TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

D I P L O M A R B E I T

An Advanced Data Structure for Large Medical Datasets

ausgeführt am

Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter Anleitung von

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

durch

Alexander Hartmann
Dreihäusergasse 12a
2345 Brunn/Geb.

10. Mai 2005 _____

Datum

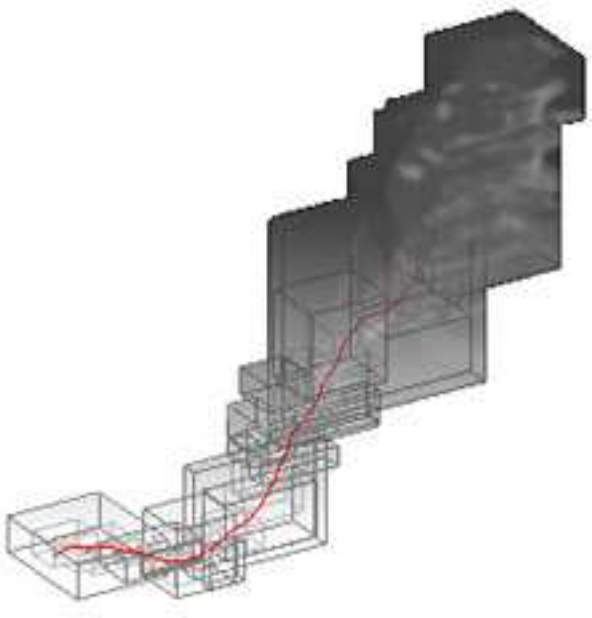
_____

Unterschrift

Alexander Hartmann

An Advanced Data Structure for Large Medical Datasets

Master's Thesis



supervised by

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

Dipl.-Inform. Dr.techn. Sören Grimm

Prof. Dr. Kenneth I. Joy

Institute of Computergraphics and Algorithms

Vienna University of Technology

Abstract

The size of volumetric data acquired from computed tomography scanning devices is steadily increasing, which often makes it impractical to store the whole data in physical memory. Therefore, efficient data structures are required. In this thesis several data structures are examined in respect to application for computed tomography-angiography. In particular, memory consumption and performance of visualization are addressed. Additionally, a data structure based on adaptive meshes is implemented. This data structure can leverage resources where they are needed. In order to generate the adaptive meshes, two different algorithms are explained and compared to each other. The most common visualization techniques for angiography are described.

Kurzfassung

Die Größe der Volumensdatensätze in der Computertomographie nimmt stetig zu, weshalb diese oft nicht mehr komplett im physischen Speicher gehalten werden können. Deshalb ist es notwendig, effiziente Datenstrukturen zu verwenden. In dieser Arbeit werden diverse Datenstrukturen auf die Eignung bezüglich Computertomographie-Angiographie Datensätzen untersucht. Im Besonderen wird auf Speicherplatzverbrauch sowie das Leistungsverhalten der Visualisierung eingegangen. Zusätzlich wird eine Datenstruktur, die auf adaptiven Gittern basiert, implementiert. Diese Datenstruktur ermöglicht einen wirksamen Einsatz der Ressourcen. Dabei werden zwei unterschiedliche Algorithmen zur Erzeugung der adaptiven Gitter beschrieben und miteinander verglichen. Die gebräuchlichsten Visualisierungsverfahren der Angiographie werden erklärt.

Contents

Chapter 1

Introduction

God heals and the Doctor takes the fee.

Benjamin Franklin

The handling of very large datasets is a very important topic in volume visualization. Especially in medical image processing Computed Tomography (CT) scanning devices can create huge datasets. A CT scanner is a device which generates a series of two-dimensional X-ray images which show the internals of an object. These X-ray images are generated using an X-ray source that rotates around the object. Several scans are progressively taken while the object is passed through the scanning device. New technology multi-slice CT scanners produce even higher resolution and more slices than conventional scanners.

Arterial diseases are a major health problem in the industrial countries, therefore a lot of research is focused on new screening techniques. One method is catheter angiography, where a catheter is placed into the persons body in order to inject a contrast agent close to the area of interest. This contrast agent highlights the vessels when X-rays are taken. Another less invasive method is Computed Tomography Angiography (CTA), here the contrast agent is injected into a vein rather than an artery. Then a CT scan is taken in order to visualize the blood flow in arterial vessels throughout the

body.

Angiography is a medical imaging technique that uses X-rays to visualize blood filled structures, such as arteries, veins, and heart chambers. Because blood has the same radiodensity as the surrounding tissues, a radiocontrast agent (which absorbs X-rays) is injected to highlight vessels, enabling angiography. Angiography is commonly performed to identify vessel narrowing and calcifications.

To investigate the arteries a sequence of 300 - 1500 (depending on the region of interest) CT images are necessary. The acquisition of these CT images usually takes 30 to 60 seconds. Due to the large amount of slices, 2D examination is a rather tedious and time consuming task. Therefore, radiologists typically use the following two visualization techniques: Maximum Intensity Projection (see Chapter 2.4.6) and Curved Planar Reformation (see Chapter 2.4.7). With these visualization methods the investigation of the arteries usually only takes a few minutes.

The large amount of data, needed for CTA, presents a challenge for current PC hardware. Therefore, it is very important to design memory efficient data structures in order to handle these large datasets. These data structures should allow to leverage resources where they are really needed. For example, in case of CTA only the aorta is of main interest. The surrounding information is only needed as context information or not needed at all. This part of the data, therefore, needs to be stored in a lower resolution or not at all. Such a data structure, which allows to efficiently leverage resources, enables that the data can easily be kept in main memory. Thus, efficient processing of the data is possible.

We based our data structure on an adaptive mesh refinement approach. This allows to store important parts of the data with high resolution and less important data with lower resolution. Thereby large datasets can be completely kept in the physical main memory.

In Chapter 2 the data processing pipeline is described. Starting with the data acquisition, then a description of the CT data itself, explaining the segmentation process and finally showing the different visualization possibilities.

Chapter 3 shows the most common data structures and how they apply

to volumetric data where only certain areas of the dataset are needed. The memory consumption, the visualization, and the resampling problem are discussed.

Chapter 4 introduces our algorithm to create adaptive meshes from a CTA dataset based on the centerline of a vessel. These meshes are created in different resolutions according to an importance classification of the data. We developed two different algorithms to generate the meshes which are described and compared. Resampling is described with respect to the used visualization techniques.

In Chapter 5 an overview of the implementation with all classes is shown. The classes are briefly introduced and the most important methods are described in more detail.

Chapter 6 presents the results of the implemented algorithm with a CTA dataset and analyzes the two different mesh generation algorithms, as well as the image quality of the images rendered with our data structure.

Chapter 7 contains a summary of the thesis with its focal point to the adaptive meshes in comparison to the common data structures. Chapter 8 is an acknowledgement of all people who helped me get this work done.

Chapter 2

Data processing

It is a capital mistake to theorize before one has data.

Doyle, Sir Arthur Conan

2.1 Data Acquisition

The data is acquired with a Computed Tomography (CT) scanner. CT employs filtered back-projection to reconstruct a volumetric dataset from the measured X-ray projection.

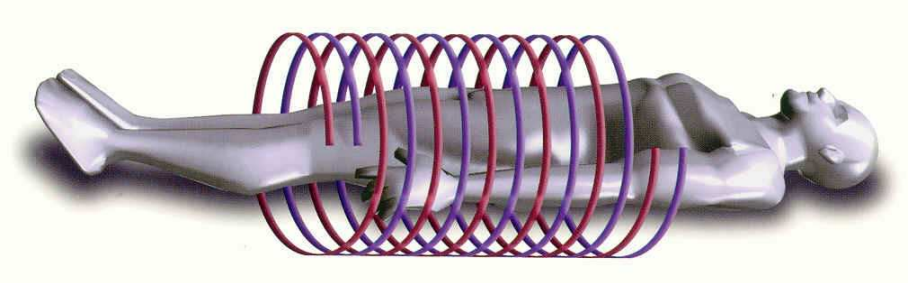


Figure 2.1: Multi-slice CT scanner: X-ray tube rotation to simultaneously acquire several slices [1].

According to the number of detector element rows available, helical CT scanners are distinguished into: single section (single slice, single detector row), dual section (dual slice, dual detector row) or multi-section (multi-slice, multi-detector, multi-row). Multi-slice CT scanners (see Figure 2.1) can produce very high resolution scans.

Multislice CT [27] refers to the ability of a CT scanner to acquire more than one slice simultaneously. The detector system of a Multislice CT is composed of more than a single row of detector elements. The advantages of Multi-slice CT scanners are characterized by

- Resolution: improved spatial resolution along the z-axis
- Volume: increased length that can be scanned for a given set of scan parameters
- Speed: reduced time for scanning a body region
- Power: improved usage of X-ray tube power

Dual slice CT scanners could only be improved in one area (Resolution, Volume, Speed or Power), while scanners with 16 and more slices are virtually unlimited. This allows a lot of new application fields, such as CTA.

An alternative method to spiral or helical scanning is the step and shoot technique. Here, a complete slice of the object is acquired while the object has to be stationary. Afterwards the object is moved in the axial direction to acquire the next slice. This process is repeated until the whole object is scanned. This is a very time consuming task. The step and shoot technique is also vulnerable to misregistration among slices, because separate breath-holds of a person can lead to different images.

In helical scanning the data is acquired much more quickly because there are no breaks during the recording phase. This leads to a better quality of the volume because the images are recorded in a more consistent state. Helical scanning is of significant benefit for CTA.

Current state of the art CT scanners produce slices with a resolution of 1024 by 1024 and usually generate 300 to 1500 slices (depending on the

region of interest). Since each data element needs 2 bytes (see Chapter 2.2) one slice needs 2 MB of memory. Thus, a dataset of 1500 slices produces a total of 3 GB memory requirement.

2.2 CT-Data

A complete CT dataset consists of a series of 2D images, which are referred to as slices. Each slice usually has a resolution of 1024 squared and is recorded in the z (or transversal) direction. Each sampling element (pixel) of the slice is the measured density of the tissue and is stored in Hounsfield Units (HU, see Table 2.1), which is a generalized scale for CT data. The HU range from -1000 to 3095, thus 12 bits are needed for every pixel. Because data in memory has to be byte-aligned, each pixel-value is stored in 2 bytes. A data element inside the dataset is referred to as a voxel (volume element).

Tissue	HU-Range
Air	-1000
Water	0
Fat	70 - 120
Soft tissue	15 - 80
Bone	> 1000

Table 2.1: Houndsfield-Units for different tissue types

The CT data has low noise, high spatial resolution and consistent values (stored as Hounsfield Units).

2.3 Segmentation

Segmentation is the process of isolating objects of interest from the rest of the scene [8], but sometimes it is also defined as the process of partitioning an image into non-intersecting regions such that each region is homogeneous and the union of no two adjacent regions is homogeneous [32].

Segmentation techniques can be distinguished into image- or knowledge-based and automatic or interactive. The image-based algorithms analyze the

image properties such as discontinuity (i.e., boundary detection, edge linking) and similarity (i.e., thresholding, region-growing), while the knowledge-based algorithms use either algorithmic information encoding like homogeneity, density range or distance (e.g., from the skull surface) or rule based systems (specified conditions). Automatic systems are processing numerous datasets with specific tasks (e.g., extract brain from MRI data). They need special parameter settings and often a visual verification is necessary. Interactive or semi-interactive systems are based on an operator's knowledge and experience, provide high precision but are laborious. 2D (slice) and 3D approaches exist.

In order to keep only necessary regions of the dataset, the important regions must first be detected by a segmentation process. This can be a difficult task sometimes (depending on the object to detect) because of non-uniform mapping, inhomogeneities, spurious artifacts and noise. Thus for some objects special algorithms and/or user interaction are necessary [40].

For detection of the centerline of a blood vessel several specialized algorithms exist [21]. These algorithms have different characteristics concerning reliability, speed and accuracy. A global optimization method for a reliable vessel-tracking was introduced by A. Kanitsar [19].

2.4 Visualization of Volume Data

2.4.1 Object/image based

Recently, two classes of volume rendering techniques have been receiving much research attention. Projection techniques [11, 42, 43] are object-order algorithms, thereby a voxel (volume element) is processed at a time. Each voxel is projected onto the image plane and scan converted to determine its contribution to each pixel that the projection covers. These techniques require a depth order of the voxels in the volume and render them from front-to-back or back-to-front. Ray-casting techniques [25, 36, 42] are image-order algorithms, determining the entire volume's contribution to a given pixel before processing the next pixel in the image plane. A volume's contribution

to a pixel is determined by casting a ray through the volume space according to a given view direction. As a ray traverses the volume, it is sampled at regularly spaced intervals. At each sample position the underlying 3D density function is reconstructed and mapped to a color and opacity. These colors and opacities are then composited along the ray to determine the final color for the ray and therefore the pixel.

2.4.2 Reconstruction Filters

To render an image of a volumetric dataset it is necessary to reconstruct the underlying 3D density function from the given samples. This reconstruction is achieved by convolving the samples with a reconstruction filter.

Sampling and Reconstruction

A point sample is represented as a scaled Dirac impulse function. Thus sampling a signal is equivalent to multiplying it by a grid of impulses, one at each sample point (see Figure 2.2).

The Fourier transformation of a two-dimensional impulse grid with frequency f_x in x and f_y in y is itself a grid of impulses with period f_x in x and f_y in y. The Fourier transform is invertible and transforms a function from the *spatial domain* to the *frequency domain* and vice versa. The convolution theorem says that the Fourier transform of a product of two functions is the convolution of their individual Fourier transforms, and vice versa:

$$\widehat{gh} = \hat{g} * \hat{h}; \widehat{g * h} = \hat{g}\hat{h} \tag{2.1}$$

To sample a continuous function $f(x)$ into a discrete function $d(x)$ at a uniform spacing s the function needs to be multiplied with a comb function $comb_s(x)$, where s is the spacing of the comb impulses:

$$d(x) = f(x)comb_s(x) \tag{2.2}$$

This is equal to a convolution of the Fourier transform of the initial function $f(x)$ with a comb function $comb_{1/s}$:

$$\hat{d}(u) = \hat{f}(u) * comb_{1/s}(u) \tag{2.3}$$

where $\hat{f}(u)$ is the Fourier transform of $f(x)$. $\hat{d}(u)$ is the function $\hat{f}(u)$ duplicated with a spacing of $1/s$. If the Fourier transform of the original function $f(x)$ is band-limited to the value range $[-1/2s, 1/2s]$ the duplicated copies of $\hat{f}(u)$ in $\hat{d}(u)$ do not overlap. Thus, the original function can be completely reconstructed by taking one copy and create its inverse Fourier transform. If $\hat{f}(u)$ has frequencies outside of this range then the original function cannot be recovered completely because information is lost due to the overlapping functions.

To avoid the effect of overlapping functions, called aliasing, the highest frequency component of $f(x)$ must be less than half the inverse of the sampling rate $1/s$. This is achieved if each signal is sampled with at least twice its highest frequency. This frequency is called the *Nyquist frequency*.

In order to reconstruct a perfect sampled signal, one copy of $\hat{f}(u)$ needs to get extracted from $\hat{d}(u)$. This can be done with multiplication in the frequency domain of $\hat{d}(u)$ with a rect function which is 1 inside $[-1/2s, 1/2s]$ and 0 everywhere else. This corresponds to a convolution in the spatial domain of $d(x)$ with a sinc function (the inverse transform of a rect function).

The sinc function is the perfect reconstruction filter but cannot be implemented because it has infinite extent. This makes it mandatory to use an imperfect reconstruction filter and thus leads to reconstruction artifacts.

Various reconstruction filters have been examined by Marschner and Lobb [28]. They classified the artifacts resulting from imperfect reconstruction into three main categories: smoothing, postaliasing and overshoot. Since reconstruction is necessarily imperfect, choosing a filter must involve tradeoffs between these three artifacts. For interactive performance a trilinear reconstruction filter (see Section 3.1) is recommended. The best results can be achieved with windowed sinc filters, but those filters are much more expensive than trilinear filters.

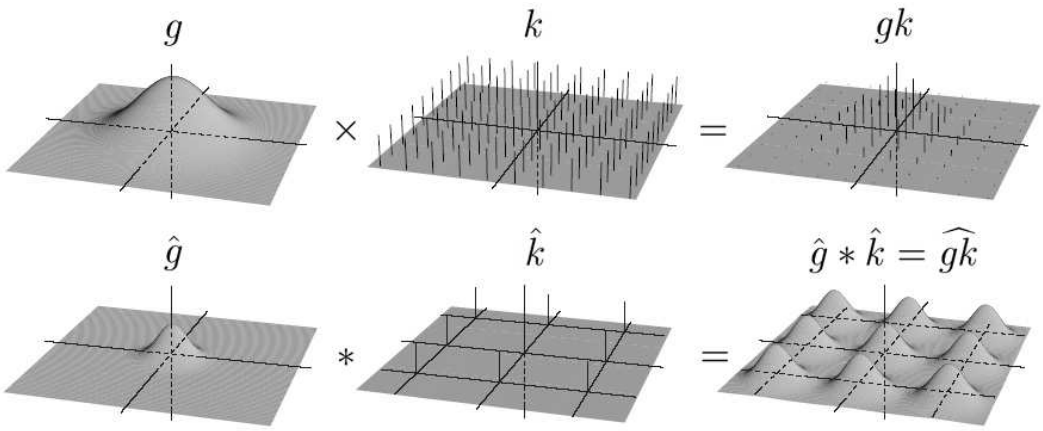


Figure 2.2: Two-dimensional sampling in the space domain (top) and the frequency domain (bottom) [28].

2.4.3 Gradient Calculation

To approximate the surface normals, which are necessary for shading and classification, the computation of a *gradient* is required. Given a continuous function $f(x, y, z)$, the gradient ∇f is defined as the partial derivative of the function with respect to all three coordinate directions:

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \tag{2.4}$$

The ideal gradient reconstruction filter is the derivative of the sinc filter, called cosc, which has infinite extent and therefore cannot be used in practise. Thus the continuous gradient has to be approximated using discrete *gradient filters*. One component of the gradient vector can be computed by convolution of the samples with the filter kernel. Most filters are derived from two-dimensional edge detection algorithms used in image processing. They are straight forward 3D extensions of those operators, such as the Prewitt or Laplacian operators.

