properties
Dataset Objects Settings
Dataset
Use the controls on this page to add and remove volume files.
Dataset
H:/data/test/skewed_head.dat
Add Remove
Volume
File name H:/data/test/ske...
Dimensions
X 184
Y 256
Z 170
Voxel units
X 1
Y 1
Z 1
Axial (259,339) SVR 23 ms 40.36 fps
Sagittal (259,339) SVR 19 ms 52.15 fps
Coronal (259,339) SVR 24 ms 41.74 fps
Renderer (791,339) SVR 137 ms 7.28 fps

Figure 7.1: The *Raybooster* rendering framework.

- *Neighbor Look-Up Table.* This table, containing 27x26 entries stored in a 32bit variable each, has the size of 2.8 kilobytes.
- *Other Variables* are required for handling the block volume, which make up several bytes in total.
- *Algorithm Specific Variables* are required by the algorithm operating on the data.

In total, processing a single block is bounded to roughly 70 kilobytes of memory. Several blocks fit smoothly into a 512KB level 2 cache. The level 2 cache is at least 8-way associative (Intel Pentium 4) or 16-way (AMD Athlon) associative, thus these cache areas of the memory locations specified above, combined with the frequent accesses to them, are unlikely to be superseded by various other reading and writing accesses requiring a cache update from the main-memory.

7.1.2 Decompression Speed

We assume a hard-disk with 25 megabytes per second interface speed and a decompression algorithm with an output speed of 100 megabytes per second. The data has an uncompressed size of 1000 megabytes and a compressed size of 500 megabytes. Loading the uncompressed data from the hard-disk takes 40 seconds. Loading the compressed data takes 20 seconds and decompressing takes 10 seconds, in total 30 seconds, which is 10 seconds faster than loading the uncompressed data. Experimental results with the LZO decompression algorithm (sub-type *lzo1x*) [11] (Table 7.1) support this assumption. The data is read sequentially slice-wise from the hard-disk.

Data-set	Uncompr. size	Compr. size	Loading uncompr.	Loading compr.
Visible Male (CT)	717 MB	116 MB	35 sec	8 sec
Christmas Tree (CT)	499 MB	247 MB	43 sec	17 sec
Visible Male (Photo)	1613 MB	357 MB	87 sec	25 sec
Mouse Embryo (Photo)	592 MB	388 MB	72 sec	27 sec
Analytic Function (Synthetic)	2048 MB	462 MB	105 sec	24 sec

Table 7.1: Comparison of loading uncompressed data with loading compressed data (with decompressing) slice wise.

7.2 Block Volume Results

In this section the memory requirements of the block volume variants are evaluated. In Table 7.2 the memory requirements for the block volume types are shown. The memory usage for the uncompressed block volumes is marginally higher, because the width of a data-set is rounded up to a multiple of the block width.

Data-set	Original	Uncompr.	Shared	Shared & Compressed
Visible Male (CT)	717 MB	770 MB	295 MB	108 MB
Christmas Tree (CT)	499 MB	512 MB	468 MB	238 MB
Visible Male (Photo)	1613 MB	1839 MB	615 MB	327 MB
Santa Claus (CT)	240 MB	256 MB	256 MB	163 MB

Table 7.2: Comparison of memory requirements of the block volume variants for volumetric data.

Data-set	Reduction by Sharing	Reduction by Compr.	Overall Reduction
Visible Male (CT)	61%	63%	86%
Christmas Tree (CT)	9%	49%	54%
Visible Male (Photo)	67%	47%	82%

Table 7.3: Comparison of memory requirement reduction using the shared block volume and the compressed shared block volume.

A significant reduction of the memory usage is achieved with the shared block volume. Data blocks are shared by several blocks reducing the overall memory requirements. Especially data blocks containing homogeneous data are shared among blocks containing identical data. With compression, a further reduction can be achieved. Table 7.3 shows the achieved reduction of three data-sets in percent.

7.3 Ray-Casting on the Block Volume

In this section several results of both strategies applied on the ray-casting algorithm introduced by Bruckner [2] are presented. As depicted in Figure 7.2, the rays are successively advanced in a ray front. The following description of the ray-casting traversal is taken from Bruckner [2]:
The volume is subdivided into blocks. These blocks are then sorted in front-to-back order depending on the current viewing direction. The ordered blocks are placed in a set of block lists in such a way that no ray that intersects

a block contained in a block list can intersect another block from the same block list. Each block holds a list of rays whose current resample position lies within the block. The rays are initially assigned to the block which they first intersect. The blocks are then traversed in front-to-back order by sequentially processing the block lists. The blocks within one block list can be processed in any order, e.g., in parallel. For each block, all rays contained in its list are processed. As soon as a ray leaves a block, it is removed from its ray list and added to the new block's ray list. When the ray list of a block is empty, processing is continued with the next block. Figure 7.2 illustrates this approach.

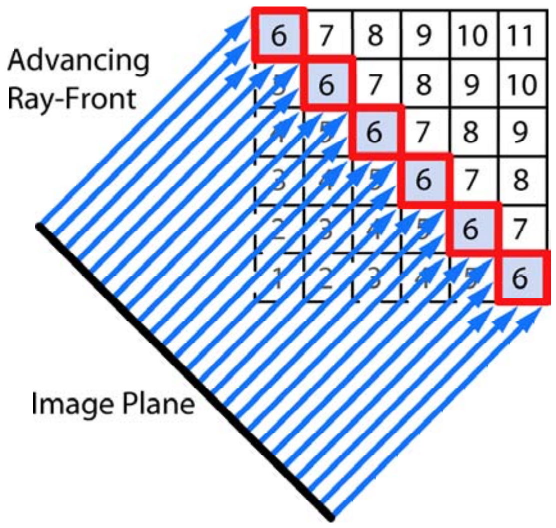


Figure 7.2: Block-wise ray-casting scheme (taken from [2]). A ray-front is advancing through the volume processing one list of blocks in each iteration step. The numbers inside the blocks denote the iteration step.

In Table 7.4 the frame rates of the rendering process of two different datasets, i.e. the *Visible Male* (see Figure 7.7) and *Santa Claus* (see Figure 7.9), are shown.