The Sobel operator [39] is one of the most widely used gradient filters for volume rendering. Equation 2.5 shows an example for a Sobel gradient for the X component. G_x is the convolution kernel for the X component

of the gradient vector. The convolution kernels for G_y and G_z can easily be computed by rotation and alignment with the positive Y and Z axes, respectively. An analysis of several gradient filters for volume rendering has been published by Goss [14] and Bentum [3].

$$G_x = \begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} \begin{bmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{bmatrix} \begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} \tag{2.5}$$

Due to computational costs many ray-casting algorithms use a *central difference* or *intermediate difference* gradient, which are computed by local differences between sample values in all three dimensions [17]. The *central difference* gradient at voxel position (x, y, z) can be calculated as:

$$\nabla f(x,y,z) = \begin{bmatrix} \nabla f_x \\ \nabla f_y \\ \nabla f_z \end{bmatrix} \approx \frac{1}{2} \begin{pmatrix} f(x+1,y,z) - f(x-1,y,z) \\ f(x,y+1,z) - f(x,y-1,z) \\ f(x,y,z+1) - f(x,y,z-1) \end{pmatrix} \tag{2.6}$$

Usually the factor $\frac{1}{2}$ can be omitted, since the gradient vector is normalized before shading operations. For the *intermediate difference* gradient the equation is:

$$\nabla f(x,y,z) = \begin{bmatrix} \nabla f_x \\ \nabla f_y \\ \nabla f_z \end{bmatrix} \approx \begin{pmatrix} f(x+1,y,z) - f(x,y,z) \\ f(x,y+1,z) - f(x,y,z) \\ f(x,y,z+1) - f(x,y,z) \end{pmatrix} \tag{2.7}$$

2.4.4 Shading

Local illumination models use the basic vectors shown in Figure 2.3. Local illumination models take only direct reflections into account. This is not realistic but reduces the computation time. The equation to compute the reflected intensity at each sample is:

$$I = I_a + \sum_l I_l k_d (N \cdot L_l) + I_l k_s (V \cdot R_l)^s \tag{2.8}$$

The equation uses the following parameters:

I is the reflected intensity of the sample from the light source towards the viewer.

I_a is the reflected intensity due to ambient light. This intensity simulates indirect light reflections which would otherwise be unaccounted for by the illumination model.

I_l is the intensity emitted by light source l.

k_d is the diffuse reflection coefficient for the surface material.

k_s is the specular reflection coefficient for the surface material.

s is the specular exponent for the surface material. A higher value results in smaller and sharper highlights, while a lower value leads to large and soft highlights.

N is the normalized surface normal at the sample location.

L_l is the normalized vector from a point in space to light source l. For a directional light this vector is the same for every position in the scene.

V is the normalized vector from the sample location towards the viewer.

R_l is the normalized reflected light vector of light source l.

The term $N \cdot L$ which is the cosine of θ (see Figure 2.3) accounts for the diffuse reflection of the surface. The term $V \cdot R_l$ which is the cosine of ϕ accounts for the specular reflection of the surface. A smaller angle θ leads to a higher contribution of the diffuse term, and a smaller angle ϕ leads to a higher contribution of the specular term, respectively.

The shaded color c_{out} at a sample is calculated by multiplying the input color c_{in} (e.g., color obtained from a transfer function) with the reflected intensity I of the sample. The light color is assumed to be white and can thus be neglected.

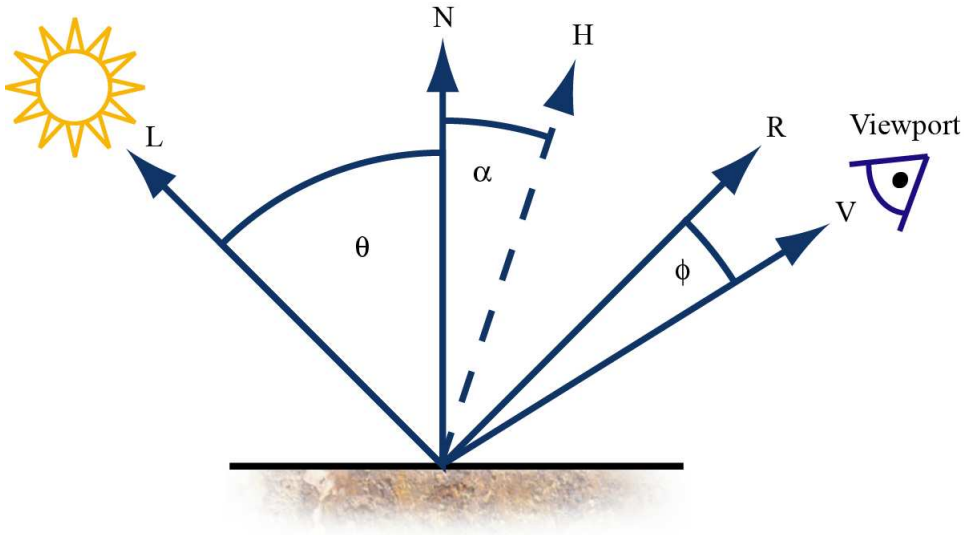


Figure 2.3: Vectors for Local Illumination Models. V is the vector towards the viewport and L is the vector towards the light source. N is the surface normal and H the halfway vector between L and V. R is the reflected light vector L.

$$c_{out} = c_{in} I \tag{2.9}$$

The evaluation of Equation 2.8 corresponds to the *Phong* Illumination model [33]. J. Blinn proposed an alternative formulation of the Phong model [7] which avoids the expensive computation of the reflection vector R. The model uses the halfway vector $H = \frac{L+V}{|L+V|}$ between the viewing vector V and the surface normal N (see Figure 2.3). It replaces $V \cdot R$ (which corresponds to $\cos\phi$) of Equation 2.8 with the dot product $N \cdot H$ (which corresponds to $\cos\alpha$). This approximation is accurate if the light sources and the viewpoint are assumed to be at infinity.

The expensive calculation of the exponentiation of the specular term can be avoided by Schlick's approximation [37] which is generally much faster to compute and shows similar visual results:

$$x^n \approx \frac{x}{n - nx + x} \tag{2.10}$$

For $k_s = 0$ the illumination model reduces to pure diffuse or Lambertian reflection [12]. Higher-order shading models, which include the physical effects of light material-interaction [10] such as subsurface scattering [15], are computationally too expensive to be considered for volume rendering.

2.4.5 Slicing

Slicing resamples the volume data on a plane perpendicular to one of the three major axes (x, y and z). This is a very simple visualization technique and allows the user to view any desired slice (plane) by changing the position of the plane within the dataset. Almost any system which deals with CT datasets offers this visualization method.

To highlight different properties within the dataset a method called windowing can be used. It defines an interval which maps each data value to a gray scale value. Data values below the lower boundary are mapped to black and values above the upper boundary are mapped to white. The values in between are interpolated accordingly.

It is standard in clinical practice to perform 2D examinations. However, due to the increasing size of the datasets it becomes more and more impractical as it is a very tedious and time consuming task. Objects are spread over several slices and the particular viewing direction for an acquired image sequence may not be optimal with respect to the real-world 3D anatomical structures. This makes it even harder to investigate the dataset with this method.

2.4.6 Direct Volume Rendering

Direct Volume Rendering (DVR) is a technique for directly displaying a sampled 3D scalar field without using an intermediate representation. The basic steps of *Direct Volume Rendering* consist of mapping the scalar values of each sample in the 3D dataset to optical properties such as color and opacity, projecting the samples onto an image plane, and then blending the projected samples. Some common methods for image composition are *Maximum Intensity Projection*, *Alpha Compositing* and *Non-Photorealistic Volume Ren-*

dering. Image composition can be used independently with any rendering technique like *Splatting* or *Ray-casting*. Rendering speed and image quality of four different direct volume rendering techniques have been evaluated by M. Meißner et al [29].

Ray-casting

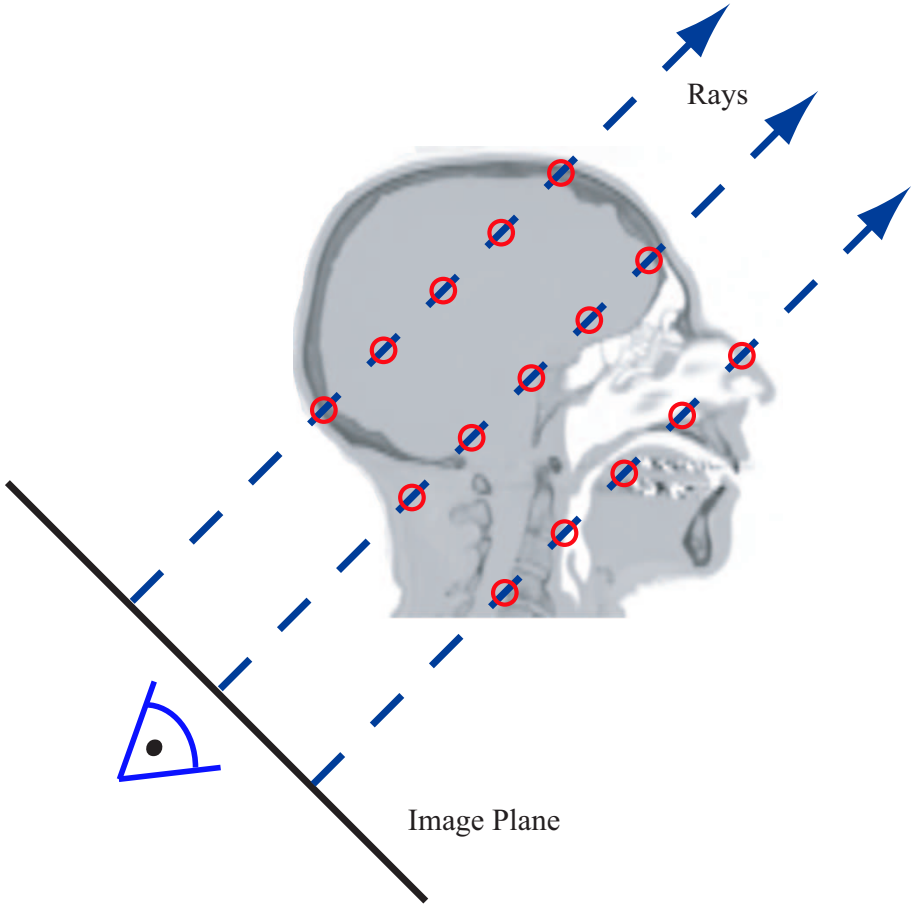


Figure 2.4: Raycasting: Rays are shot from the image plane into object-space. The rays are sampled at an equi-spaced distance (red circles).

Ray-casting is an image-order technique where rays from each pixel of the viewing plane are cast through the volumetric space and the final color and opacity are accumulated along each ray [34]. The volume is rendered

without extracting any surfaces. The formula for Ray-casting is

$$I_{i,j} = \sum_{k=0}^{L/\Delta t} f(P_{i,j} + k \cdot \Delta t \cdot r_{i,j}) \cdot \Delta t \tag{2.11}$$

The volumetric function $f(x, y, z)$ is sampled at equi-spaced distances Δt along the ray $r_{i,j}$ starting at $P_{i,j}$. The samples are accumulated and the resulting intensity $I_{i,j}$ is written to the image at location (i, j) (see Figure 2.4).

Maximum Intensity Projection

Maximum Intensity Projection (MIP) generates a high intensity image from a volumetric dataset by traversing each ray of the image plane and displaying the highest occurring value along a ray. Hereby high intensity regions can be occluded by regions with higher intensity. For medical datasets, for example, bones usually occlude all other parts of the CT-dataset (see Table 2.1).

The MIP compositing can be written as:

$$I_{i,j} = \max_{0<=k<=l/\Delta t} (f(P_{i,j} + k \cdot \Delta t \cdot r_{i,j})) \tag{2.12}$$

The volumetric function $f(x, y, z)$ is sampled at equi-spaced distances Δt along the ray $r_{i,j}$ starting at $P_{i,j}$. The highest intensity $I_{i,j}$ along each ray is written to the image at location (i, j).

Maximum Intensity Projection is a very popular way to visualize volumetric data. This technique is fairly forgiving when it comes to noisy data because only the highest value along each ray contributes to the final image. MIP produces images that provide an intuitive understanding of the underlying data. One problem is that spatial information is lost.

Figure 2.5 shows a Maximum Intensity Projection of the lower part of a human body. Here the aorta is occluded by bones due to higher intensity of the bones. To visualize blood vessels with MIP the dataset has either to be segmented so that all bones are removed or a contrast enhancing agent has to be injected before the CT-Scan is performed. Thereby, the blood vessels obtain higher intensity values.

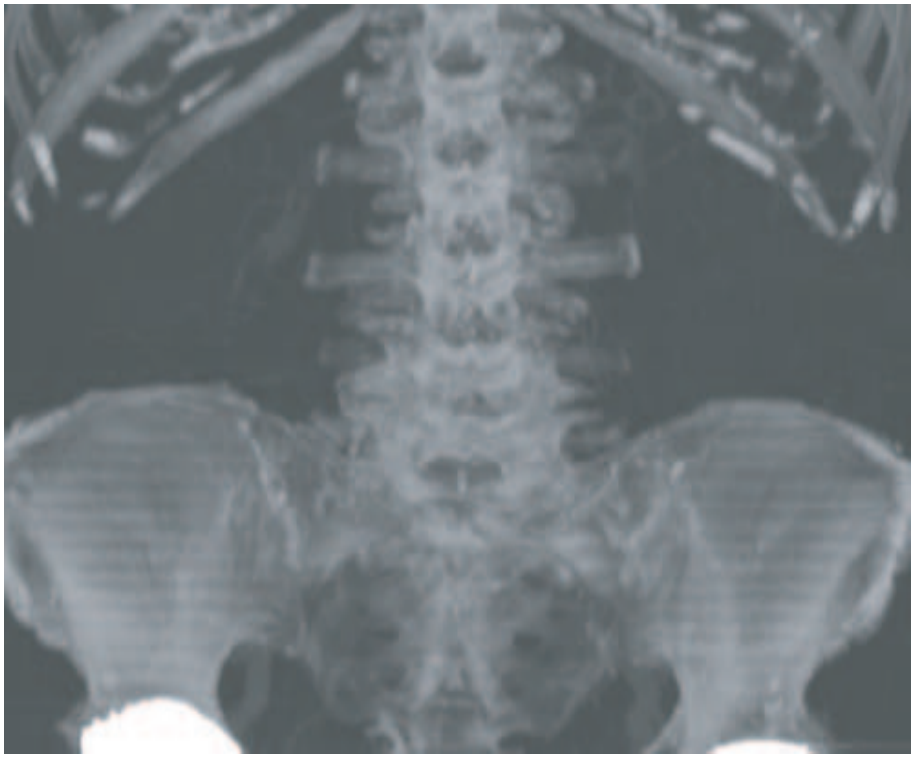


Figure 2.5: Volume dataset rendered with a Maximum Intensity Projection.

Alpha Compositing

par With *Alpha compositing* the final image is generated by integrating the density values along each ray of the image plane. The density values along a ray are mapped to color and opacity [25]. The final color and opacity for each pixel of the image plane can either be calculated in *front-to-back* (Equation 2.13) or *back-to-front* (Equation 2.14) order. The advantage of the *front-to-back* approach is that as soon as $\alpha_i \geq 1-\epsilon$ the ray can be terminated (early ray termination) and thereby the processing is accelerated.
The formula for *Front-to-back* Compositing is:

$$C_i = C_{i-1} + (1 - \alpha_{i-1}) \cdot \alpha(x_i) \cdot c(x_i) \tag{2.13}$$

with

$$\alpha_i = \alpha_{i-1} + (1 - \alpha_{i-1}) \cdot \alpha(x_i)$$

The formula for *Back-to-front* Compositing is:

$$C_i = C_{i-1} \cdot (1 - \alpha(x_i)) + \alpha(x_i) \cdot c(x_i) \tag{2.14}$$

C_i and α_i are the current accumulated color and alpha values along the ray. $c(x_i)$ and $\alpha(x_i)$ are the color and alpha values of the sample x_i (see Figure 2.6 for a dataset rendered with alpha compositing and shading).

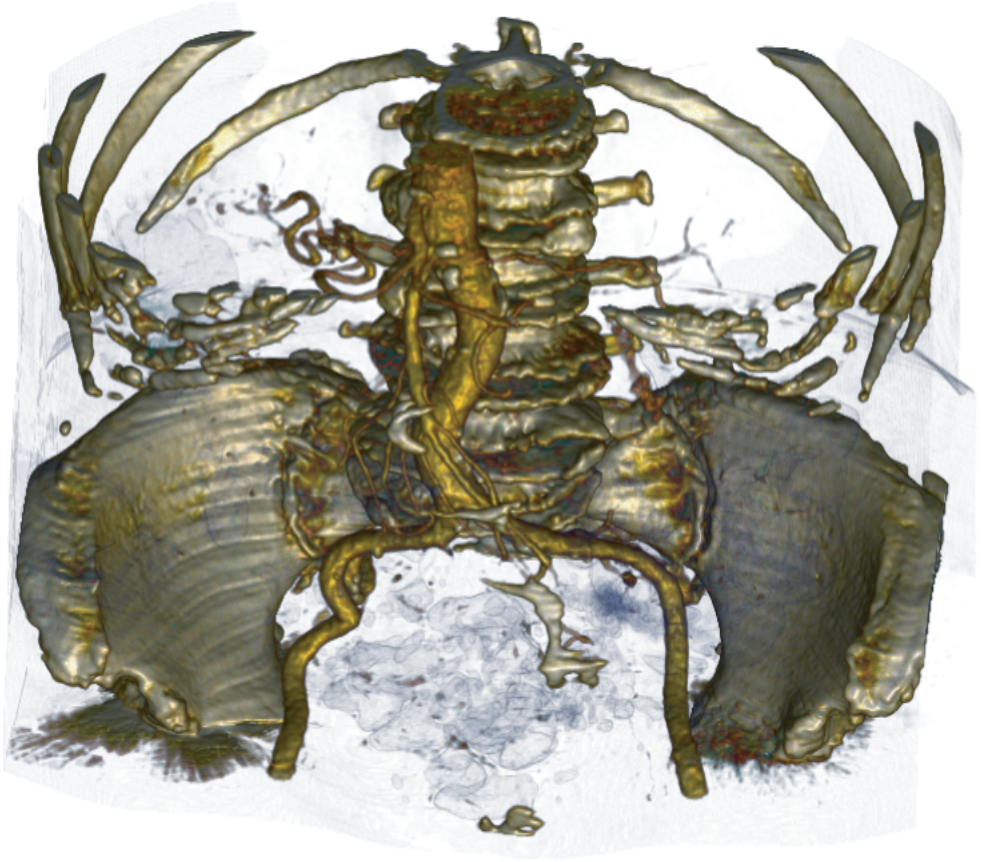


Figure 2.6: Volume dataset rendered with ray-casting using alpha compositing and shading.

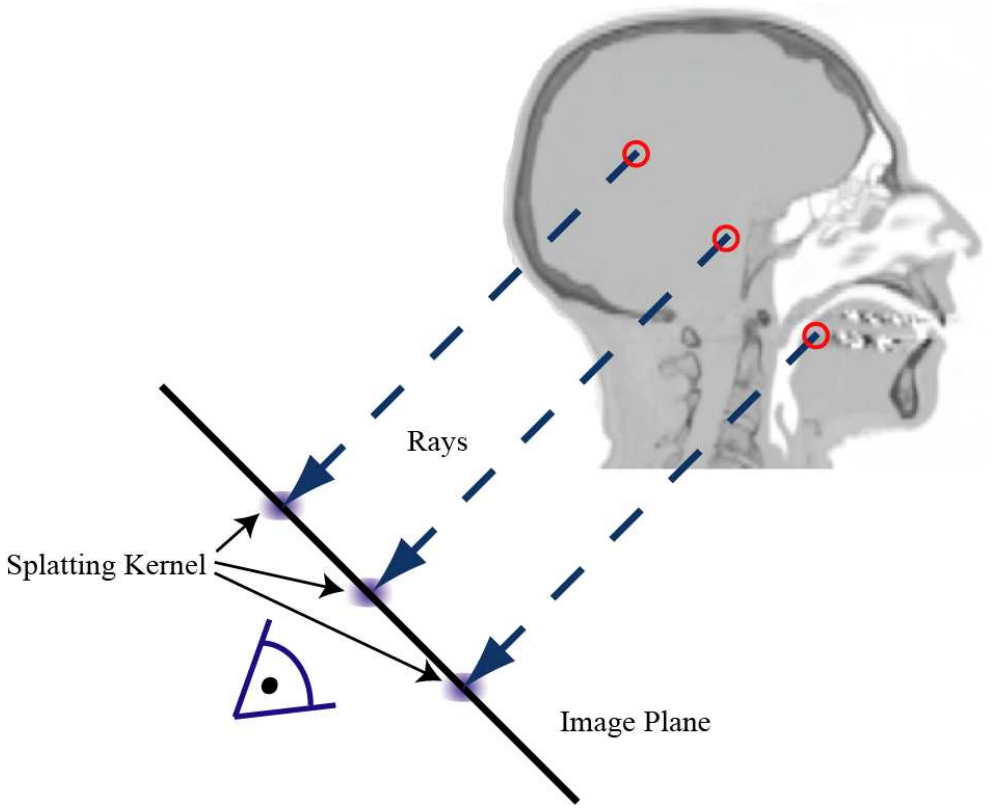


Figure 2.7: Splatting: Samples are projected from object-space to the image plane.

Splatting

Splatting differs from ray casting in the projection method (see Figure 2.7). Splatting is an object-order approach and projects voxels on the 2D viewing plane [43, 24]. It approximates this projection by a so-called Gaussian splat, which depends on the opacity and on the color of the voxel (other splat types, like linear splats can also be used). A projection is made for every voxel. The resulting splats are composited on top of each other in back-to-front order, which guarantees correct visibility, or in front-to-back order, which is faster, because the process can be stopped when pixels are fully opaque.

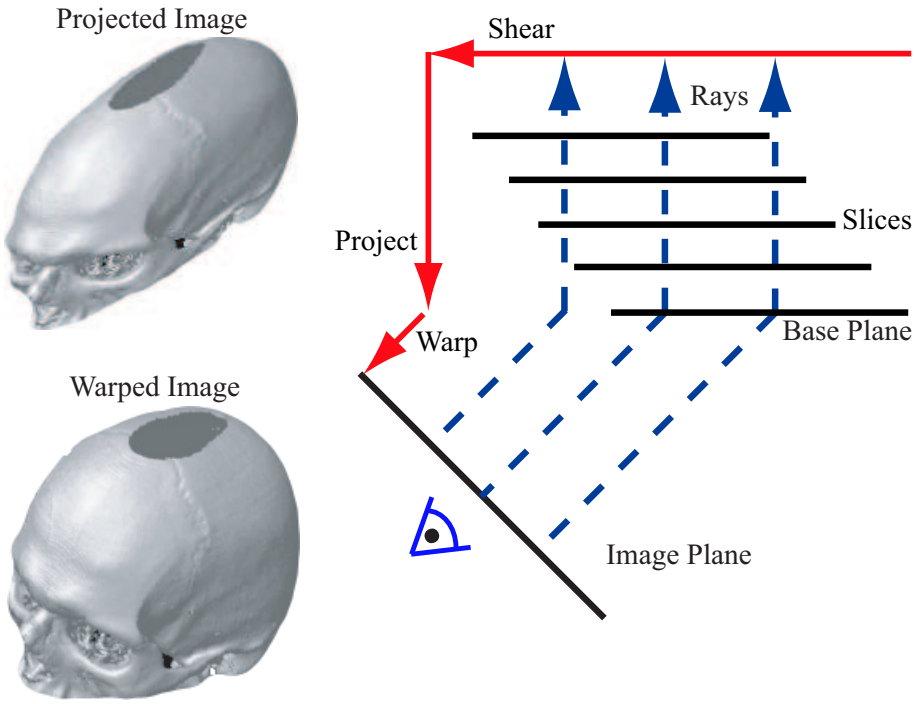


Figure 2.8: Parallel projection using shear-warp factorization of the viewing transformation. Parts of this image are from [30].

Shear-Warp Factorization

Shear-warp Factorization [23, 31] is considered to be the fastest software-based rendering approach. The algorithm is based on a shear-warp factorization of the viewing transformation. As can be seen in Figure 2.8, the voxels are sheared so that the viewing rays are orthogonal to the base plane. This avoids expensive address computations of the resampling positions inside the volume. The rendered image of this sheared volume is a distorted image which needs to get transformed by a 2D warp to result in the final undistorted image.

A problem of the shear-warp approach is that current algorithms use only a 2D reconstruction filter which may cause image artifacts. Furthermore, the sample rate is dependent on the viewing direction and a pre-classification is needed. Solutions for some of these problems exist [41], but the high image quality of some other methods like ray-casting still cannot be achieved.

2.4.7 Curved Planar Reformation

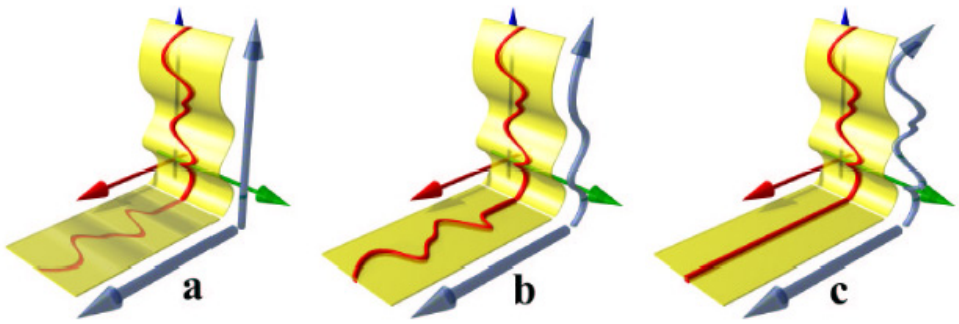


Figure 2.9: Different CPR types: (a) Projected CPR, (b) Stretched CPR, (c) Straightened CPR [20]

Curved Planar Reformation (CPR) is a very important visualization technique for tubular structures in medical imaging that generates longitudinal cross-sections to show the structures and their surrounding tissue in a curved plane [20]. To investigate specific objects with a CPR the centerline of those objects has to be extracted first (see Section 2.3). The advantage of this method is that the object of interest can be viewed in its entireness and without any occlusions. Radiologists can then easily detect vascular abnormalities (i.e., calcifications, stenoses, occlusions and aneurysms).

An image rendered with a CPR can be seen in Figure 2.10. It shows the aorta of a human body in its entirety without any occlusion at a longitudinal cross-section. If the viewing direction of the cross-section is rotated then any disease can be recognized immediately since the whole aorta is visible.

The three common types of how the projection of the plane is performed are (see Figure 2.9): projected CPR, stretched CPR and straightened CPR.

The projected CPR provides a good overview, allows fast inspection and is easy to handle. It resamples the data line-wise along so called generating lines and projects it to the corresponding pixel of the image plane. The projected CPR offers a good spatial perception but does not maintain isometry. Furthermore, occlusions are possible and geometric artifacts can occur.

The stretched CPR resamples each generating line from the dataset and

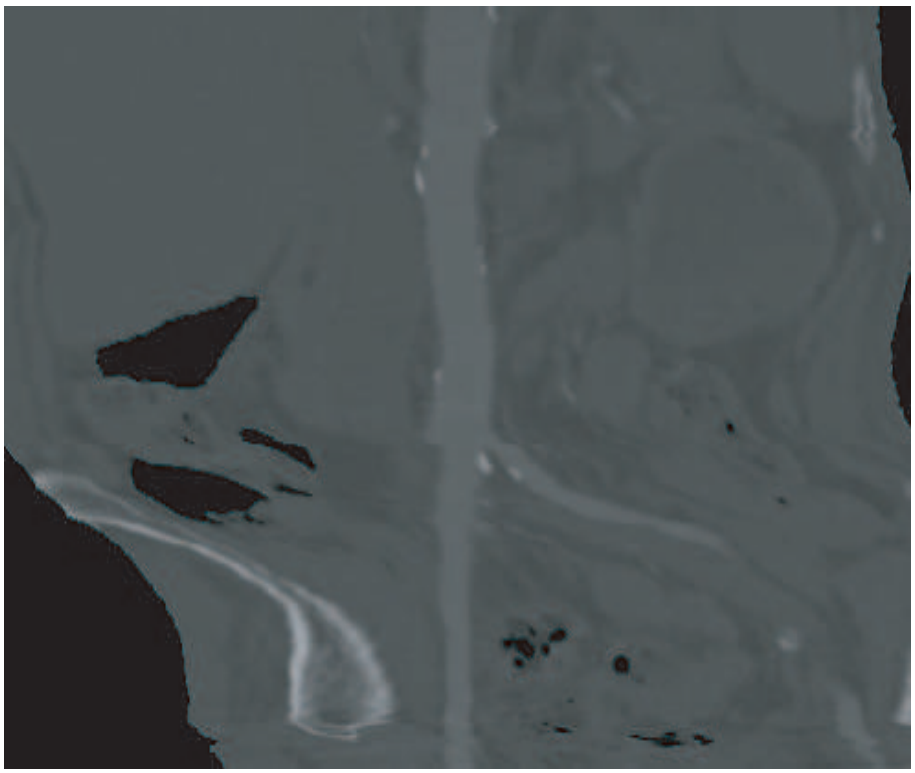


Figure 2.10: Volume dataset rendered with a Curved Planar Reformation

unrolls the plane onto the image. The height of the image depends on the vessel length and the curvature of the vessel in the image. The advantages of this projection method are: length-preservation, reduction of geometric artifacts and no possible occlusions. The disadvantage is sensitivity to high curvature within the vessel.

The straightened CPR resamples each generating line from the cross-section of the vessel. The height of the image only depends on the vessel length. The straightened CPR has the advantages of length-preservation and no possible occlusions. The disadvantages are no spatial relationship in the image and undesired artifacts of rotating coordinate frames.

2.4.8 Indirect Volume Rendering

Unlike *Direct Volume Rendering*, *Indirect Volume Rendering* generates an intermediate representation of the volume data in a pre-processing step. It assumes that the volume contains surfaces. The output of this reconstruction step are polygons (i.e., triangles) which can be rendered by conventional surface rendering algorithms [25]. Even though surface rendering algorithms are widely supported they do not enable the user to see inside structures or render 'fuzzy' objects. Additionally, surfaces are incorrectly depicted as having no thickness. As a consequence, important data is often incorrectly shown or hidden from the user. These problems often lead to erroneous conclusions and lost productivity.

The most common technique for surface reconstruction is the Marching Cubes algorithm [26]. The algorithm creates triangle models of constant density surfaces from 3D data. It uses a divide-and-conquer approach to generate inter-slice connectivity, and creates a case table that defines triangle topology. The 3D medical data is processed in scan-line order and triangle vertices are calculated using linear interpolation. The gradients of the original data are normalized and used as a basis for shading the models. The detail in images produced from the generated surface models is the result of maintaining the inter-slice connectivity, surface data, and gradient information present in the original 3D data. Other common methods for surface reconstruction are Contour Tracking [22], Opaque Cubes [16], Marching Tetrahedra [38] and Dividing Cubes [9].

2.5 Summary

For medical data diagnosis a wide range of very useful visualization techniques exist. Curved Planar Reformation is a very powerful rendering algorithm for CTA. It shows tubular structures in its entirety without any occlusion. Direct Volume Rendering Algorithms render volumes without extracting any surfaces. Maximum Intensity Projection generates a high intensity image, while Alpha Compositing integrates the density values along each

ray of the final image. Indirect Volume Rendering needs to extract surfaces from the volume data first. This can sometimes lead to erroneous conclusions during investigations because surfaces have no thickness.

Chapter 3

Data structures

640K ought to be enough for anybody.

Bill Gates

This section gives an overview of the most common data structures that are used for volumetric data. The data structures are analyzed with respect to medical datasets where often certain regions (e.g., blood vessel, colon) are of interest. To define which regions are of interest a segmentation process needs to be performed at first (see Section 2.3). These kind of datasets can be very large (see Section 2.1) and often can not be stored entirely in memory. It would be advantageous to represent the non-important regions of the dataset with lower resolution or accuracy while maintaining the important information. A loss (or unintended decrease of resolution) of important data can have serious consequences because it could lead to a misdiagnosis of the medical dataset by a radiologist.

The data structures are also investigated regarding an applicable resampling technique and its performance. This is very important since it should be possible to perform the visualization in an adequate time with accurate results.

3.1 Mesh

A Mesh is the easiest and most straightforward data structure to store volumetric data. For a rectilinear grid, the position of each sample is given implicitly by its position within the data structure.

Memory Usage

The position of the voxels needs not be stored explicitly, therefore only 2 bytes (for the CT-data) per voxel are necessary. This seems very efficient but the big disadvantage of a grid is that even unimportant data is explicitly stored with full resolution. Usually a dataset with the dimensions $x \times y \times z$ needs a total of $2 \cdot x \cdot y \cdot z$ bytes. The only way to reduce memory space for a grid is to define the bounding box enclosing the regions of interest and store only the grid of the bounding box. This can work well for fully covered box-shaped regions, but usually it results in storing a lot of non-important samples. For example, if the object of interest is very thin but goes diagonally through the volume, the bounding box would be the size of the whole volume.

Resampling and Performance

The resampling in a rectilinear grid is very fast and easy to implement. Usually trilinear interpolation is used for reconstruction. The resampled value at a given position is a trilinear interpolation of the eight closest voxels (see Figure 3.1).

Each position f_{xyz} with x, y and z either zero or one denotes the sample value of one corner of the voxel (see Figure 3.1). The value at position (x,y,z) within the cube is denoted by f_{xyz} and is given by

$$
\begin{aligned}
f_{xyz} = {} & f_{000} \cdot (1-x) \cdot (1-y) \cdot (1-z) + \\
& f_{100} \cdot x \cdot (1-y) \cdot (1-z) + \\
& f_{010} \cdot (1-x) \cdot y \cdot (1-z) + \\
& f_{001} \cdot (1-x) \cdot (1-y) \cdot z + \\
& f_{101} \cdot x \cdot (1-y) \cdot z + \\
& f_{011} \cdot (1-x) \cdot y \cdot z +
\end{aligned}
$$

$f_{110} \cdot x \cdot y \cdot (1 - z) +$
$f_{111} \cdot x \cdot y \cdot z$

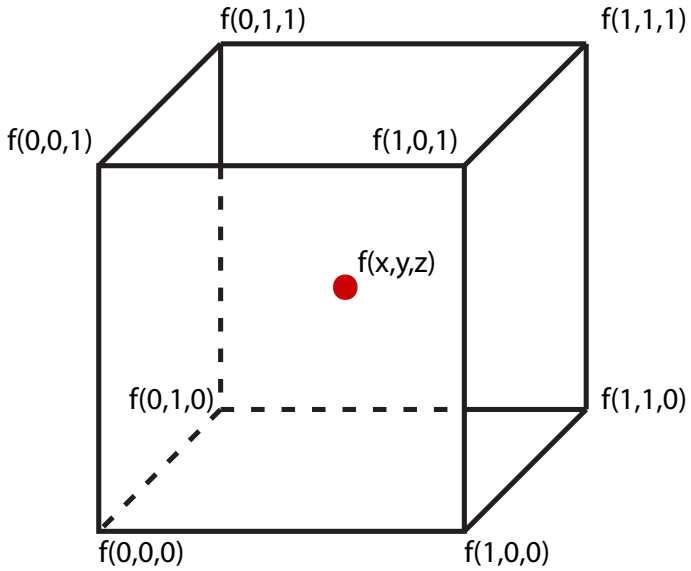


Figure 3.1: Trilinear interpolation: A value inside the cell is resampled by a weighted sum of all eight voxels.

3.2 Octree

An octree is a hierarchical tree structure, where each node corresponds to a region of three-dimensional space. Using this kind of hierarchical space partitioning, spatial coherence can be exploited to reduce storage requirements for three-dimensional objects.

The octree encoding procedure for a three-dimensional space is an extension of an encoding scheme for the two-dimensional space, called quadtree encoding. Quadtrees are generated by successively subdividing a two-dimensional region into quadrants until a desired depth is reached. For octrees the three-dimensional space is recursively subdivided into octants, thereby each node contains eight child nodes.

Figure 3.2 shows an example of a quadtree. Each quadtree node which is not homogenous (in our case homogenous means that a node has either

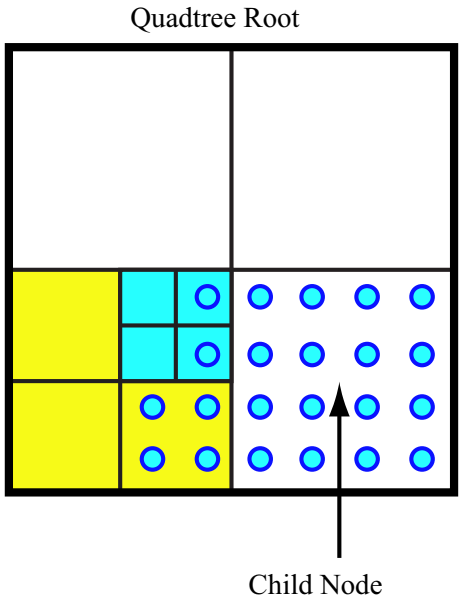


Figure 3.2: Quadtree: Example of a quadtree subdivision.

no samples at all or is completely filled with samples) gets subdivided. The quadtree root node is subdivided into four quadrants. The two upper child nodes and the lower right child node (white quadrants) are homogenous and thus do not need to get subdivided any further. The lower left child node is subdivided again. Its upper and lower left child nodes are empty and its lower right node is completely filled (yellow quadrants) and need no further subdivision. Only its upper right node gets subdivided into four new child nodes (turquoise quadrants).

Memory Usage

An *Octree* can now be generated in a way that a node is only subdivided if an octant contains a region of interest that needs finer resolution than the current octant. To avoid having too many octree nodes the level of subdivision should be limited. A dataset with the dimension of $2^n \times 2^n \times 2^n$ can have a maximum number of n levels. If the octree node boundaries are aligned on the voxels many nodes produce an overhead because lots of voxels are stored twice (or more times at the corners). This can be seen in the

quadtree example in Figure 3.3 where green samples are stored twice because they occur on a boundary between two nodes. The red sample is even stored 3 times, in the upper right and the two lower child nodes. Each node also produces an overhead for the pointers to the octants, thus it's desirable to reduce the number of nodes.

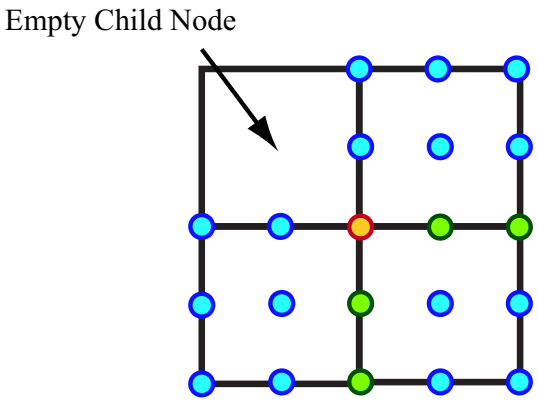


Figure 3.3: Samples at the boundary between two filled quadtree nodes are stored multiple times.