Data-set	Size	DBA SBV	DBA CSBV	MBV 40MB	MBV 80MB
Visible Male (CT)	717 MB	0.44	0.32	0.09	0.13
Santa Claus (CT)	240 MB	1.13	0.72	0.52	0.52

Table 7.4: Frame rates (frames per second) of variants of the dynamic block allocation (DBA) and mapped block volume (MBV). The DBA is applied on the shared block volume (SBV) and compressed shared block volume (CSBV). For the mapped block volume a sub-volume size of 256^3 is used and a memory limit of 40 and 80 megabytes.

7.3.1 Dynamic Block Allocation

Figure 7.3 shows two successive iteration steps during the ray-casting algorithm. During an iteration step the *block list* contains the relevant data blocks (shaded background) inclusive their neighboring blocks (for example, required for gradient calculation). In total, during 15 iterations the following numbers of blocks will be successively processed: 0, 6, 6, 9, 11, 19, 24, 24, 21, 19, 19, 9, 11, 0, 0. 175 blocks in total. The lists of block between successive iteration steps contain a number of common blocks, thus the common blocks are reused for the next iteration step and only the new blocks are prepared for processing. For example, in Figure 7.3 the number of common blocks is 14, thus only 10 new blocks (instead of 24) are prepared. Figure 7.4 shows numbers of the rendering process of the *Skewed Head* data-set (see Figure 7.10).

7.3.2 Mapped Block Volume

In a compressed block volume each block has a status flag indicating if it is compressed or not. Therefore it is straightforward to incorporate out-of-core support. This has been done for the mapped block volume.

The diagrams in Figure 7.5 show the relation between the *memory usage*, *memory limit* and *time limit* during runtime of the out-of-core volumetric data processing feature for a series of frames. Basically, the *Memory Usage* decreases with a decreasing *Time Limit* and increases with an increasing

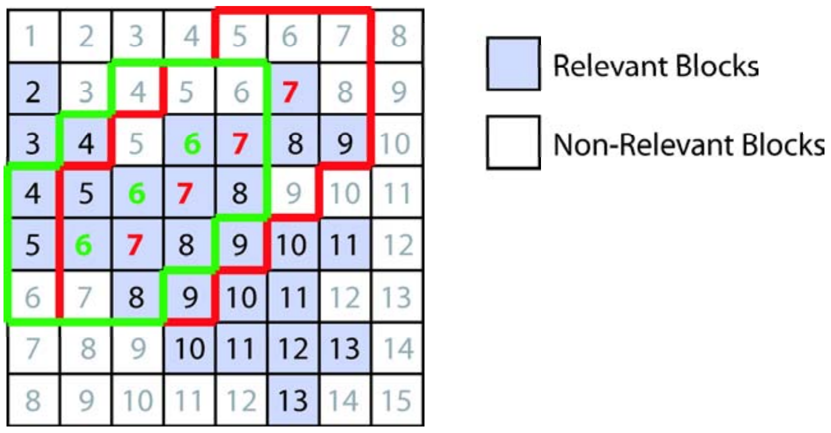


Figure 7.3: The list of blocks of two successive iterations are shown: The blue shaded blocks inclusively their neighbors are the relevant blocks for the ray-casting algorithm, the others are omitted, i.e. they are *non-relevant*. Iteration 6 is colored green and iteration 7 is colored red. In iteration 6 (green) 19 blocks are in the list, in iteration 7 (red) there are 24 blocks. The overlapping area between both iterations contains 14 blocks.

Time Limit.
Memory usage is the total amount of memory allocated for compressed data.
The relation between the three variables is described as follows: The *memory limit* is a fixed variable. When the *memory usage* is greater than the *memory limit* then the *time limit* is decreased, otherwise increased. A lower *time limit* means a shorter life-time for data blocks, causing them to be swapped out earlier and thus the *memory usage* decreases.
The soft *memory limit* makes the algorithm keeping the *memory usage* below this limit, but it is allowed to exceed it when necessary. As long as the total *memory usage* stays below the limit, no swapping out is necessary and the *time limit* is increased by one after each *block memory analysis*. As soon as the total *memory usage* is above the limit, the oldest data blocks, i.e. the data blocks with a difference between the *current time* and the *time counter* greater than the *time limit*, are swapped out and the *time limit* is decreased by one.
A data block is *swapped in* when the volume handler prepares this data

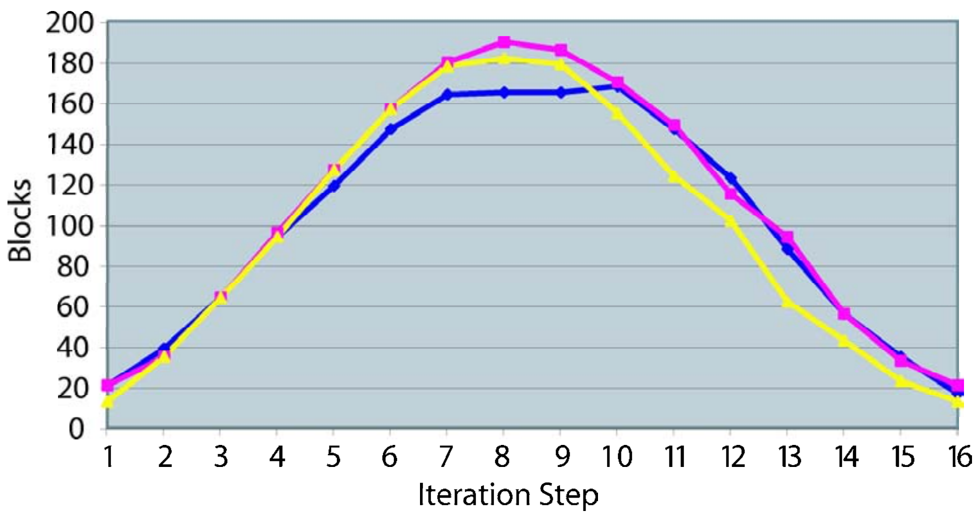
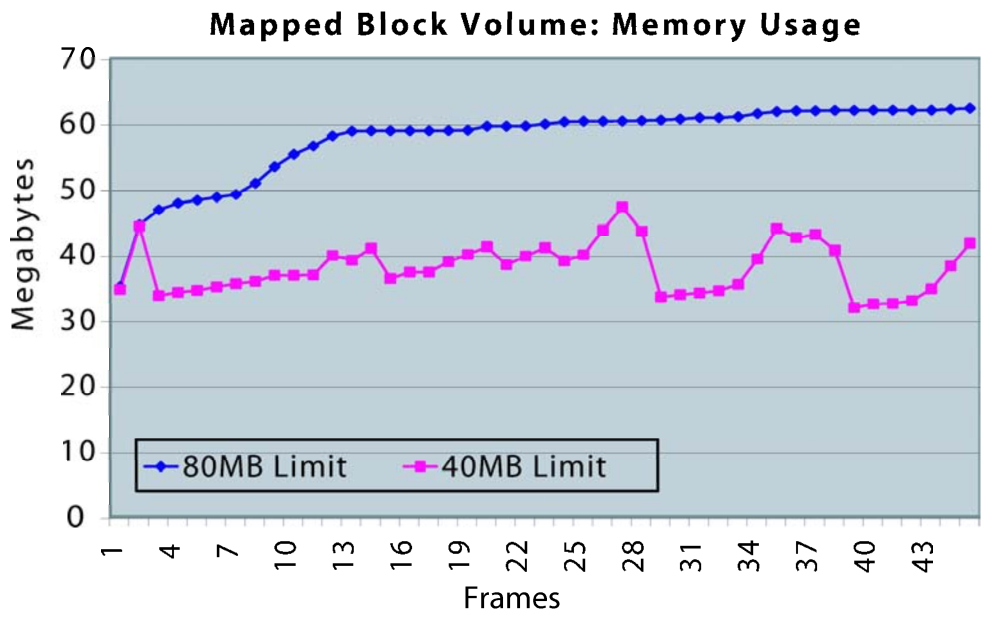