The actual memory usage of a dataset depends on the objects of interest and how they are aligned. An octree node is always divided into eight equally sized cubes, so especially important objects around the center of the octree root node could produce bad results. Each octant would cover only a small region of the important objects and thus many octree nodes would be required.

Resampling and Performance

Neighboring nodes of the octree share the voxels on their common faces, thus each node stores the voxels on the boundary on its own (see green and red samples in Figure 3.3). This produces some overhead but makes resampling much easier and faster. Every leaf of the octree stores a conventional mesh (see Section 3.1) of the corresponding samples. Thus, the resampling inside a leaf can be handled like on a normal mesh since each leaf contains all necessary voxels.

The performance is very good since a normal trilinear interpolation can be performed and lots of unimportant regions are not considered because they are not stored in any node. Further it is very efficient to find the node of the current resampled position since the octree is always split into eight equally sized elements.

3.3 Adaptive Mesh Refinement

Adaptive Mesh Refinement (AMR) generates a hierarchical structure of meshes where each level of the structure corresponds to a specific resolution of the underlying data. AMR was introduced to computational physics by Berger and Oliger [5]. A modified version of their algorithm was published by Berger and Colella [4]. AMR has become very popular and is used in a variety of applications.

An AMR consists of a hierarchical structure of axis-aligned rectilinear grids. Each grid consists of cells which are defined by its cell width. The cell width corresponds to the resolution of the grid (a higher cell width results in a lower resolution and vice versa) and has to be constant in every axial direction.

Each grid can contain an arbitrary number of children grids as long as they do not overlap. A children grid has a finer resolution than its parent grid, thus it is referred to as a *fine grid*. The parent grid is called *coarse grid* because it has a coarser resolution than its children. The *refinement ratio* defines how many cells of a fine grid fit into a cell of a coarse grid. The root node of the AMR structure is the coarsest grid of the hierarchy. Each grid must be aligned at the boundaries of its parent grid cells.

Figure 3.4 shows an AMR hierarchy. The thick lines mark the boundaries of a grid while the thin lines show the cell boundaries. The turquoise grid is the root node and has a cell width of four. It contains two fine grids (red and orange box) with a cell width of two. The left grid (red box) again contains another fine grid with a cell width of one, which is the finest resolution. The refinement ratio is four for all grids because a coarse grid cell is always represented by four fine grid cells.

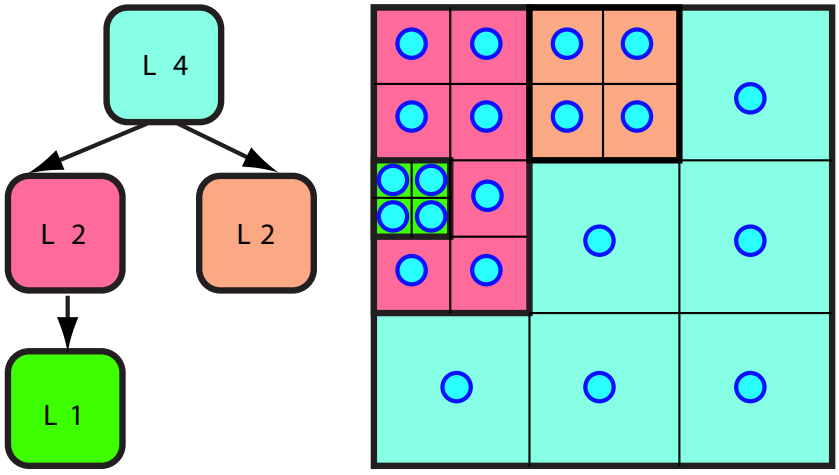


Figure 3.4: AMR hierarchy: The root level L4 has a cell width of 4 and contains two fine grids of the level L2, and one of those two grids contains one fine grid with the finest resolution L1.

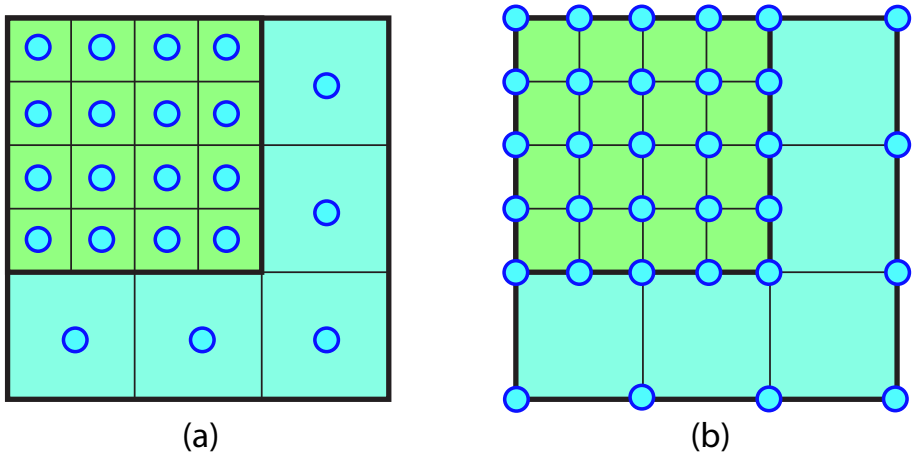


Figure 3.5: AMR data formats: (a) cell-centered (b) grid-aligned

The grids can either use a *cell-centered* data format where the values are associated with the centers of cells, or a *grid-aligned* data format which stores the values at the grid points (see Figure 3.5).

Memory Usage

A grid of an AMR has the same memory requirements for the sample points as a mesh. Each grid also has to store the boundaries of the grid and the pointers to the finer grids and optional a pointer to the coarser grid. The benefit of an AMR is that the grids can be created according to the underlying data. Grids will only be created were they are needed and thus non-important regions need not to be stored. Adaptive Meshes also offer the possibility to store data in a reduced resolution. The storage space of AMR mainly depends on how the grids are generated and if they are well aligned to the important regions. A *grid-aligned* format with neighboring grids sharing samples at their common faces is preferred. Adaptive Meshes are very efficient since the sample positions are stored implicitly and the overhead for a mesh is very low.

Resampling and Performance

Resampling inside a grid (with a *grid-aligned* format) is performed the same way as in a mesh and thus is very efficient. The only difference to a mesh is that the finest grid of the current position has to be found which can be a time consuming process. Starting at the root grid the current point needs to be tested against the boundaries of all finer grids. If it is inside a finer grid the same procedure has to be repeated recursively.

3.4 Point Clouds

Point clouds consist of a set of unstructured sample points. Each sample point is stored individually and is encoded as position and sample value. This section shows some problems that arise with all structures that have to deal with sample points without a mesh structure in some way.

Memory Usage

The spatial resolution of CT-Data is usually higher than 256 for every axis so 2 bytes are needed for the x, y and z position. This makes 6 bytes for every

sample along with the 2 bytes for the data itself. Thus the original volume has to be reduced to at least 1/4 to gain any memory space reduction.

Resampling and Performance

Resampling of point clouds without an indexing structure is nearly impossible because all samples would need to be examined and thus makes it impracticable.

Even with a given indexing structure the resampling of sample points is an issue. There is no best way for resampling and the used method mainly depends on the field of application and performance demands. For example, a point could be resampled with the closest sample point, the n nearest sample points weighted equally, with a reconstruction kernel, or with an interpolation of precalculated density values over equally spaced regions.

3.5 KD-Tree

A *kd-tree* is a data structure for storing a set of points from a k-dimensional space [2, 13]. A *kd-tree* is a binary tree. Each sample is represented as a node element in the kd-tree. The contents of each node is depicted in Table 3.1.

field name	field type	description
sample-pos	position	x,y and z coordinates of sample position
sample-val	value	data value of sample
split	integer	the splitting dimension
left	kd-tree	a kd-tree for the points to the left of the splitting plane
right	kd-tree	a kd-tree for the points to the right of the splitting plane

Table 3.1: Kd-tree node values

The *sample-pos* field represents the position and the *sample-val* field represents the actual value of the sample. Each kd-tree node has a splitting

plane which passes through *sample-pos* and is perpendicular to the direction stored in the *split* field. The splitting plane divides all points into a left and a right subspace. The left kd-tree contains only points which are to the left of the splitting plane, and the right kd-tree stores only those points to the right. If a node has no children, then no splitting plane is needed.

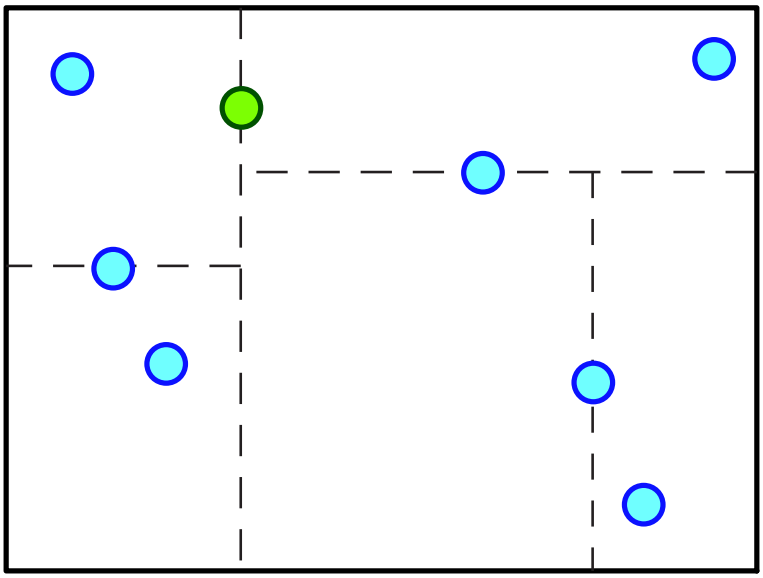


Figure 3.6: Kd-tree: Example of a two-dimensional kd-tree

An example of a two-dimensional *kd-tree* can be seen in Figure 3.6. Each line represents a splitting plane that divides the points into a left and right subtree. A node is subdivided until its left and right subtree contains no more than one point. The green sample represents the root node of the kd-tree.

Memory Usage

The *kd-tree* needs even more memory than point clouds because every node in the *kd-tree* has some overhead for the values shown in Table 3.1.

Resampling and Performance

The problem of resampling is the same as described in Section 3.4. The performance is usually quite good but mainly depends on the distribution of

the samples in the tree. To find the nearest neighbor of a position at least $O(logN)$ inspections are necessary and could go up to a maximum of N (each node will be visited at most once).

3.6 O-Buffer

The O-Buffer [35] is a framework for sample based graphics where the samples are not restricted to a regular grid. The position of a sample in the O-buffer is recorded as an offset to the nearest grid point of a regular base grid.

The O-buffer can be classified into three categories:

- Uniform O-buffer: The number of offset samples stored in every grid cell of the O-buffer is the same
- Nonuniform O-buffer: the number of offset samples stored in every grid cell of the O-buffer is not the same
- Adaptive O-buffer: the regular grid is adaptive

Memory Usage

The size of the O-buffer depends on the regions-of-interest in the dataset and on the type of O-buffer that is used. For medical datasets where only a small part of the dataset is important (e.g., blood vessels) an adaptive O-buffer should be used. This allows to refine the cells of the regular grid where it is needed. The memory space for the offset of the samples can be reduced by storing the offset to the nearest grid point. Additionally to the offsets of the samples the information of the overlying adaptive grid has to be stored. If an O-buffer representation needs less memory than a regular Octree mainly depends on how many samples have to be stored in each cell. If 2 bytes for the sample offsets are used (this results in 32 offsets for each axis), the O-buffer should not store more than half of the samples per cell to require less memory than an octree cell.

Resampling and Performance

Since the O-buffer is a framework for sample points it faces the same issues as point clouds. The hierarchical structure makes it easier to handle samples with spatial coherence but it still does not solve the problem which reconstruction kernel should be used. This could lead to some undesired resampling artifacts.

3.7 Conclusion

To store large medical datasets where only certain objects (regions) are of interest, a more sophisticated data structure is required. Since the dataset does not fit into physical memory entirely it is necessary that the data structure can cut off most parts of the non-important regions. This is usually not possible with a basic data structure such as a mesh. Point-based structures (e.g., kd-tree) are a good choice for very sparse data but need too much overhead for bigger objects. Because the position has to be stored for every sample, bigger objects can be stored much more efficiently with other data structures.

Octrees work very well because they can skip non-important regions and offer a great visualization performance. Nevertheless their memory performance mainly depends on the spatial properties of the objects (i.e., how they are aligned to the octant boundaries).

Adaptive meshes are a good option since they can adapt their mesh boundaries to the underlying data. They also offer the possibility to store regions in a reduced resolution to give a good overview of the dataset without requiring too much memory.

Chapter 4

Adaptive Meshes for Large Datasets

Any sufficiently advanced technology is indistinguishable from magic.

Arthur Clarke

This chapter describes our *Adaptive Mesh Refinement* (AMR) implementation which can be used for large medical datasets. It allows to store the data in a resolution according to their importance. The primary objective of the data structure is to reduce memory consumption. It is described how the *Adaptive Meshes* are implemented and how the meshes are generated so that they represent the underlying data with appropriate accuracy. Finally the visualization process is shown using our data structure based on *Adaptive Meshes*.

Our AMR data structure operates on a CTA dataset. The dataset contains the segmentation information for the aorta. The aorta is characterized by a centerline and a radius for the control points of the centerline. The radius defines the thickness of the aorta at a given point. The algorithms in the following sections can be applied to any dataset where an object of interest is described by a centerline.

4.1 Adaptive Mesh

The general AMR has been introduced in Section 3.3. For simplicity reasons the following descriptions are only for two dimensions. Nevertheless they can be easily extended to 3D. We implemented the AMR in a way that the resolution of each mesh has to be a power of two. A resolution of n means that only every n^{th} sample in each axial direction is stored. The refinement ratio of a refined grid is always set to eight. Although the general AMR restricts finer grids to be aligned to the boundaries of grid cells of the parent level, our implementation also allows finer grids to start or end at the center of a parent grid cell for greater flexibility. The *Adaptive Meshes* are stored in a grid-aligned format, thus neighboring grids share data values at their common faces. The coarsest grid stores every point according to its resolution. Finer grids only store points that are not already stored in any coarser grid. As can be seen in Figure 4.1, the outer grid is the coarsest grid of the AMR structure with a cell width of two and stores all samples according to its resolution (green points). The inner grid (orange box) has a cell width of one but stores only samples which are not already stored in the outer grid (red points). This offers the possibility to store multiple resolution levels without generating any overhead for the data.

The sample points of each grid are stored in a one-dimensional array. Each grid is defined by the two points p_{min} (origin of the grid), p_{max} (endpoint of the grid) and the refinement level $reflevel$. $reflevel$ defines the cell width for all axial directions. $plevel$ is the refinement level of the parent mesh ($plevel = reflevel \cdot 2$). The relative position p_{rel} inside a grid of an absolute point p_{abs} is calculated as

$$p_{rel} = (p_{abs} - p_{min})/reflevel \tag{4.1}$$

For the coarsest grid level the address of the relative position in the data array is simply given as

$$addr = p_{rel,y} \cdot ((p_{max,x} - p_{min,x})/reflevel) + p_{rel,x} \tag{4.2}$$

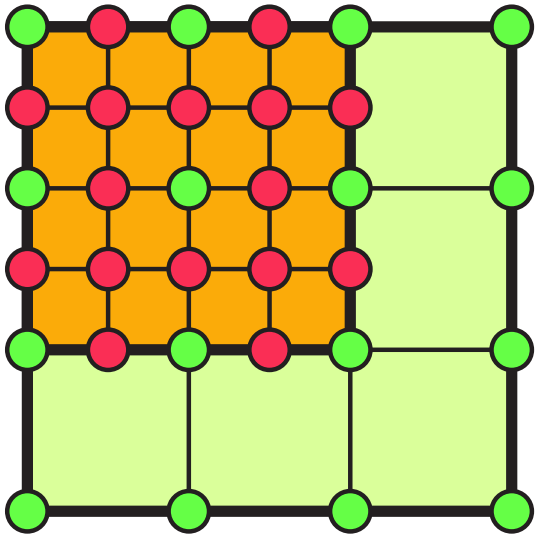


Figure 4.1: AMR implementation: green points are stored in the outer (coarser) mesh, red points are stored in the inner (finer) mesh

For a refined grid the address of a position in the one-dimensional array is

$$addr = \lfloor((p_{rel,y} + 1)/2)\rfloor \cdot s_1 + \lfloor(p_{rel,y}/2)\rfloor \cdot s_2 + \lfloor p_{rel,x}/(p_{max,x} - p_{min,x}) \cdot s_{sy}\rfloor \tag{4.3}$$

where s_1 is the number of points of every other slice starting at the first slice, s_2 is the number of points of every other slice starting at the second slice and the index sy in s_{sy} is calculated as $sy = p_{rel,y}\ mod\ 2 + 1$. The computations of s_1 and s_2 are shown in Table 4.1 and Table 4.2 respectively.

4.2 Preprocessing for Mesh Generation

4.2.1 Marking Data

We introduce three different types of importance: high importance, medium importance and low importance. Areas marked with high importance need to be stored in a grid with a cell width of one which corresponds to the highest possible resolution. Areas marked with medium importance need to

case	**s1**
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$\lfloor p_{max,x} - p_{min,x}/plevel \rfloor$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$\lfloor (p_{max,x} - p_{min,x} + 1)/plevel \rfloor$
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$

Table 4.1: First slice size for different grid alignments

case	**s2**
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$\lfloor p_{max,x} - p_{min,x}/plevel \rfloor$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$\lfloor (p_{max,x} - p_{min,x} + 1)/plevel \rfloor$

Table 4.2: Second slice size for different grid alignments

be stored in a grid with a cell width of at least two. Finally, areas marked with low importance need to be stored in a grid with a least a cell width of four.

First of all the data has to be categorized so that each sample point has a specific importance. According to the importance an appropriate grid resolution is chosen. All points that are close to the centerline (path of the object of interest) are very important which means that these points must be stored in full resolution in the *Adaptive Meshes*. The points which are farther away but still within a certain range to the centerline have medium importance, therefore these points are stored at least in half resolution (every second point in each axis has to be stored).

Figure 4.2 illustrates how the samples are categorized. Thick points are

marked as very important because they are close to the centerline (red dashed line). Thin points are marked with medium importance because they are farther away but still within a certain range to the centerline. All remaining points are not marked at all. Alternatively they could be marked with low importance to keep them in the lowest available resolution.

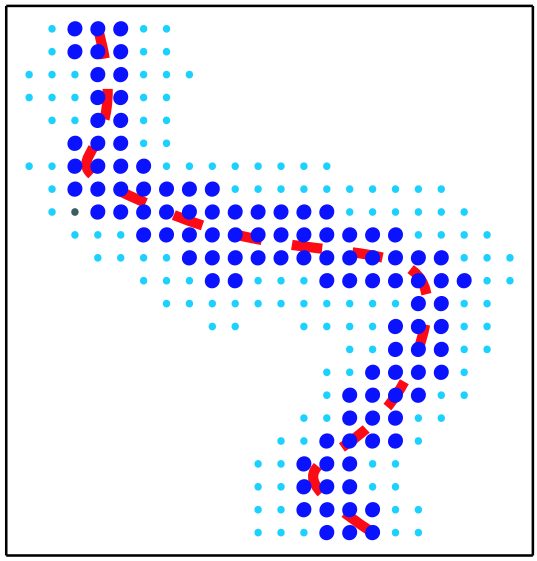


Figure 4.2: Marked data: importance of samples is set according to distance to the centerline (red). Thick points will be stored in full resolution, thin points will be stored in half resolution or higher.

Furthermore each sample point is assigned the major axis alignment of the tangent at the corresponding point on the centerline. The major axis alignment is the component of the tangent vector with the highest magnitude. The major axis alignment is needed for the *Growing Algorithm* (see Section 4.3.1).

4.2.2 Calculation of Volume Density

In this section we explain how the volume density d is calculated. The volume density defines the percentage of important points in a volume. It is needed for the algorithms in Section 4.3.1 and Section 4.3.2 to determine if a volume can be expanded further and requires further subdivision respectively. For

easier understanding the following explanations are made for areas instead of volumes. Furthermore, the volume density refers to a specific importance level. The important points are all points which have the specified importance or higher. For example, if the importance level is medium importance then all points with medium or high importance are called important points.

In order to quickly determine the number of important points for any given area a summed area table is build. We refer to an entry of the summed area table as $D(x, y)$. The summed area table contains the number of important points from $(0, 0)$ to any point (x, y) of the whole area.

$$D_{(x,y)} = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j) \tag{4.4}$$

where $I_{(x,y)}$ returns 1 if the point at the position (x,y) is important and 0 otherwise.

The summed area table is calculated with dynamic programming with the following formula:

$$D_{(x,y)} = I(x, y) + D_{(x-1,y)} + D_{(x,y-1)} - D_{(x-1,y-1)} \tag{4.5}$$

The number of important points for the boundaries with $x = 0$ or $y = 0$ have to be calculated separately before.

The number of important points for any area from p_{min} to p_{max}, where the entry of the summed area table for each corner is given by D_i (see Figure 4.3), can be calculated as

$$D = D_3 - D_2 - D_1 + D_0 \tag{4.6}$$

The density d of an area is the ratio between the number of important points and the size of the area:

$$d = D/((p_{max,x} - p_{min,x}) \cdot (p_{max,y} - p_{min,y})) \tag{4.7}$$

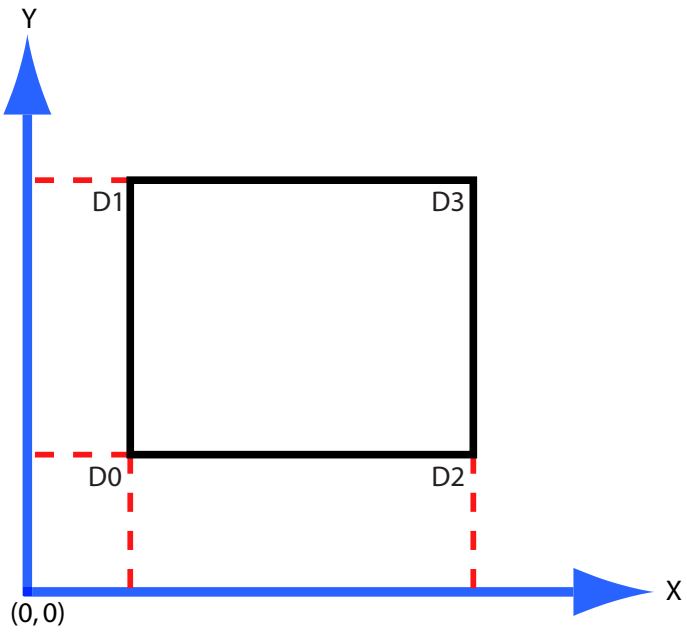


Figure 4.3: Summed area table: Table contains number of important points for all areas from origin to any point.

4.3 Mesh Generation

We present two different algorithms for clustering cells into axis-aligned regions. The algorithms try to create meshes that have a density d over a certain density threshold value. This guarantees that the whole AMR stores at least a certain amount of important samples in every mesh. The higher the density threshold is set, the more meshes will be generated. Thus a higher density threshold usually results in less memory consumption as long as the meshes do not become too small and produce too much overhead (for the mesh information and the sample points which are shared along the common faces of two neighboring meshes). The summed volume table is build for both algorithms for fast computation of the volume density of any volume.

4.3.1 Growing Algorithm

The *Growing Algorithm* (see Table 4.3) starts at the coarsest refinement level. A bounding box of all points with an importance (low, medium, or high) is calculated and passed to the initial call of **FindBoxesEXP** as box_{in}. The algorithm creates grids so that all important points (points with an importance according to the current refinement level or higher) are covered, and stores them in $boxes_{out}$. As long as there is an important point inside box_{in} which is not covered by a box of $boxes_{out}$ yet (Line 1), this point (Line 2) is used for the initial position of a new box box_{new} (Line 3). This new box gets expanded (Line 4) and the expanded box is added to $boxes_{out}$ (Line 5). The points inside this box are cleared so that they will not be stored by further meshes and thus overlapping grids are avoided (Line 6). The boundaries of the box will be marked separately to signal these points as marked but also to allow other meshes to include these points. The procedure is repeated recursively with the next finer refinement level for every refined mesh of $boxes_{out}$ (if the mesh has a refinement level of 2 or lower).

FindBoxesEXP(box box_{in}**, boxlist** $boxes_{out}$**)**
1. While not all important points inside box_{in} are covered by $boxes_{out}$
2. *fpoint* = Find point inside box_{in} which is not already stored
3. Set box_{new} to cell of *fpoint*
4. Call ExpandVolume(box_{in}, box_{new})
5. Add box_{new} to $boxes_{out}$
6. Clear points inside box_{new}
7. End While

ExpandVolume(box box_{in}**, box** box_{new}**)**
8. box_{best} = Expand Area of box_{new}
9. Repeat
10. $box_{new} = box_{best}$
11. box_{best} = Expand Volume box_{best} by one slice
12 box_{best} = Expand Area of box_{best}
13. Until density d of box_{best} < density-threshold

Table 4.3: Simplified version of our implementation of the *Growing Algorithm*.

To expand a volume, the area of the initial cell is first expanded in the plane perpendicular to the major axis alignment (see Section 4.2.1), and stored in box_{best} (Line 8). An area can only be expanded as long as it is inside box_{in} and does not contain cleared points (this avoids overlapping grids). The box gets expanded by one slice along the major axis alignment, in the direction where the new box has the highest volume density (Line 11). Afterwards, the area of the newly acquired slice is expanded again to include all adjacent important points in this slice (Line 12). These steps are repeated until the volume density of the expanded box is below a density threshold (Line 13).

The volume expansion can also be seen in the 2D example of Figure 4.4. In (a) an important point is found (point inside blue rectangle) and the red arrow shows its major axis alignment. In (b) the box (blue rectangle) is expanded so that all points in the row perpendicular to the major axis alignment are covered. In (c) the box is expanded one step along the major axis alignment. In the next step (d) the box is expanded so that all points of the newly acquired slice are covered. The same procedure is repeated from (e) to (h). The volume expansion is stopped after (h) because the volume density for the next bigger volume would be below a certain volume density threshold. A 3D example of the same procedure is shown in Figure 4.5. The blue point is the important point which is not covered. The area is expanded in the plane perpendicular to the major axis alignment (rectangle around blue point). All further steps are indicated by the dotted lines for the expanded volume.

4.3.2 Signature Algorithm

The *Signature Algorithm* was proposed by Berger and Rigoutsos [6] and is adopting signature-based methods used in computer vision and pattern recognition. It has become the standard approach to generate cells for an AMR because it is very efficient and fast. In Table 4.4 we provide a simplified version of our implementation of the algorithm.

First a few terms which are used in this section are described briefly. *cut-*

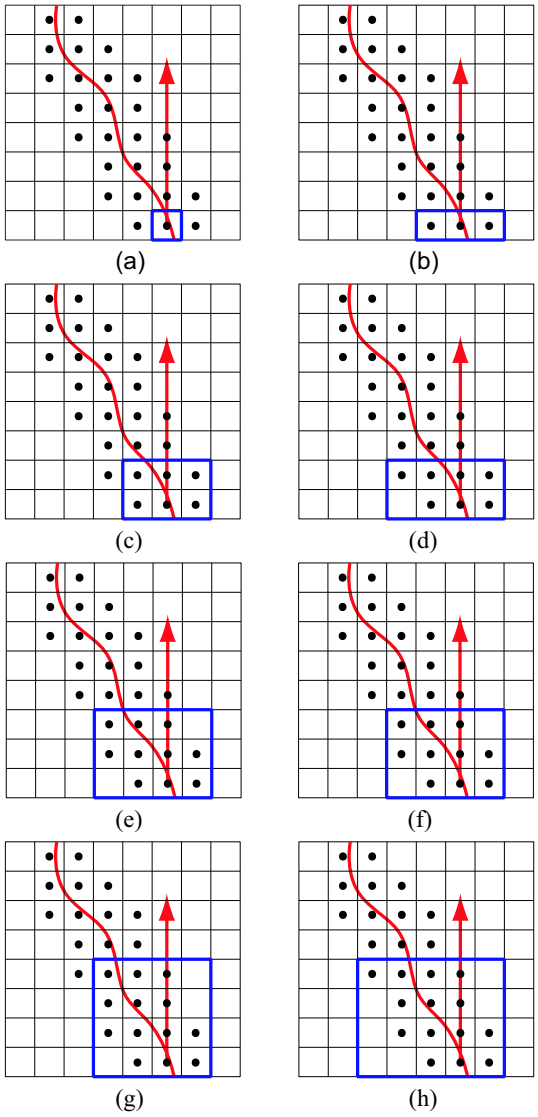


Figure 4.4: Basic steps of volume expansion: The blue box shows the current expanded volume. The red arrow indicates the major axis alignment of the centerline. (a) first an important point that is not covered by a box is selected. In (b), (d), (f), (h) the box is expanded perpendicular to the major axis alignment so that all important points of the current slice are covered. In (c), (e), (g) the box is expanded one step along the major axis alignment. The volume expansion is stopped after (h) because the density is below a threshold.

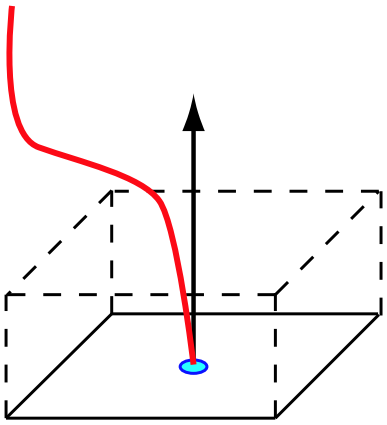


Figure 4.5: Volume expansion in 3D: First an important point is found at the blue circle. Then the area is expanded in the plane perpendicular to the axis alignment of the point, indicated by the arrow, to cover all important points in that plane. Finally the volume is expanded (along dotted lines) as long as the volume is over a certain density threshold.

point is used to store a point which defines the position of a cutting plane. The *tag histogram* contains the number of important points on all slices perpendicular to the points on every axis. For example, the entry for slice number i parallel to the yz plane is given by

$$S_{yz}(i) = \sum_{y=1}^{m} \sum_{z=1}^{n} I(i, y, z) \tag{4.8}$$

where m is the size of the box in y-direction and n is the size of the box in z-direction. Further the Laplacian second derivative

$$\Delta_{yz}(i) = -2 \cdot S_{yz}(i) + S_{yz}(i-1) + S_{yz}(i+1) \tag{4.9}$$

is calculated for every entry in the histogram.

An *inflection point* occurs at a sign change of two neighboring Δ entries. The biggest *inflection point* is the *inflection point* with the highest absolute difference between the two neighboring Δ entries.

Figure 4.6 shows an example for the *tag histogram*. The $\sum$ entry contains

the number of important points for every slice (in this 2D example a slice is either a row or column). The Δ entry contains the Laplacian second derivative. Each sign change of two neighboring entries of Δ indicate an *inflection point*. The biggest *inflection point* occurs between the 4^{th} and the 5^{th} column (the absolute difference between the two neighboring Δ entries is 8) and is taken as the splitting index (blue line).

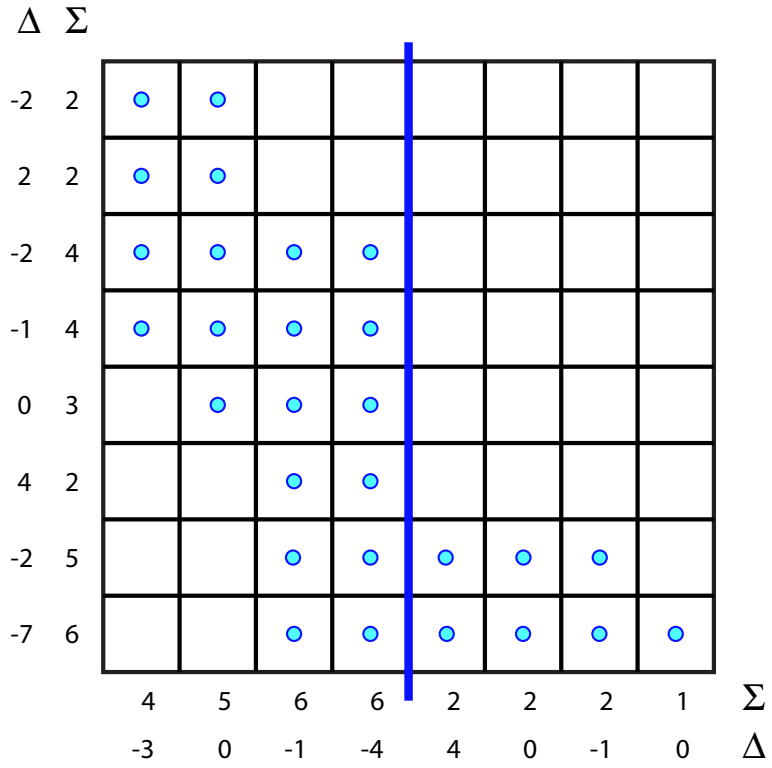


Figure 4.6: Histogram values: The Sum $\sum$ and the discrete Laplacian Δ are calculated for every slice. An *inflection point* occurs at a sign change in Δ. The biggest inflection point is taken as a splitting index (blue line).

The algorithm (see Table 4.4) starts at the coarsest refinement level and creates meshes so that all important points are covered. A bounding box of the whole volume is passed to the initial call of **FindBoxesBR** as box_{in}. The procedure generates a list of non-overlapping boxes ($boxes_{out}$) where each of the boxes has a density which is bigger than a certain density-threshold (Line 2) and all important points are covered by the boxes. First the bound-

FindBoxesBR(box box_{in}**, boxlist** $boxes_{out}$**)**
1. Set box_{new} = bounding box of box_{in}
2. If density d of box_{new} < density-threshold
3. Calculate tag histogram for each dimension in box_{new}
4. If ∃ *zero histogram value* in box_{new}
5. Set *cutpoint* to assoc. zero value cell index
6. Else
7. Set *cutpoint* to assoc. inflection cell index
8. End If
9. Split box_{new}into box_{left}, box_{right} at *cutpoint*
10. Call FindBoxesBR(box_{left}, $boxes_{out}$)
11. Call FindBoxesBR(box_{right}, $boxes_{out}$)
12. Else add box_{new} to $boxes_{out}$

Table 4.4: Simplified version of our implementation of the Berger-Rigoutsos Algorithm.

ing box of box_{in} is calculated and stored in box_{new} (Line 1). If this bounding box has a density over a certain density-threshold it will be added to the output list $boxes_{out}$ (Line 12). If the density of the box is not high enough a tag histogram will be computed (Line 3).

If the *tag histogram* contains an entry with a zero value (Line 4) the *cutpoint* will be set to the index of this entry (Line 5). If more than one zero value entry exists the *cutpoint* will be set to the entry with the highest minimum distance to the bounding box. If no such entry is found the *cutpoint* will be set to the biggest *inflection point* in the histogram (Line 7). Afterwards the current bounding box box_{new} will be split at the *cutpoint* into two boxes (Line 9). The function **FindBoxesBR** will be called for each of these two new boxes (Lines 10 and 11).

Once all boxes are generated the algorithm will be repeated for every box of the list $boxes_{out}$ with the next finer refinement level (if the current refinement level is greater than one).

Figure 4.7 shows one step of the algorithm. In (a) the initial box box_{in} is shown. In (b) the bounding box of the initial box is set (Line 1). Because the density value of this box is below a certain density threshold the box has

to be split. The histogram contains one zero value entry inside the bounding box so the *cutpoint* is set to this index (Line 5). In (c) the box is split into two new boxes at the *cutpoint* (Line 9).

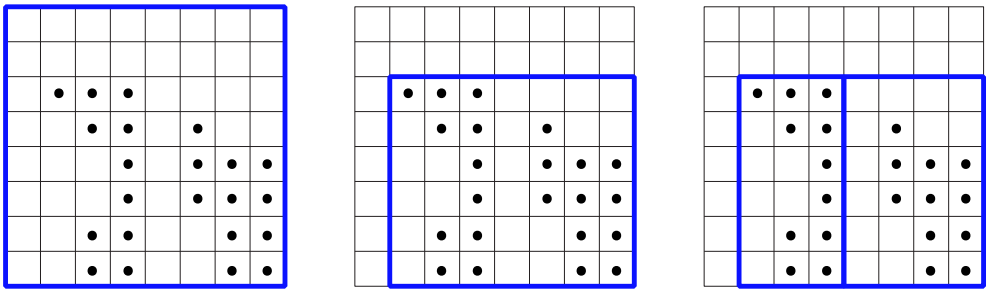


Figure 4.7: One step of the *Signature Algorithm*: (a) the blue box marks the initial box. (b) the bounding box is set. (c) the box is split at an empty slice into 2 new boxes.

4.3.3 Merging Meshes

After all meshes of one refinement level have been created both algorithms try to merge small meshes. This is to reduce the overhead of the meshes itself and of the sample points which are shared on common faces of neighboring meshes.

First all boxes are sorted depending on their volume size. Then the algorithm starts at the first (smallest) box and tries to merge it with any of the following boxes. Two boxes can be merged if they do not overlap with any other boxes of the same refinement level and if the volume density of the new volume is above a certain density threshold. The density threshold for merging two boxes should only be a little bit lower than the density threshold for the mesh creation. If the threshold is too low the combined meshes will have a low density and thus store many unimportant points. If the threshold is higher than the creation density threshold then no or very few meshes will be merged.

After the first box has been processed with all other boxes the same

procedure will be applied to the next box. The algorithm stops when all boxes have been processed.