Figure 7.4: Number of used blocks per iteration step of the dynamic block allocation: The results of three renderings (ray-casting) from similar view-points are shown (yellow, blue and purple). The number of blocks differs depending on the view-point. The data-set (*Skewed Head*, see Figure 7.10) has a total count of 288 blocks (6 x 6 x 8 blocks).

block for being accessed by an iterator and the data block is not memory resident. In that case, memory is allocated and the data block is loaded from the mass storage device. After that it is decompressed into its associated *processing memory block*. In Figure 7.6 the numbers of blocks processed for each rendered frame are shown. *Not Used Blocks* is the number of blocks which are not in the main memory at all. *Swapped In Blocks* is the number of blocks loaded from the hard-disk for a specific frame. *Decoded Blocks* is the number of blocks decompressed. *Swapped Out Blocks* is the number of blocks swapped out, i.e. removed from the main memory.



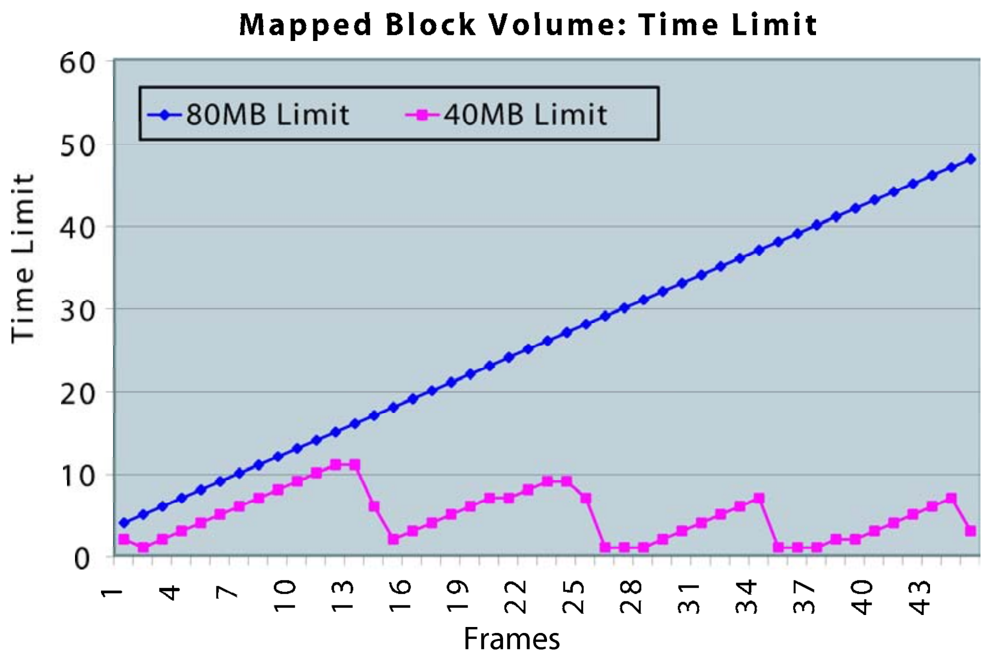


Figure 7.5: Mapped Block Volume. Rendering of the *Visible Male* data-set (716MB) with a memory limit. The memory usage is controlled by the time limit. Top: At the 80 megabyte limit the memory usage grows up-to 63 megabytes, then stays at this level. With the 40 megabyte limit the memory usage oscillates around the limit. Bottom: At the 80 megabyte limit as long the memory limit is not reached, the time limit is increased, whereas at the 40 megabyte limit when the memory usage exceeds the limit the counter is decreased.

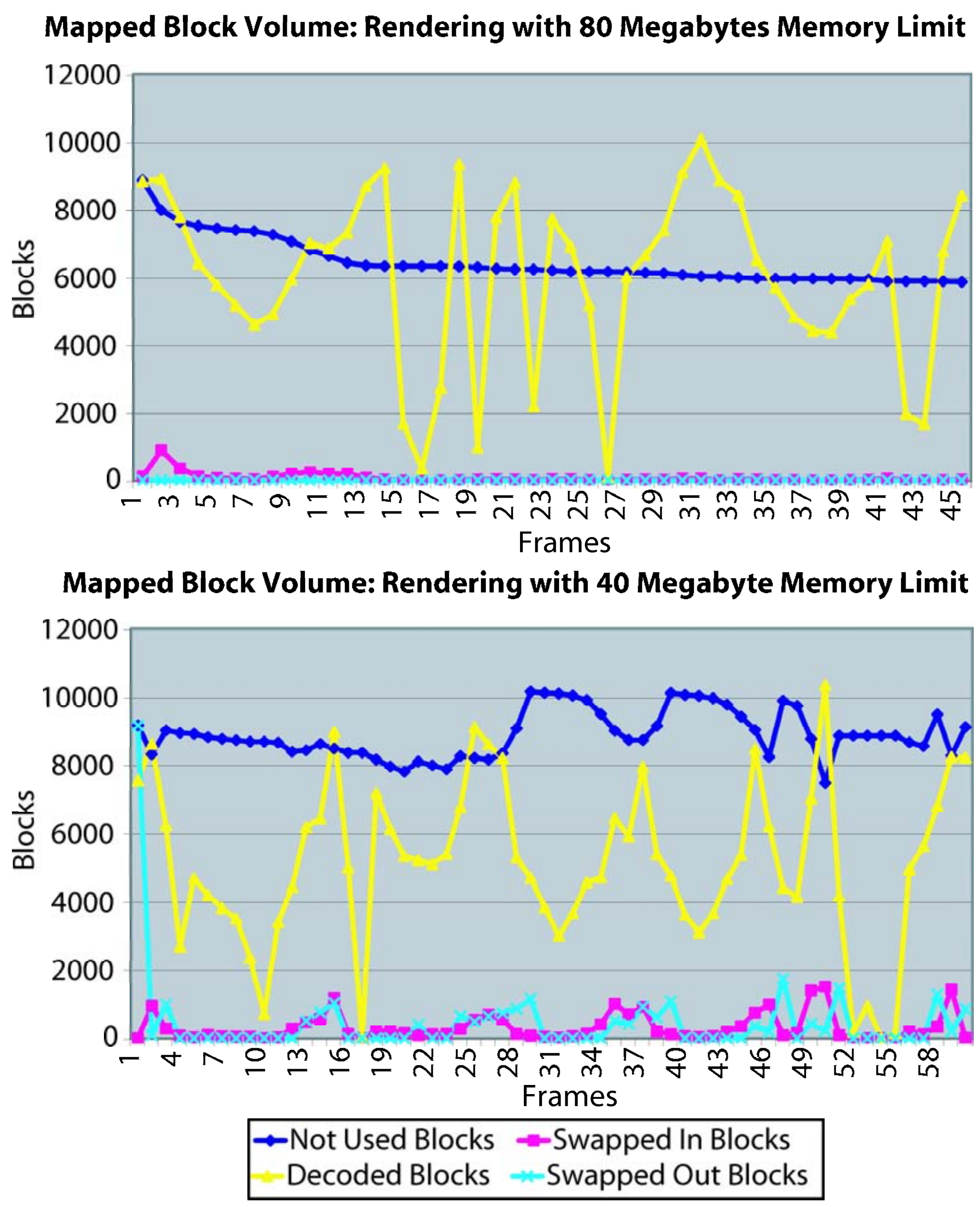


Figure 7.6: Mapped Block Volume. Rendering of the *Visible Male* data-set (716MB) with a fixed soft memory limit. Top: Rendering with a memory limit of 80 megabytes. Since enough memory is available (for rendering only 62 megabytes are required) the memory usage stays below the limit, thus no swapping out of blocks occurs. Bottom: Rendering with a memory limit of 40 megabytes. Since the memory limit is exceeded, blocks are swapped out and swapped in again later.

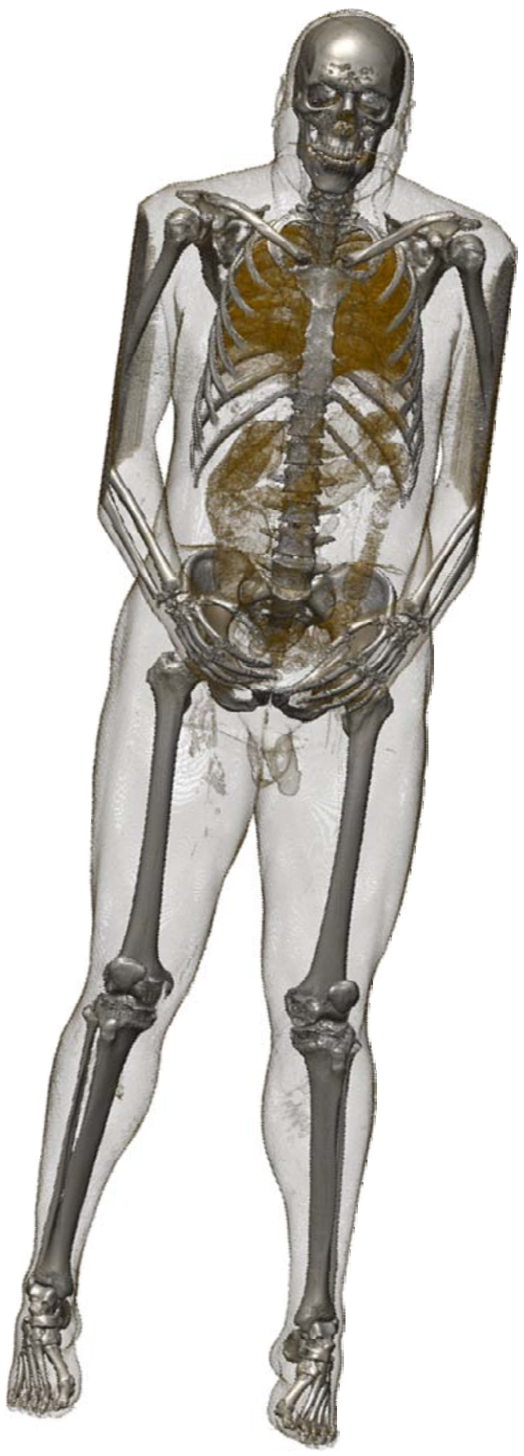


Figure 7.7: An image of the *Visible Male* data-set (587 x 341 x 1878 voxel).