4.3.4 KD-Tree

Each mesh contains pointers to all refined meshes and one pointer to its parent mesh if it exists. If a mesh contains many refined meshes a kd-tree (see Section 3.5) structure is build. Each node (see Table 4.5) contains a splitting plane which divides the meshes into a left and right subtree. Since the grids do not overlap it is always possible to find a splitting plane between two grids. A mesh which is cut by the splitting plane is inserted in both subtrees. The splitting plane is inserted where the minimum of the number of meshes which are completely to the left and to the right is the highest. This guarantees a well-balanced tree and allows to find a refined mesh in $O(logn)$ where n is the number of refined meshes.

field name	field type	description
split-value	integer	the splitting value
split	integer	the splitting dimension
left	kd-tree	a kd-tree representing those grids to the left of the splitting plane
right	kd-tree	a kd-tree representing those grids to the right of the splitting plane
mesh	mesh	a pointer to the mesh of this leaf-node

Table 4.5: Kd-tree node values for AMR

4.4 Visualization

The *Adaptive Meshes* can be visualized with many volume visualization techniques, such as CPR and DVR. All visualization techniques need to reconstruct a value at an arbitrary position. Therefore the meshes which contain the eight surrounding voxels of the current position have to be identified because each mesh only stores the points which are not stored already in

a coarser mesh. Afterwards these eight voxels have to be interpolated to reconstruct the value at the current position.

4.4.1 Finding the Current Mesh

To resample a point it is necessary to find the finest grid which is available for this position. The pseudocode of this algorithm is shown in Table 4.6.

First a coarse grid which contains the point has to be found. If the current grid of the last resampled position exists (Line 1) then the coarse grid will be searched starting at the last grid. As long as the point is not inside the last grid, the last grid is set to its parent grid (Line 2-4). If the grid of the last resampled position does not exist than the coarsest grid is used as the coarse grid (Line 6).

Starting at the coarse grid we search for the finest available grid inside this coarse grid (Line 8). If a grid has a kd-tree of its children (Line 11) then this tree is searched for a finer grid (Line 12). If a grid has no kd-tree then all children are tested sequentially (Line 17-21). If a child grid which contains the given point was found (Line 13 and 18 respectively) the function is called recursively to search for a finer grid (Line 14 and 19 respectively) inside this child grid. If no child grid contains the given point then the current mesh is returned (Line 23).

4.4.2 Resampling

To get the value at a certain position $(x_{cur}, y_{cur}, z_{cur})$ the eight closest voxels have to be found. Starting at the finest available mesh with the refinement level $reflevel$, the positions of the eight closest voxels are:

$p_0 = (x_{left}, y_{left}, z_{left})$
$p_1 = (x_{right}, y_{left}, z_{left})$
$p_2 = (x_{left}, y_{right}, z_{left})$
$p_3 = (x_{right}, y_{right}, z_{left})$
$p_4 = (x_{left}, y_{left}, z_{right})$
$p_5 = (x_{right}, y_{left}, z_{right})$
$p_6 = (x_{left}, y_{right}, z_{right})$

GetCurrentMesh(point p)
1. If ∃ $Mesh_{last}$
2. While point not inside $Mesh_{last}$
3. $Mesh_{last}$ = parent of $Mesh_{last}$
4. End While
5. Else
6. $Mesh_{last} = Mesh_{coarsest}$
7. End If
8. $Mesh_{last}$ = Call GetMesh(p) of $Mesh_{last}$
10. Return $Mesh_{last}$

GetMesh(point p)
11. If ∃ kd-tree
12. Set Mesh = Mesh of kd-tree leaf-node for point p
13. If p is inside Mesh
14. Return Call GetMesh(p) of Mesh
15. End If
16. Else
17. For all children
18. If p is inside of current child
19. Return Call GetMesh(p) of current child
20. End If
21. End For
22. End If
23. Return current mesh

Table 4.6: Simplified version of our algorithm for finding the finest available mesh for a given position.

$p_7 = (x_{right}, y_{right}, z_{right})$

where the left and right coordinates can be calculated as follows:

$x_{left} = \lfloor x_{cur}/reflevel \rfloor \cdot reflevel$
$x_{right} = x_{left} + reflevel$
$y_{left} = \lfloor y_{cur}/reflevel \rfloor \cdot reflevel$
$y_{right} = y_{left} + reflevel$
$z_{left} = \lfloor z_{cur}/reflevel \rfloor \cdot reflevel$

$z_{right} = z_{left} + reflevel$

A voxel (x, y, z) is stored in a mesh if the current mesh is the coarsest mesh or if $x \ mod \ plevel \ \neq \ 0 \vee y \ mod \ plevel \ \neq \ 0 \vee z \ mod \ plevel \ \neq \ 0$ where *plevel* is the refinement level of the parent mesh. Starting at the finest available mesh for the resampled position (see Section 4.4.1) all eight voxels are tested if they are stored in this mesh. The value of the voxels which are contained in this mesh are retrieved. As long as some voxel values are missing this procedure is repeated recursively for the parent mesh.

The value at the position $(x_{cur}, y_{cur}, z_{cur})$ can now be calculated as a trilinear interpolation of the eight closest voxels.

Chapter 5

Implementation

Creativity without implementation is irresponsibility.

Ted Leavitt

We implemented our prototype in C++. Abstract classes for the main objects were used which makes it easy to switch between different renderers and volume data structures. Microsoft Visual C++ 6.0 was used as the Integrated Development Environment (IDE). All graphical operations which are needed for the rendering process are performed with OpenGL 1.2. The non-commerical version 3.2.1 of Qt was used for the graphical user interface.

5.1 Class Overview

Figure 5.1 shows an overview of the most important classes, their relations and their most important member functions. Only the function parameters that are needed for the description of the functions are shown. Table 5.1 contains a description of the symbols used in the class overview.

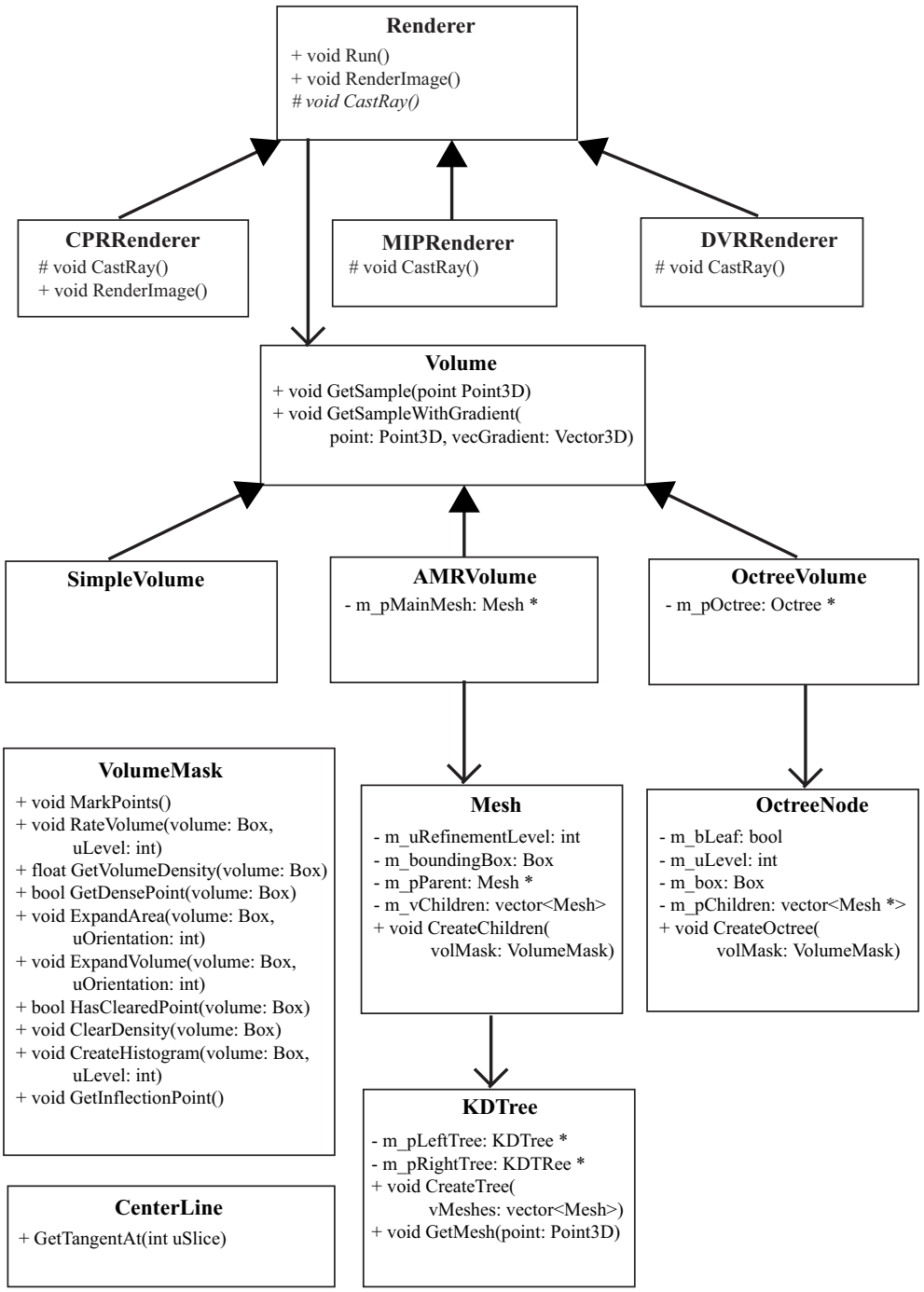


Figure 5.1: Class overview showing the most important classes.

symbols	description
+	public member
#	protected member
-	private member
Italic	virtual function

Table 5.1: Description of different symbols in class overview

5.2 Class Description

This section describes the classes shown in Figure 5.1 and its most important methods.

5.2.1 Renderer

Renderer is an abstract class to render all available volumes. This class is derived by any Direct Volume Renderer to generate images of the volumes.

Run

Run starts the rendering process for every available volume. In our application it renders the image from the original volume and the data structure based on Adaptive Mesh Refinement. When both images are rendered a difference image is computed.

RenderImage

RenderImage performs ray-casting to generate a DVR image by calling the virtual method *CastRay* for every single ray. The output image depends on the current viewport parameters such as position, zoom and re-sample distance.

5.2.2 CPRRenderer

The *CPRRenderer* class implements a Curved Planar Reformation renderer. It overrides the method *RenderImage* because no ray-casting is needed.

5.2.3 MIPRenderer

MIPRenderer is a renderer implementation which overrides the method *CastRay* to generate an DVR image with Maximum Intensity Projection.

5.2.4 DVRRenderer

DVRRenderer is a renderer implementation which overrides the method *CastRay* to generate an DVR image with alpha compositing. It takes a transfer function into account and evaluates the rendering equation using the gradients of the sample points. The gradients of the Adaptive Mesh Refinement are precalculated. Thus, the resampling is very fast because the gradient can be calculated with trilinear interpolation of the eight nearest gradients. The gradients for rendering the whole volume have to be calculated on-the-fly because storing precalculated gradients for the whole volume requires too much memory.

5.2.5 Volume

Volume is an abstract class that provides access to the volume data. The data can be accessed by the following member functions which have to be overridden by a specific volume implementation:

GetSample

GetSample returns the CT-data value at the given position.

GetSampleWithGradient

GetSampleWithGradient returns the CT-data value and the gradient value at the given position.

5.2.6 SimpleVolume

SimpleVolume is a volume implementation which stores the whole volume in a linear array in x-y-z order.

5.2.7 AMRVolume

AMRVolume stores only the important regions of the volume in an Adaptive Mesh Refinement as described in Chapter 4. The member *m_pMainMesh* is a pointer to a Mesh with the coarsest resolution. It is the bounding box of all important points and contains the pointers to the finer meshes. The gradients are precalculated for every sample point. It is possible to also store precomputed gradients with the data values as the AMR representation is usually very small.

5.2.8 Mesh

Mesh represents one grid of the Adaptive Mesh Refinement. It stores the actual data values with its gradients, the bounding box of the corresponding region, the refinement level, and pointers to its parent and children meshes.

CreateChildren

CreateChildren creates the AMR with either the signature or growing algorithm.

5.2.9 KDTree

KDTree is a kd-tree data structure to store a list of meshes in an efficient way. Each kd-tree element contains a splitting plane and two pointers *m_pLeftTree* and *m_pRightTree* to its left and right trees. If a kd-tree element is a leaf node it contains a pointer to a mesh.

CreateTree

CreateTree creates the kd-tree data structure for the given meshes. It creates a balanced tree.

GetMesh

GetMesh finds the Mesh which stores the given point. If no such point exists no Mesh is returned.

5.2.10 OctreeVolume

OctreeVolume is a basic octree implementation of the volume. The octree is subdivided until an octree element is homogenous (this means the octree element is either empty or completely filled with important points) or the octree element is below a certain size. This is to avoid too small octree elements and reduce the overhead produced by each octree element.

5.2.11 OctreeNode

OctreeNode represents one octree element of the octree data structure. It stores the corresponding bounding box, the current subdivision level, a flag if this element contains data and pointers to its children elements. Only a leaf element of the octree volume contains data points while the intermediate elements only contain pointers to its children.

CreateOctree

CreateOctree creates the octree data structure so that all important points are covered by an octree element.

5.2.12 VolumeMask

VolumeMask is a helper class used to generate the meshes for the signature and growing algorithm.

MarkPoints

MarkPoints marks all points of the volume according to the distance to the centerline. Furthermore, the major axis alignment, which is the maximum component of the tangent of the centerline, is set for all important points.

RateVolume

RateVolume calculates the number of important points with an importance higher than the given importance. Each point contains the number of important points from the origin to the point as described in Section 4.2.2.

GetVolumeDensity

GetVolumeDensity returns the ratio between the number of important points and the size of the given volume. This ratio is used for both mesh generation algorithms.

GetDensePoint

GetDensePoint finds an important point that is not covered by a mesh yet and stores it in the parameter *point*. If no such point exits the method returns false, otherwise it returns true.

ExpandArea

ExpandArea expands the given volume in the plane perpendicular to the major axis alignment so that all adjacent important points in that plane are covered by the volume.

ExpandVolume

ExpandVolume expands the given volume along the major axis alignment step by step until the volume density falls below a certain threshold. The volume can only be expanded if it does not overlap with any existing box. For each newly added slice the volume is expanded with the method *ExpandArea*.

HasClearedPoint

HasClearedPoint returns true if the given volume has a point that is already stored in an existing mesh. This method is used to avoid storing points in multiple meshes because meshes are not allowed to overlap.

ClearDensity

ClearDensity marks all point of the given volume as cleared so that they are not stored in any other meshes. Points at the border of the volume are marked differently since neighbor meshes can share points at their common faces.

CreateHistogram

CreateHistogram creates a histogram which contains the sum and discrete Laplacian derivative for every slice of the given volume. This method is used for the signature algorithm.

GetInflectionPoint

GetInflectionPoint returns the inflection point (see Section 4.3.2) of the previous calculated histogram.

5.2.13 CenterLine

CenterLine maintains the storage of a cubic B-spline by storing the control points. It contains a control point with the radius of the aorta for every slice of the volume.

GetTangentAt

GetTangentAt returns the tangent of the b-spline for the given z-position of the volume. This is needed for marking the data when the major axis alignment of the samples is set.

Chapter 6

Results

You always succeed in producing a result.

Anthony Robbins

6.1 Dataset

For testing purposes we used a CTA dataset with a resolution of 512x512x264. The coarsest mesh of our AMR data structure has a refinement level of four. Therefore we extended the dataset to the size of 513x513x265 so that the coarsest mesh can be aligned to the volume boundary. Since every sample value needs two bytes the total size of the volume is 139.5 MB.

Furthermore, the centerline of the aorta was given as a center point and the corresponding radius for each slice.

6.2 Mesh Density

We examined different density threshold values to find the best settings for creation of the adaptive mesh structure with respect to the memory consumption. Usually a higher volume density (i.e., higher percentage of important points inside a mesh) results in less memory consumption and more meshes.

More meshes are generated because the meshes are smaller to achieve the desired volume density. This also means more total overhead since every mesh has to store internal information (bounding box, pointers to children, etc.) and neighboring meshes share the points at their common faces. Thus it is desirable to keep the total number of meshes small.

The points inside the radius to the centerline are marked with high importance (they need to be stored in full resolution) and the points inside two times the radius to the centerline are marked with medium importance (they need to be stored at least in half resolution). As a result, the theoretical minimum size of the volume which only includes the marked data points is 246 KB. This is because every point with high importance needs 2 bytes (for the CT data) and every point with medium importance needs 1/4 byte on average (because only every second point is stored in each axial direction).

Especially higher volume density thresholds result in many small meshes which generate a lot of overhead. We tried to reduce the number of small meshes by combining the smallest grids as described in Section 4.3.3.

6.2.1 Mesh Density with Growing Algorithm

Figure 6.1 shows the memory consumption of the adaptive meshes for different density thresholds with the *Growing Algorithm* (Section 4.3.1). The lines in this Figure show the volume size for different volume density settings. For the red line no mesh combination was used. For the green line the meshes were combined with the same volume density threshold which was used for the creation of the meshes. For the blue line the meshes were combined with a volume density threshold that is 10 % lower than the volume density for the mesh creation (for example, if a volume density threshold of 50 % is used, the meshes were combined with a volume density threshold of 40 %). For a volume density higher than 60% the mesh overhead gets bigger and bigger and thus prevents any further reduction of the memory requirements. It can also be seen that merging small meshes does not lead to the desired memory reduction. Only for a high density the combined meshes sometimes result in little less memory usage although this difference is hardly existent.

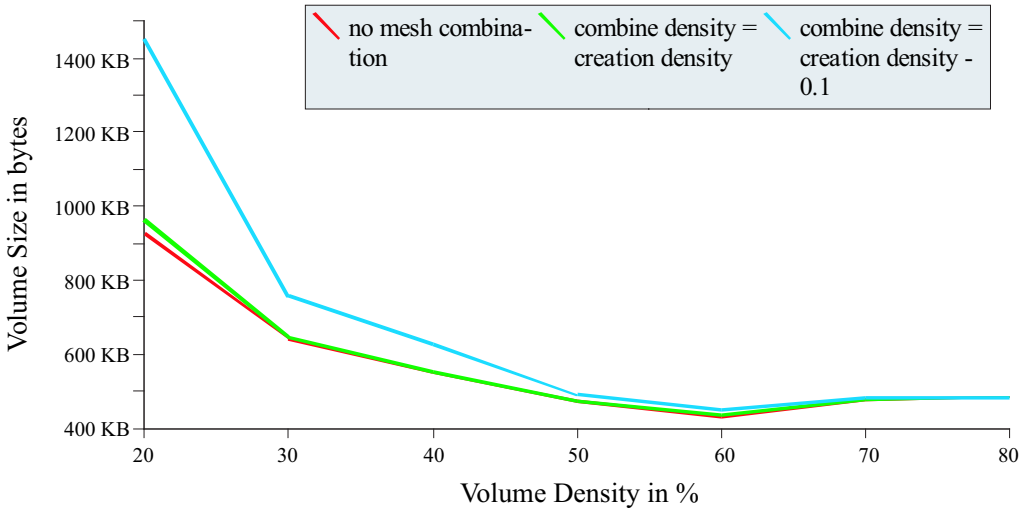


Figure 6.1: Volume size with different density settings for the *Growing Algorithm*.

The best result was achieved with 60% volume density and no mesh combination. The memory size of this AMR structure is only 432 KB. 35.6 KB of the 432 KB are used for the data structure itself (pointers and bounding box), leaving 396.4 KB memory space for the samples.