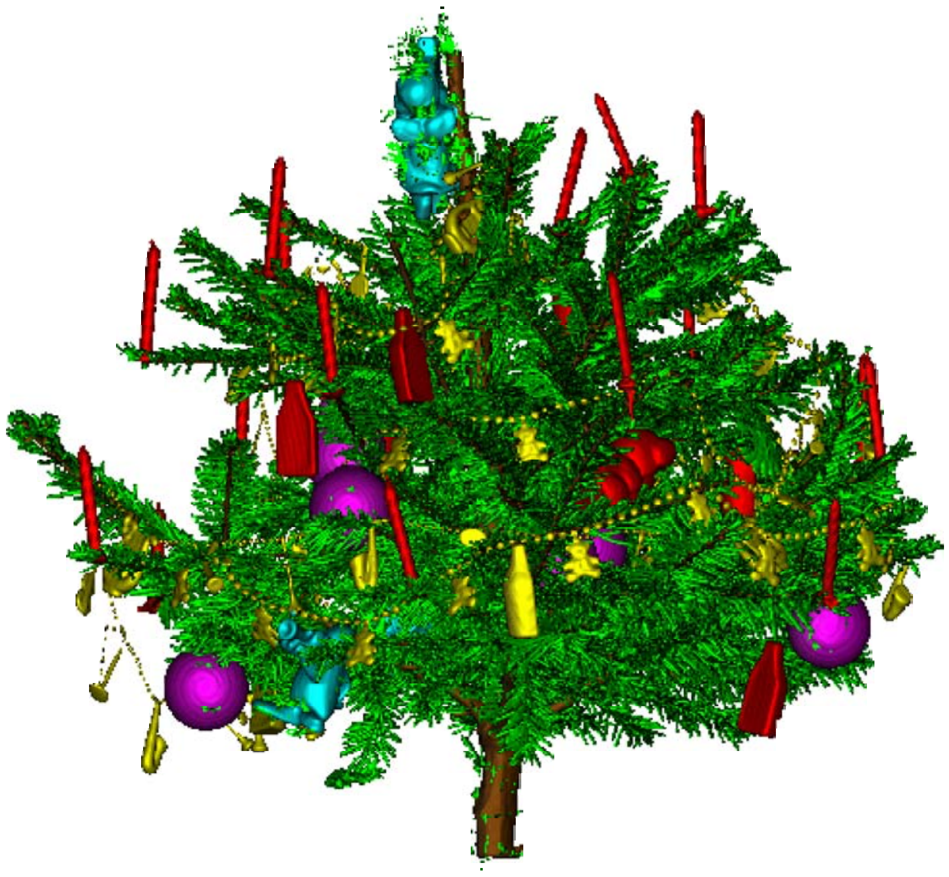


Figure 7.8: An image of the *Christmas Tree* data-set (512 x 512 x 999 voxel).

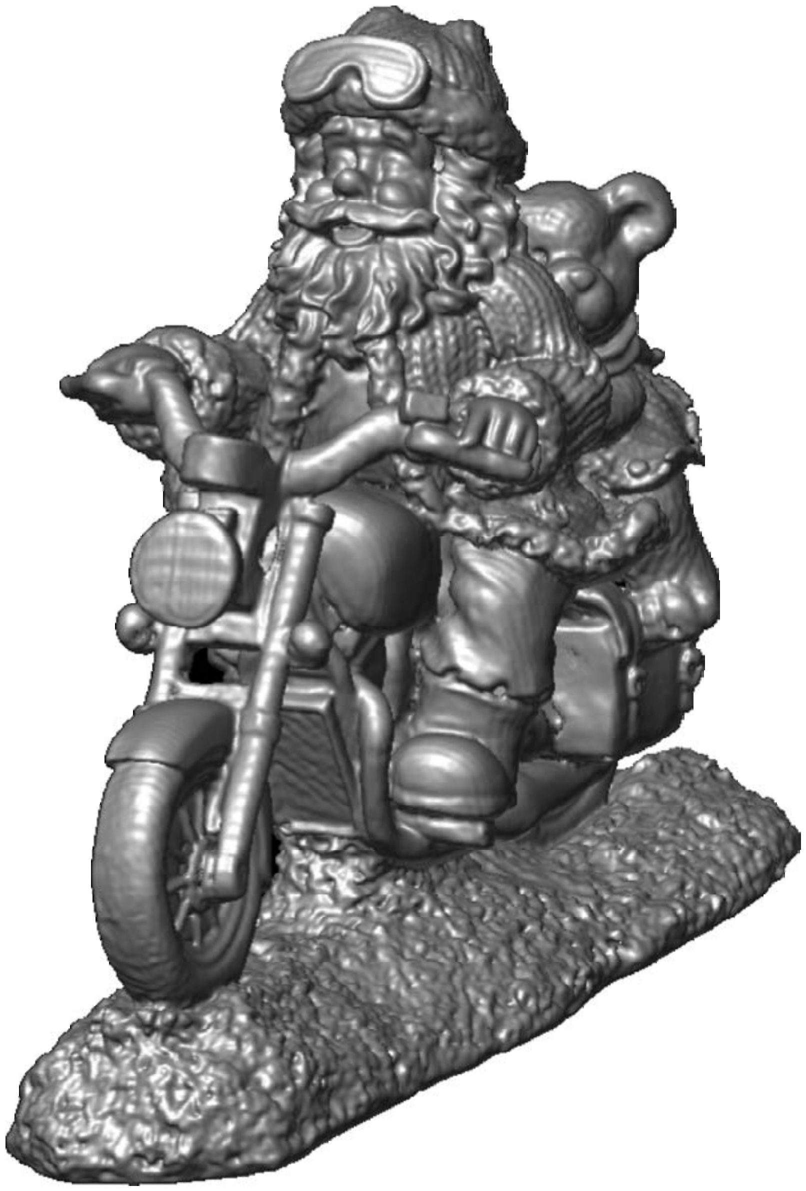


Figure 7.9: An image of the *Santa Claus* data-set (512 x 512 x 481 voxel).

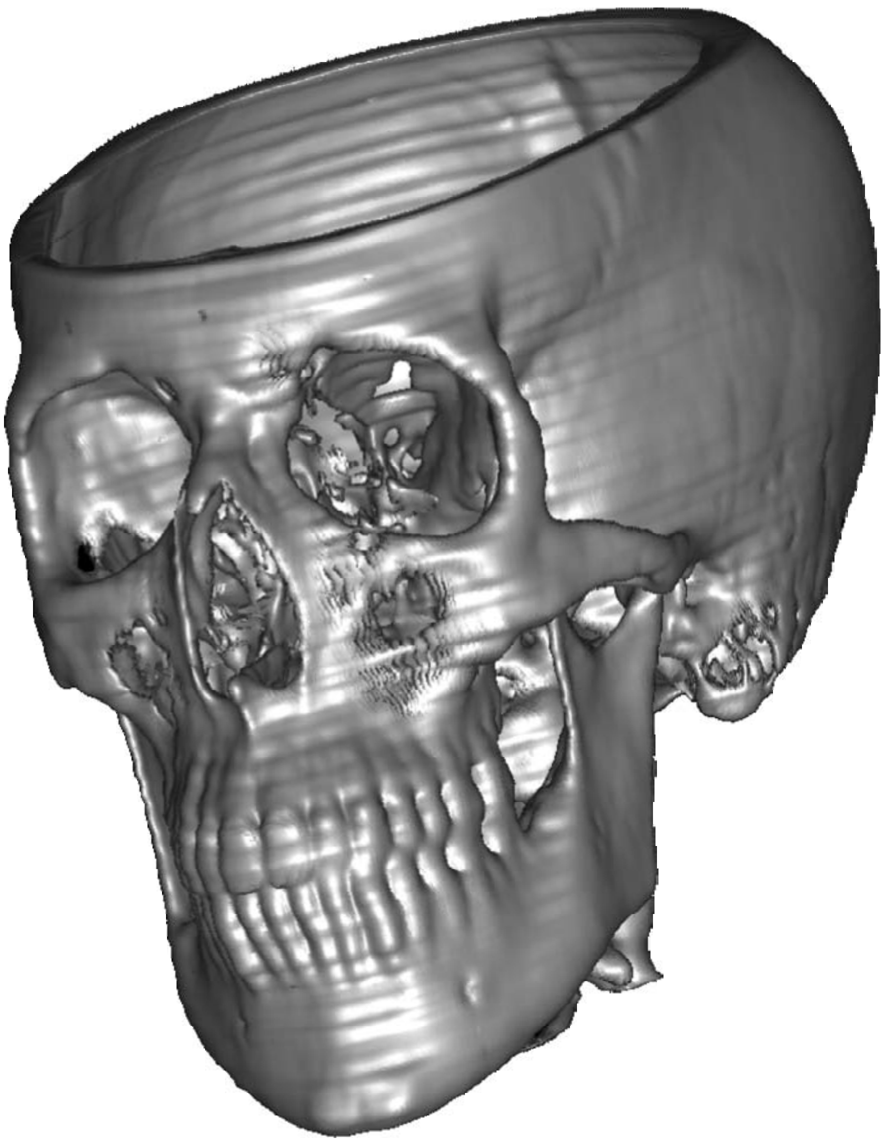


Figure 7.10: An image of the *Skewed Head* data-set (184 x 256 x 170 voxel).

Chapter 8

Summary

8.1 Introduction

Since the introduction of the computed tomography scanner by G.N. Hounsfield in the early 1970s processing of volumetric data tremendously gained in importance. Computed tomography is the reconstruction of cross-sectional images from multiple one dimensional scans along lines from different angles on a plane. It is also referred to as computed tomography. The reconstruction of a series of such cross-sectional dimensional images results in a three dimensional, i.e. volumetric, data-set.
In 1972 the first clinical diagnostics based on computed tomography images were made. A few years later several thousand computed tomography scanners were in use worldwide. The first scanners had a slice size of 80x80 pixel and a thickness of 13 millimeter.
In the early days of computed tomography, it was sufficient to examine a single slice, in order to conduct a medical diagnosis. At this time, scanning a single slice took 300 seconds. Later, with the reduction of the scanning time down to a few seconds, the acquisition of series of slices at once became common.
With the up-come of personal workstations in the 1980s, digital processing of the computed tomography data became more and more popular. Over the years, the computed tomography technology improved. Now, the slice

resolution of modern scanners is at 1024x1024 with a slice thickness of 0.5 millimeter. A scan usually results in a series of several hundred slices.
This is the point, where volumetric data processing becomes indispensable: Examining hundreds of slices one by one is not feasible anymore. The introduction of an additional processing step between the data acquisition and the visualization for diagnostic purpose is necessary. This processing step is referred to as *volumetric data processing.*
The past years have shown, that the size of volumetric data-sets has increased as fast as the CPU processing power and memory size. Thus efficient data handling has ever been an important issue of volumetric data processing. Today the common silicon technology reached its physical limits, whereas in the computed tomography area there is still much potential for further increase of the data quantity: the recent introduction of 16-slice scanners, better detectors and the emergence of multi-modal data let expect a further increase of the amount of data to be processed.
Common computer architecture is sufficient for average applications, however its complex architecture cannot serve optimally applications with exceptional resource requirements. Such exceptional requirements are needed in the field of volumetric data processing, where very efficient data transfer capabilities in all components, i.e. CPU, memory and mass storage devices, are required. For optimally utilizing the data transfer capabilities of these components, the development of specific algorithms exploiting the architectural advantages of the common computer hardware is desirable.

8.1.1 Data Processing Issues

Memory Access The hardware components involved in the data processing are optimized for sequential contiguous access. A sequential access is simply an access to a contiguous block of data with specific starting address and length. In volumetric data processing fast access is very important in order to achieve high performance. Most memory hardware support sequential contiguous accesses by so-called *burst transfers.* After the starting address, and optionally, if supported by the architecture, the length of the desired

block, has been sent to the memory controller, a whole sequence of words is transferred at once instead of a single word per addressing action. A word is the smallest transferable element. Thus it is important to exploit this sequential access behavior when accessing the main-memory or any mass storage device.
Irregular accesses constantly trigger cache updates. This leads to a degradation of performance if memory hierarchy levels are affected which are sensitive to non-sequential accesses. The affected levels are usually the main-memory and the mass storage devices. As long as no update of the caches through those levels is constantly required, no significant performance decrease occurs.
If an algorithm accesses a memory or mass storage device in an irregular pattern or, more simply speaking, not in a sequential contiguous fashion, the performance drastically degrades. Methods overcoming this issue by exploiting the advantages of the current computer architecture hierarchy are presented. The intention of these methods is to allow irregular accesses while maintaining sequential accesses for performance reasons. Before the block volume is introduced, the sequential layout, a commonly used storage format for volumetric data is discussed.

8.2 Sequential Layout

The sequential layout has arisen in the 1970s with the upcoming of medical three dimensional imaging systems like computed tomography devices. The data values of the volumetric data-set are sequentially stored starting with the first line of the first slice going to the last line of the last slice. Basically the volume data-sets acquired from computed tomography are organized according to this layout.
This layout allows an effective sequential line-by-line respective slice-by-slice processing of the data. Therefore it is most suited for algorithms which process data in a line-by-line respective slice-by-slice manner.
The most profoundly disadvantage of this layout is, that the effectiveness decreases dramatically when algorithms require random access to the data,

or, more simply, require access in a different way than in a slice-by-slice or line-by-line manner. Even accessing slices in the xz-plane results in processing of single lines across the entire data-set. Even worse, accesses to slices in the yz-plane results in processing of single voxels, which totally compromises the caching mechanisms in the hardware optimized for sequential accesses.