6.2.2 Mesh Density with Signature Algorithm

In Figure 6.2 the memory consumption of the adaptive meshes for different density thresholds with the *Signature Algorithm* (Section 4.3.2) has been analyzed. Unlike with the *Growing Algorithm*, merging small meshes can reduce the memory usage for a volume density with 60 % and higher. This is because the *Signature Algorithm* produces much more meshes (see Figure 6.3) and thus much more overhead as a consequence which can be reduced by merging meshes.

The best result was achieved with 50% volume density and no mesh combination. For this AMR structure 570 KB are needed. For a higher volume density the overhead of the huge number of grids gets too big to allow any further memory reduction.

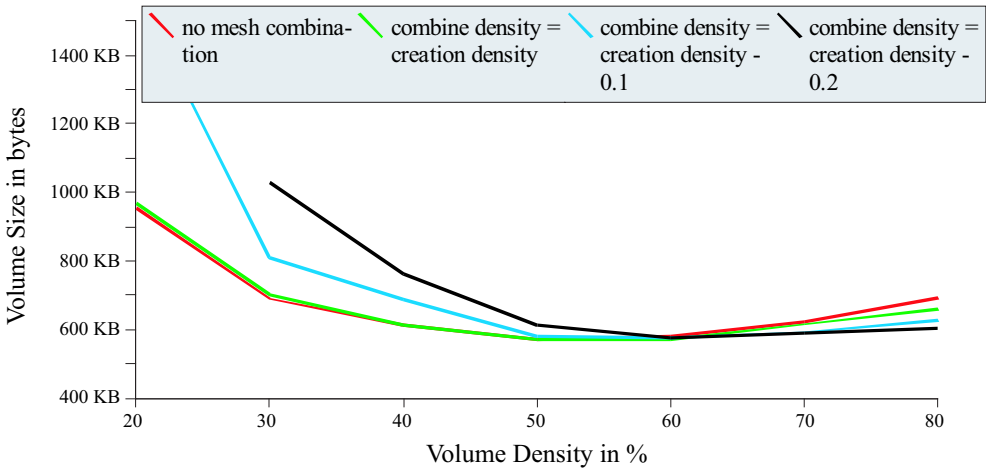


Figure 6.2: Volume size with different density settings for the *Signature Algorithm*.

6.2.3 Algorithm Comparison

The *Signature Algorithm* generates much more meshes than the *Growing Algorithm* (see Figure 6.3 and 6.4). The *Growing Algorithm* takes advantage of the knowledge of the underlying data while the *Signature Algorithm* is a general approach for AMR and needs more meshes to achieve the same volume density.

As can be seen in Figure 6.5 the *Growing Algorithm* even stores more points in full resolution which were marked for half resolution than the *Signature Algorithm*. So the *Growing Algorithm* can store more points while keeping the total volume size smaller because it generates fewer meshes.

6.3 Image Quality

We compared the image quality of a CPR image rendered from the AMR data structure to the same image rendered from the whole volume (i.e., all sample values). The difference images in Figure 6.6 - Figure 6.9 show the difference of these two images. A white pixel means that this pixel is completely equal in both images. As a pixel in the difference image gets darker the difference

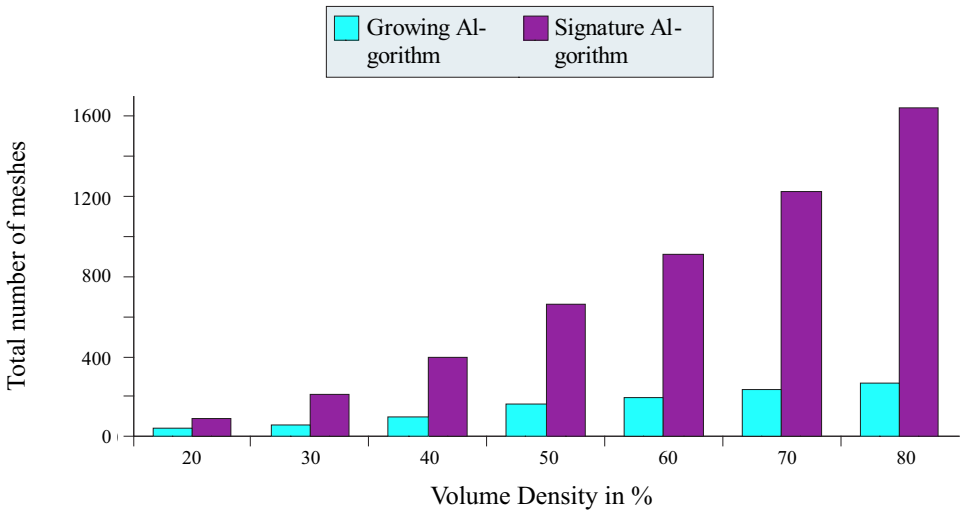


Figure 6.3: Total number of meshes for *Growing Algorithm* and *Signature Algorithm*.

between the rendered images grows. We used different settings for marking the data points. The points are marked depending on their distance to the centerline (see Section 4.2.1).

In Figure 6.6 the adaptive meshes store all points inside the radius to the centerline in full resolution and all points inside two times the radius to the centerline at least in half resolution. The remaining points will be stored at least in a low resolution (every 4^{th} point in each axis). As can be seen a lower resolution leads to more artifacts. Obviously points inside meshes with full resolution are resampled without artifacts. Figure 6.8 and Figure 6.9 do not store any points in low resolution.

The strong artifacts on the upper and right side of the difference images are due to the fact that the volume is enlarged for the AMR structure. Points sampled at these borders are interpolated in the adaptive Meshes while the same points are outside of the original volume. This effect is not crucial since it only occurs at the boundaries of the volume.

Furthermore, strong artifacts occur at the boundaries of certain objects which have a different data value range than its neighborhood. Since samples are missing, the interpolation is visible due to the rapid change of the data

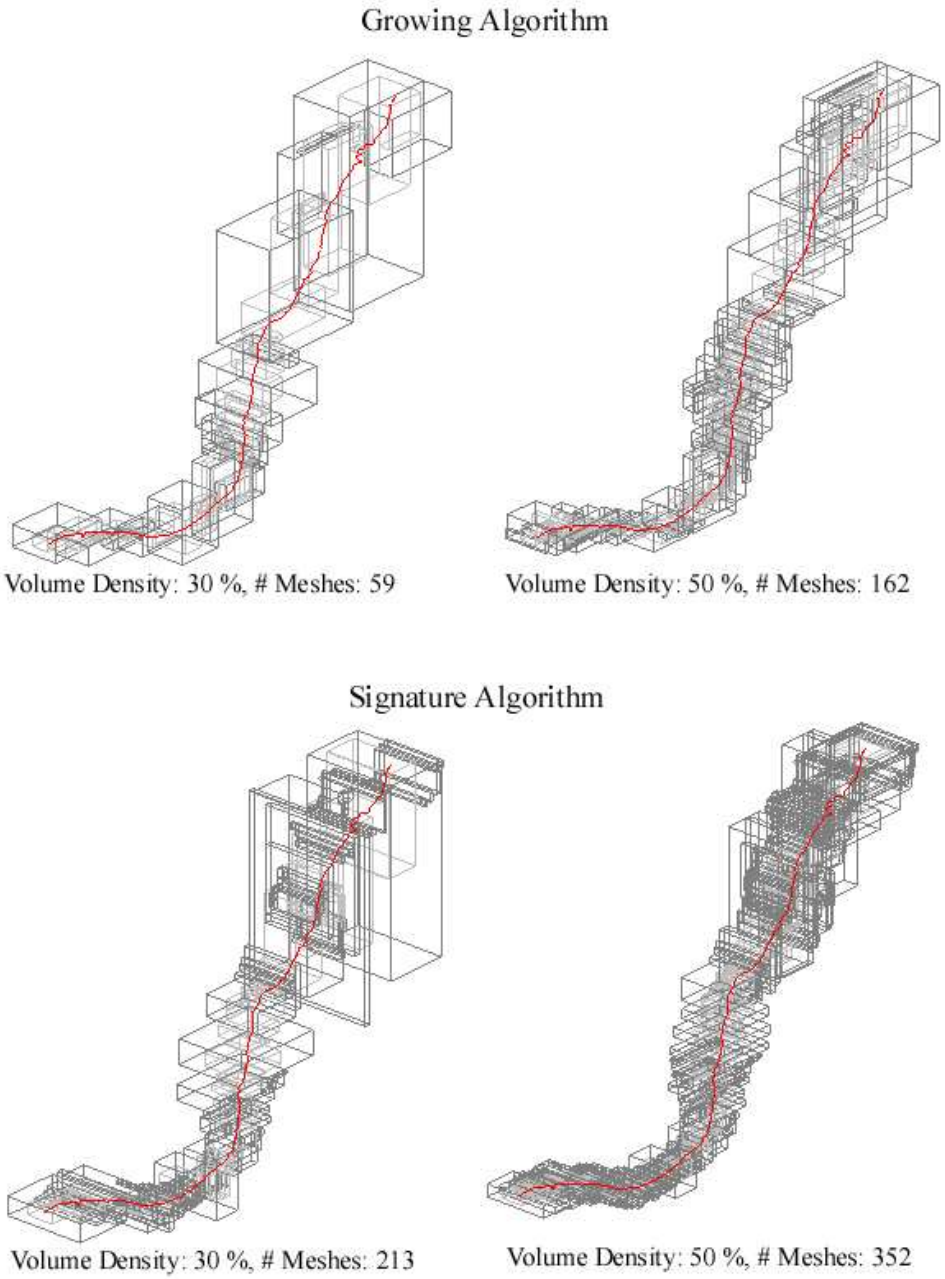


Figure 6.4: AMR structure for different density settings with *Growing Algorithm* and *Signature Algorithm*.

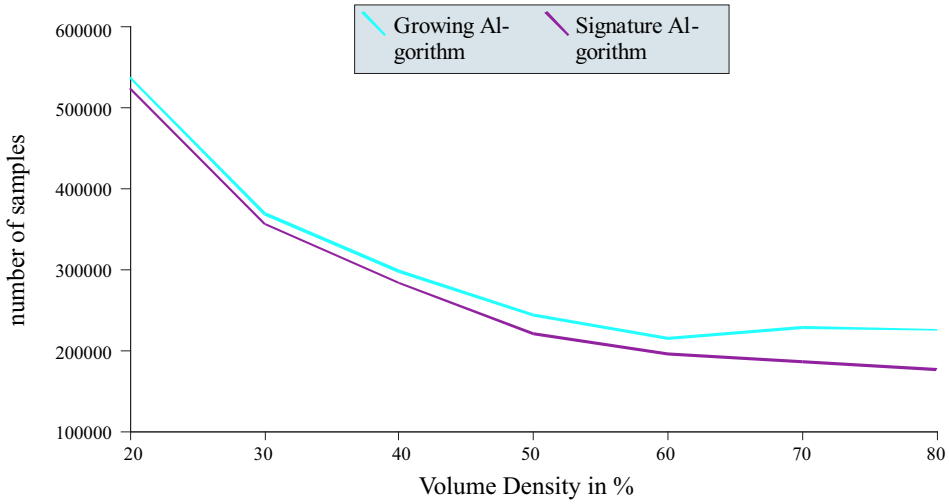


Figure 6.5: Total number of points stored in high resolution although they were marked with medium importance.

values. Except those objects very few artifacts are visible. Regions which are stored in full resolution have no artifacts as can be seen in Figure 6.9 where all points close to the centerline are stored in full resolution.

6.4 Data Structure Comparison

The total size of our dataset is 139.5 MB. For comparison with other data structures the points have been marked as described in Section 6.2. The easiest way to reduce the volume size is to use one bounding box to include all important samples which results in a total size of 16 MB.

We also implemented an Octree data structure where each node is subdivided if it is not homogeneous (a node is homogeneous if it only contains important samples or is empty). As in the adaptive meshes, neighboring nodes share sample points at their common faces. This produces overhead and makes it necessary to stop the subdivision of an octree node at a certain subdivision level. The smallest size of this data structure was 1.5 MB with a maximum of eight levels.

With our AMR implementation the size could be reduced to 432 KB

which is only about 0.3 % of the original volume size (see Table 6.1 for a comparison of the data structures).

data structure	memory space
Mesh (total volume)	139.5 MB
Mesh (bounding box of aorta)	16 MB
Octree	1.5 MB
AMR	432 KB

Table 6.1: Memory consumption of CTA-dataset for several data structures.

Kähler et al [18] have also implemented an AMR which uses a *Signature Algorithm* to generate the meshes. They optimized their data structure for speed while our data structure is optimized for memory size. They store the samples in textures which results in a loss of memory usage because the texture size has to be a power of 2 in modern graphic cards. Sample points are only stored in leaf nodes of the AMR hierarchy while our data structure allows storage of samples in intermediate nodes. We can store areas in different resolutions without producing any overhead for the data because each grid only stores samples which are not already contained in coarser grids.

6.5 Performance

With the *Growing Algorithm* the creation time of the AMR data structure is almost constant for any volume density. With the *Signature Algorithm* the creation time increases with a higher volume density threshold (or more meshes respectively). This is due to the fact that the *tag histogram* (see Section 4.3.2) is calculated for every mesh. Table 6.2 shows the creation times on a Pentium 4 1800MHz with 512MB of RAM.

The rendering time for CPR, MIP, or DVR with the AMR data structure mainly depends on the size of the data structure. The more samples are stored the more points have to be resampled which is the most costly operation of the visualization process. See Table 6.3 for the specific render-

volume density	*Growing Algorithm*	*Signature Algorithm*
10 %	1.7 sec	6.6 sec
20 %	1.5 sec	6.9 sec
30 %	1.3 sec	7.1 sec
40 %	1.3 sec	7.5 sec
50 %	1.3 sec	8.2 sec
60 %	1.3 sec	9.9 sec
70 %	1.3 sec	12.2 sec
80 %	1.3 sec	13.4 sec

Table 6.2: Creation time of the AMR data structure with the *Growing Algorithm* and the *Signature Algorithm* on a Pentium 4 1800MHz with 512MB of RAM.

ing times of a MIP on a Pentium 4 1800MHz with 512MB of RAM. The rendering time for the MIP with the whole volume takes 13 seconds.

volume density	*Growing Algorithm*	*Signature Algorithm*
10 %	8.8 sec	6.0 sec
20 %	7.2 sec	4.5 sec
30 %	3.6 sec	3.5 sec
40 %	2.9 sec	3.1 sec
50 %	2.7 sec	2.7 sec
60 %	2.6 sec	2.7 sec
70 %	2.7 sec	2.5 sec
80 %	2.6 sec	2.5 sec

Table 6.3: Rendering time of a MIP with AMR data structure with the *Growing Algorithm* and the *Signature Algorithm* on a Pentium 4 1800MHz with 512MB of RAM.

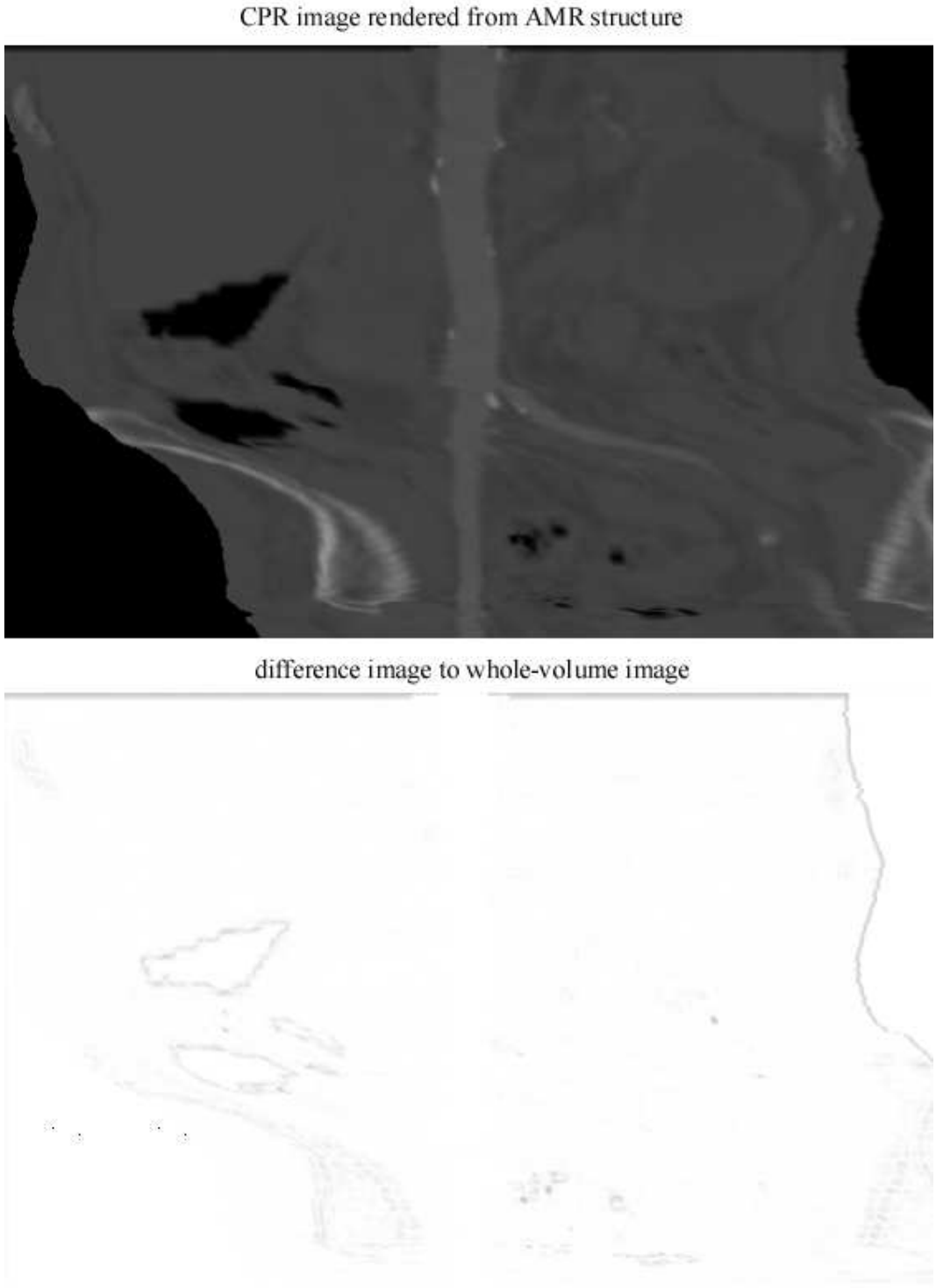


Figure 6.6: Points inside the radius to the centerline are stored in full resolution, points inside two times the radius to the centerline are stored at least in half resolution. Remaining points are stored at least in low resolution.