8.3 Contiguous Block Volume

The basic data structure for the block allocation strategies is a *block volume*. A block volume is a two level application of the sequential layout. The volume data-set is organized in sequentially organized fixed-size blocks, which are internally organized again according to the sequential layout. The block volume demands a more complex addressing scheme compared to a simple sequential volume. The size of the data blocks is selected in away so that it fits into the processor's cache. In a contiguous block volume the blocks are sequentially stored in a contiguous memory block.

8.3.1 Absolute Addressing

The block address and further the voxel offset are derived from the position in the three dimensional space as described: Since the width of a block is an integer power of two, simple arithmetic, masking and shifting operations can be used for calculating the block and voxel offset. In the first step the block index and the voxel address relative to the block are calculated from its position. In the second step the voxel address in memory is calculated by multiplying the block index with the block size.

8.3.2 Relative Addressing Scheme

Once the absolute address for accessing a specific voxel is calculated, it is advantageous to have an efficient method for accessing neighboring voxels. The specific voxel is called the *active voxel* respectively the 3x3x3 cube, where this voxel is located in the center, is called the *active cube*.

For each voxel in a block there are 26 different neighbors. In this approach, the absolute addresses of the neighbors are calculated by adding an offset to the address of the active voxel. This offset is called the *relative address.* The relative address is the difference between the absolute address of the active voxel and one of its 26 neighbors. The neighbor-lists are classified into 27 different cases. In each case the 26 relative addresses are identical for each voxel within the case. Ultimately the neighbor look-up table consists of 27 lists with 26 entries each, in total 702 entries.
These 27 cases are derived from the voxel position within a block. A case is denoted by an *case index.* The case index is derived from the position within the block in the following way: For the one dimensional space there are only 3 cases: most left voxel, inner voxels and most right voxel. For the inner voxels neighbor addressing is straightforward: each neighbor voxel is located in the same block as the active voxel. But for the left voxel, its left neighbor is located in a different block, i.e. the left neighboring block of the active block. This case is analogous for the most right voxel: The right neighbor voxel of this voxel is located in the right neighbor block.

$$CaseIndex_{dim} = ((((((Position_{dim}\&31) - 1)\&63)|1) + 1) >> 5) \tag{8.1}$$

$Position_{dim}$	$CaseIndex_{dim}$
0	2
1..30	0
31	1

The formula above depicts the *case index function.* In the calculation only simple arithmetic, bit-manipulation and shifting operations are used, thus the calculation is very fast. All possible function values are shown in the table below the formula.
The case index for three dimensional space is calculated as follows: Since the range of function values of the case index function is limited to 0..2, the three dimensional case index is simply derived as depicted in the following formula.

$$CaseIndex = CaseIndex_z * 9 + CaseIndex_y * 3 + CaseIndex_x \tag{8.2}$$

Since for each block the relative addresses are identical, only a single neighbor look-up table is required for the entire block-volume. Additional border blocks are added at to ensure a consistent relative addressing also at the border voxels of the block volume.

8.4 Fragmented Block Volume

In a fragmented block volume, the blocks are irregularly distributed in the memory space. Therefore the second step of the absolute address calculation has to be modified as follows: The absolute address of a block is looked up in the *block address list* instead of deriving it directly from the block index. Since for each block the neighboring blocks have different relative addresses, an own neighbor look-up table for each block is required.

8.5 Shared Block Volume

Basically a shared block volume is a fragmented block volume. The shared block volume is a more sophisticated structure, which effectively tries to reduce memory requirements by sharing blocks.
In this concept a block and its data block are separated from each other. The idea behind the shared blocks is, that at creation time the block volume is "empty". At the beginning all voxels have the same density value, for example 0 (zero), therefore each data block in the volume contains the same values. Therefore it is possible, that initially one data block is shared by all blocks. The shared block technique reduces the amount of required memory significantly for volumetric data-sets with large homogeneous regions.

Data Block Split

If a new value is written to a data block, which differs from the value at the current location, the data block is replicated and the new value is written to the replica. First the data block is shared by n blocks, after this step the data block is shared by $n - 1$ blocks, and the replicated and modified data block is shared by only one block.

Data Block Merge

If a set of m blocks are sharing a data block and a set of n blocks are sharing another data block, both sets are disjunct, but the data blocks contain identical data, thus both sets can be merged. One data block can be disposed and the other data block is shared then by $n + m$ blocks.

8.5.1 Border Issue

Since the data blocks are separated from the blocks, it is not necessary to add additional border data blocks. A special data block is provided for serving as border data block required for updating the neighbor look-up tables. Instead of adding extra border data blocks like in the contiguous or fragmented block volume, these border data blocks are replaced by this special border data block.

8.6 Compression and Out-of-core

Until now, only data structures storing uncompressed data have been presented. In these data structures the data is completely memory resident and instantly accessible. At the point, where the memory usage of the volume data exceeds the available memory, the organization of the volume data according to the sequential layout or blocked layout is not sufficient anymore. Introducing compression is a solution to this memory availability problem.
A new issue comes up with the use of compression: Memory for storing the compressed *and* uncompressed data is required concurrently. The memory

requirements of the compressed data is proportional with the size of the data-set, whereas the memory requirements for the uncompressed data depends on the way of processing the data.
In the implementation for compressing the data blocks a LZ77 descendent compression algorithm is used. It is the so-called LZO [11], the Lempel-Ziv-Oberhumer compression algorithm. The compression and decompression is slightly faster than of other LZ77 variants. Also implementing out-of-core methods combined with compression is straightforward.

8.7 Dynamic Block Allocation

In this allocation strategy, the shared block volume model is used for storing the volume data. For allocating memory to the blocks, a list of all *data blocks* which are currently prepared for being accessed and pointer to a valid *processing memory block* is maintained. A data block can be in two exclusive states:
1. *Frozen.* In this state, the data block exists only in compressed form and the pointer to the *processing memory block* is null;
2. *Defrosted.* In this state the data block is uncompressed and stored in a *processing memory block.* For managing the access to the data a *volume iterator* is used. It maintains a list of its currently used data blocks. Before the blocks can be accessed, the volume iterator triggers the volume handler to prepare the required data blocks for being accessed.

8.7.1 Operation

The *volume iterator* operates in the following way: After initialization of the iterator, the processing loop is entered. In the loop groups of blocks are successively processed in iteration steps. The memory for processing is allocated as follows: First a list of all blocks to be processed is generated. After that all blocks, which have been processed in the previous iteration step are removed from this list, because they are still in *defrosted* state. Then, memory to the remaining blocks is allocated as follows: In the list of all *defrosted* blocks a block is searched, which was not used for a specified time interval. If such a block is found, it is *frozen*, removed from the list and its memory area is assigned to the new block, which is added to the list. If no block has been found a new memory area is allocated.

8.7.2 Discussion

Since for each block a neighbor look-up table for the relative neighbor addressing has to be stored, the space required for this tables increases with the size of the data-set. The space required for the neighbor look-up tables will be significantly large, making this strategy inoperative.
For example a 2048^3 volumetric data-set consists of $64^3 = 262144$ blocks. The neighbor look-up table has usually the size of 2.8 kilobytes. This would result in a memory requirement of 716 megabytes for the neighbor look-up tables only. This is not feasible for really large data-sets. Since this issue cannot be solved easily, a complete new method was developed which circumvents this problem.

8.8 Mapped Block Volume

In this approach, the sequential layout is applied in a three level stage. The block volume consists of a number of fixed sized *data sub-volumes* holding the *data blocks* and one single *processing sub-volume* holding the *processing blocks*. Since the size of a sub-volume is fixed, therefore the whole block volume consists of a fixed number of blocks. The sub-volumes are cubes with a width of 2^n blocks, where n is a fixed number. Also, different from the *Dynamic Block Allocation* strategy, the size of the overall block volume is fixed regardless of the size of the volumetric data-set. Instead of having a dynamic allocation of *processing blocks* for compressed data, the allocation is fixed and held by the *processing sub-volume*. The data blocks at a specific position in each *data sub-volume* share a single *processing block* of the *processing sub-volume*.
The *processing sub-volume* is a small block volume organized like a frag-

mented block volume. It holds the *processing blocks* and the neighbor look-up tables for the relative addressing scheme. Each *processing block* has a list to its data blocks. At the border the address wraps around to simulate a large, theoretically infinite volume data-set.
Addressing the data in this layout is marginally more complicated than for a simple block volume. The sub-volume to load the data from can be easily derived from the voxel position.

8.8.1 Operation of the Mapped Block Volume

This access strategy is extended with a simple out-of-core data handling feature. This out-of-core approach operates only on the compressed block data. For each data block a state describing whether the compressed block data is in memory or not is maintained. So the volume handler knows when to load a block from the mass storage device. For each data block the time when it was accessed the last time is stored. After a *volume iterator* has ended its operation, the set of all memory resident data blocks is analyzed for blocks, where the difference between the current time and their last access time exceeds a specific time limit. If the limit was exceeded, the block is *swapped out*: If it is not yet stored on the mass storage device, it is written to it, then its memory is deallocated.
The memory usage is controlled by the memory limit. This limit is a soft limit which means that the algorithm tries to keep the memory usage below this limit, but it is allowed to exceed the limit when necessary. As long as the total memory usage stays below the limit, no swapping out is necessary and the time limit can be increased. But, if the total memory usage is above the limit, the blocks above the specified time limit are swapped out and the time limit is decreased.

8.8.2 Discussion of the Mapped Block Volume

The advantage of this strategy is, that the resources required for processing have a fixed size. Also the size of the data-set has no impact on the size of the resources required for processing. For multiprocessing purposes several processing sub-volumes can operate on the same volumetric data-set.

8.9 Conclusion

An effective memory layout for volumetric data, the *shared block volume* and two memory allocation strategies have been introduced. The first strategy, the *Dynamic Block Allocation* is based on the shared block volume. The memory requirement is controlled by allocating memory only to the processed blocks. The second strategy, the *Mapped Block Volume* has a fixed memory usage for block processing, but the memory usage is controlled by an out-of-core handling of the compressed data.
With the use of compression in combination with out-of-core processing large data-sets, several times larger than the available physical memory, can be processed.

Acknowledgements

First of all, I want to thank my supervisor Sören Grimm for all his advise and help. I further thank Stefan Bruckner for helping me with several questions related to the Raybooster framework. Again I want to thank both for many nice colorful images. Finally I want to thank our Beloved Master, before His enlightenment known as Eduard Gröller, for his supervision and support.

Bibliography

[1] Ch. Bajaj, I. Ihm, S. Park, and D. Song. Compression-based ray casting of very large volume data in distributed environments. In *Proceedings of Pacific Graphics 2001*, pages 212–222, 2001.

[2] S. Bruckner. Efficient volume visualization of large medical datasets. In *Diploma Thesis*, 2004.

[3] Y. Chiang and C.T. Silva. I/O optimal isosurface extraction. In *Proceedings of Visualization*, pages 293–ff. IEEE Computer Society Press, 1997.

[4] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. In *IEEE Transactions on Visualization and Computer Graphics*, pages 158–170, 1997.

[5] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of Visualization*, pages 235–244, 1997.

[6] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computer and Graphics*, 28:719–729, 2004.

[7] S. Guthe and W. Strasser. Real-time decompression and visualization of animated volume data. In *Proceedings of Visualization 2001*, pages 349–356, 2001.

[8] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. In *Proceedings of Visualization*, 2002.

[9] G. Knittel. The ultravis system. In *IEEE Volume Visualization and Graphics Symposium*, pages 71–79, 2000.

[10] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface construction algorithm. In *Proceedings of SIGGRAPH*, pages 163–169, 1987.

[11] M.F.X.J. Oberhumer. *LZO lossless compression algorithm.* http://www.oberhumer.com/ (December 2004), 1994.

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

[13] A. Said and W.A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6:243–250, 1996.

[14] D. Saupe and J. Toelke. Optimal memory constrained isosurface extraction. Technical report, Department of Computer Science, Universität Leipzig, 2001.

[15] J.M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. In *IEEE Transactions on Signal Processing*, pages 3445–3462, 1993.

[16] T. Welch. A technique for high-performance data compression. In *Computer*, 1984.

[17] Ch. Yang and T. Chiueh. I/O-conscious volume rendering. In *Proceedings of IEEE TCVG Symposium on Visualization*, pages 263–272, 2001.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. In *IEEE Transactions on Information Theory*, 1977.

List of Figures

List of Tables