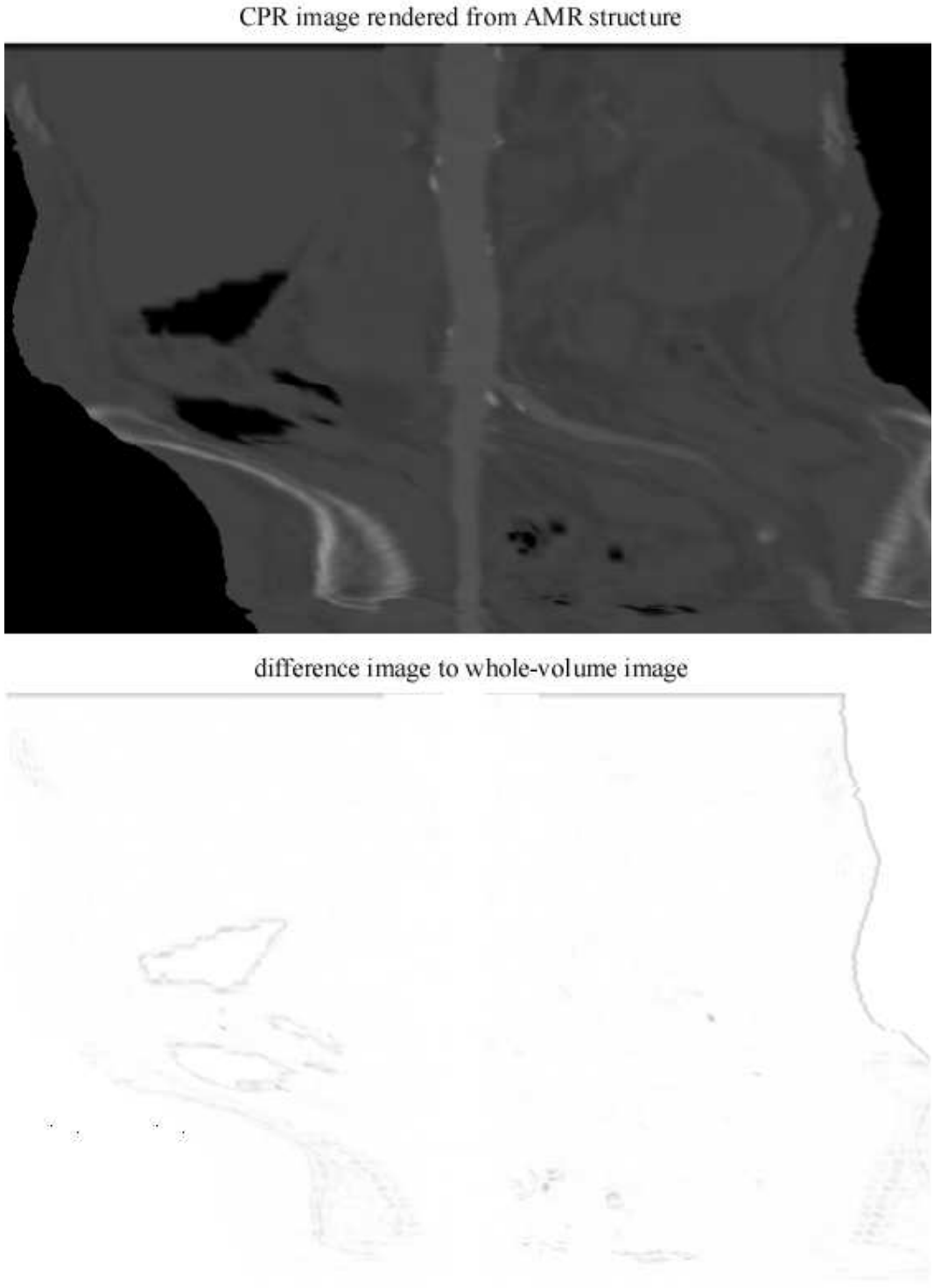


Figure 6.7: Points inside two times the radius to the centerline are stored in full resolution, points inside four times the radius to the centerline are stored at least in half resolution. Remaining points are stored at least in low resolution.

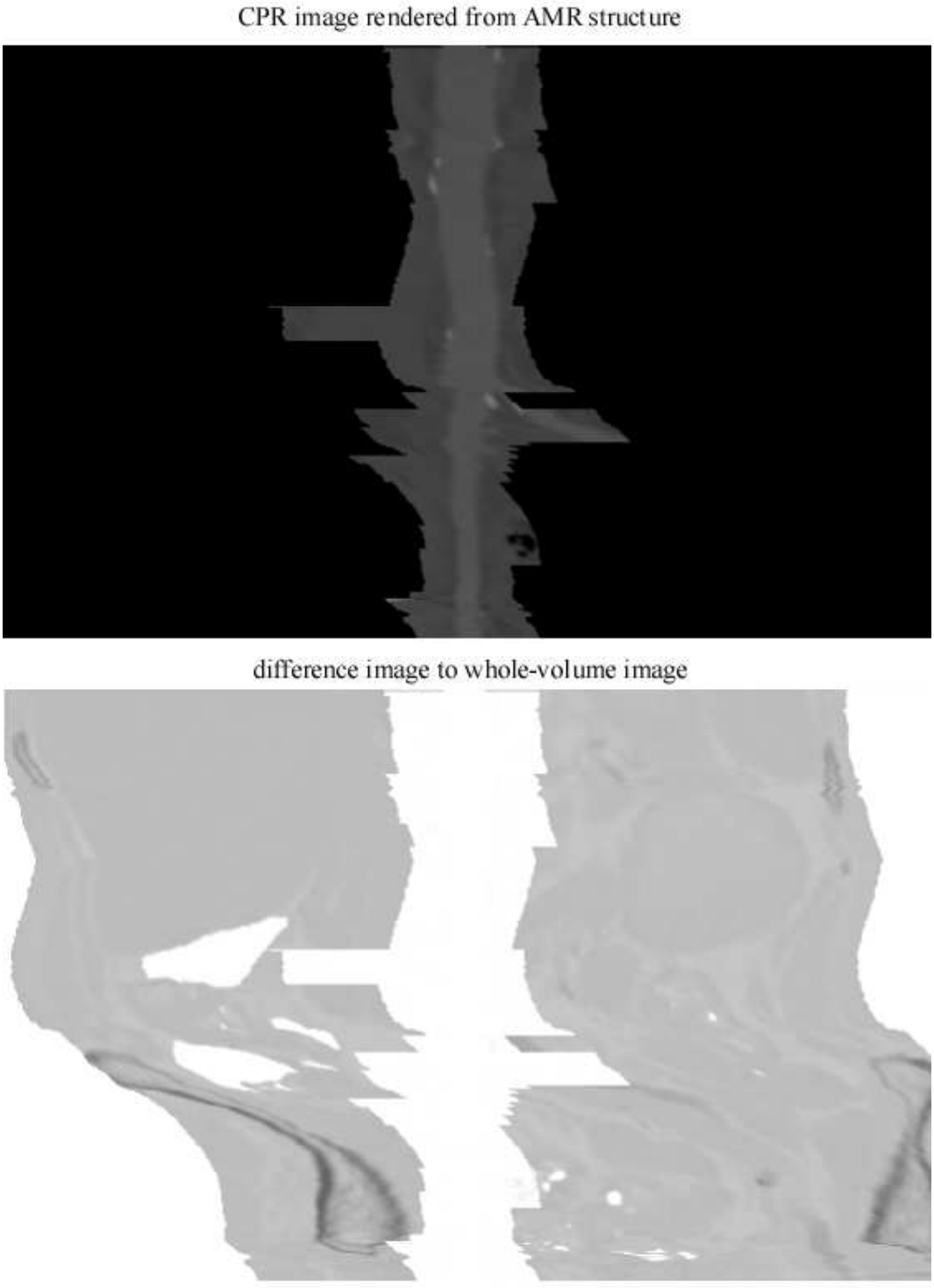


Figure 6.8: Points inside the radius to the centerline are stored in full resolution, points inside four times the radius to the centerline are stored at least in half resolution.

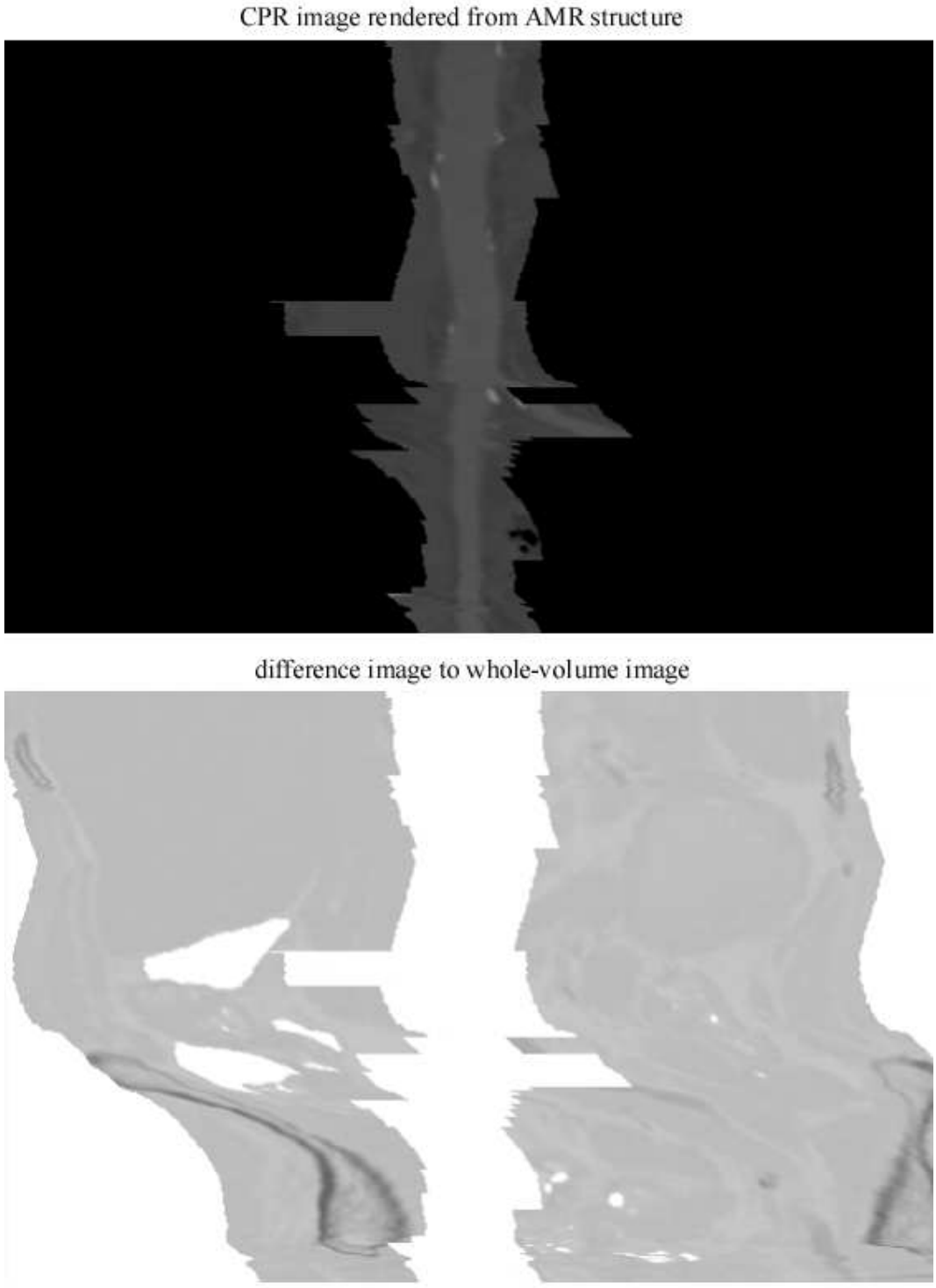


Figure 6.9: Points inside four times the radius to the centerline are stored in full resolution.

Chapter 7

Summary

Anyone who has never made a mistake has never tried anything new.

Albert Einstein

7.1 Introduction

The handling of very large datasets is a very important topic in volume visualization. Especially in medical image processing Computed Tomography (CT) scanning devices can create huge datasets. A CT scanner is a device which generates a series of two-dimensional X-ray images which show the internals of an object. These X-ray images are generated using an X-ray source that rotates around the object. Several scans are progressively taken while the object is passed through the scanning device. New technology multi-slice CT scanners produce even higher resolution and more slices than conventional scanners.

Angiography is a medical imaging technique that uses X-rays to visualize blood filled structures, such as arteries, veins, and heart chambers. Because blood has the same radiodensity as the surrounding tissues, a radiocontrast agent (which absorbs X-rays) is injected to highlight vessels, enabling an-

giography. Angiography is commonly performed to identify vessel narrowing and calcifications.

To investigate the arteries a sequence of 300 - 1500 (depending on the region of interest) CT images are necessary. The acquisition of these CT images usually takes 30 to 60 seconds. Due to the large amount of slices, 2D examination is a rather tedious and time consuming task. Therefore, radiologists usually use the following two visualization techniques: Maximum Intensity Projection and Curved Planar Reformation. With these visualization methods the investigation of the arteries usually only takes a few minutes.

The large amount of data, needed for CTA, presents a challenge for current PC hardware. Therefore, it is very important to design memory efficient data structures in order to handle these large datasets. These data structures should allow to leverage resources where they are really needed. For example, in case of CTA only the aorta is of main interest. The surrounding information is only needed as context information or not needed at all. This part of the data, therefore, needs to be stored in a lower resolution or not at all. Such a data structure, which allows to efficiently leverage resources, enables that the data can easily be kept in main memory. Thus, efficient processing of the data is possible.

7.2 Data Structures

A simple data structure like a *Mesh* is not sufficient since it cannot skip non-important regions. Point based structures such as a kd-tree are only useful if the important data is very sparse. Otherwise the storage of the position for every sample takes too much space.

An *Octree* is a good choice because the data can be accessed very fast and octree nodes can be created only where they are needed. Its disadvantage is that the memory consumption mainly depends on the spatial distribution of the objects of interest. An *Adaptive Mesh* is a *Mesh* which can adapt its boundaries and store the data in a coarser resolution.

We implemented an *Adaptive Mesh Refinement* (AMR) which is able to

store only the objects of interest of the dataset and to keep additional data available in a coarser resolution. This enables the possibility of storing huge datasets by creating meshes only where they are needed, thus discarding non-important samples.

7.3 Adaptive Meshes

Our AMR data structure operates on a CTA dataset. The dataset contains the segmentation information for the aorta. The aorta is characterized by a centerline and a radius for the control points of the centerline. The radius defines the thickness of the aorta at a given point. The algorithms in the following sections can be applied to any dataset where an object of interest is described by a centerline.

For simplicity reasons the following descriptions are only for two dimensions. Nevertheless they can be easily extended to 3D. We implemented the AMR in a way that the resolution of each mesh has to be a power of two. A resolution of n means that only every n^{th} sample in each axial direction is stored. Our *Adaptive Meshes* are stored in a grid-aligned format, thus neighboring grids share data values at their common faces. The coarsest grid stores every point according to its resolution. Finer grids only store points that are not already stored in any coarser grid. As can be seen in Figure 7.1, the outer grid is the coarsest grid of the AMR structure with a cell width of two and stores all samples according to its resolution (green points). The inner grid (orange box) has a cell width of one but stores only samples which are not already stored in the outer grid (red points). This offers the possibility to store multiple resolution levels without generating any overhead for the data.

The sample points of each grid are stored in a one-dimensional array. Each grid is defined by the two points p_{min} (origin of the grid), p_{max} (endpoint of the grid) and the refinement level $reflevel$. $reflevel$ defines the cell width for all axial directions. $plevel$ is the refinement level of the parent mesh ($plevel = reflevel \cdot 2$). The relative position p_{rel} inside a grid of an absolute point p_{abs} is calculated as

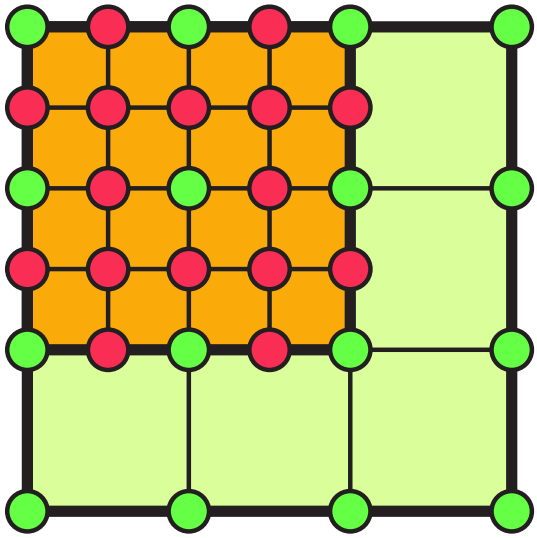


Figure 7.1: AMR implementation: green points are stored in the outer (coarser) mesh, red points are stored in the inner (finer) mesh

$$p_{rel} = (p_{abs} - p_{min})/reflevel \tag{7.1}$$

For the coarsest grid level the address of the relative position in the data array is simply given as

$$addr = p_{rel,y} \cdot ((p_{max,x} - p_{min,x})/reflevel) + p_{rel,x} \tag{7.2}$$

For a refined grid the address of a position in the one-dimensional array is

$$addr = \lfloor((p_{rel,y} + 1)/2)\rfloor \cdot s_1 + \lfloor(p_{rel,y}/2)\rfloor \cdot s_2 + \lfloor p_{rel,x}/(p_{max,x} - p_{min,x}) \cdot s_{sy}\rfloor \tag{7.3}$$

where s_1 is the number of points of every other slice starting at the first slice, s_2 is the number of points of every other slice starting at the second slice and the index sy in s_{sy} is calculated as $sy = p_{rel,y}\ mod\ 2 + 1$. The computations of s_1 and s_2 are shown in Table 7.1 and Table 7.2 respectively.

case	s1
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$\lfloor p_{max,x} - p_{min,x}/plevel \rfloor$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$\lfloor (p_{max,x} - p_{min,x} + 1)/plevel \rfloor$
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$

Table 7.1: First slice size for different grid alignments

case	s2
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel = 0$	$(p_{max,x} - p_{min,x})/reflevel\ +\ 1$
$p_{min,x}\ mod\ plevel = 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$\lfloor p_{max,x} - p_{min,x}/plevel \rfloor$
$p_{min,x}\ mod\ plevel \neq 0 \wedge$ $p_{min,y}\ mod\ plevel \neq 0$	$\lfloor (p_{max,x} - p_{min,x} + 1)/plevel \rfloor$

Table 7.2: Second slice size for different grid alignments

7.3.1 Marking Data

We introduce three different types of importance: high importance, medium importance and low importance. Areas marked with high importance need to be stored in a grid with a cell width of one which corresponds to the highest possible resolution. Areas marked with medium importance need to be stored in a grid with a cell width of at least two. Finally, areas marked with low importance need to be stored in a grid with a least a cell width of four.

First of all the data has to be categorized so that each sample point has a specific importance. According to the importance an appropriate grid resolution is chosen. All points that are close to the centerline (path of the object of interest) are very important which means that these points must

be stored in full resolution in the *Adaptive Meshes*. The points which are farther away but still within a certain range to the centerline have medium importance, therefore these points are stored at least in half resolution (every second point in each axis has to be stored).

Figure 7.2 illustrates how the samples are categorized. Thick points are marked as very important because they are close to the centerline (red dashed line). Thin points are marked with medium importance because they are farther away but still within a certain range to the centerline. All remaining points are not marked at all. Alternatively they could be marked with low importance to keep them in the lowest available resolution.

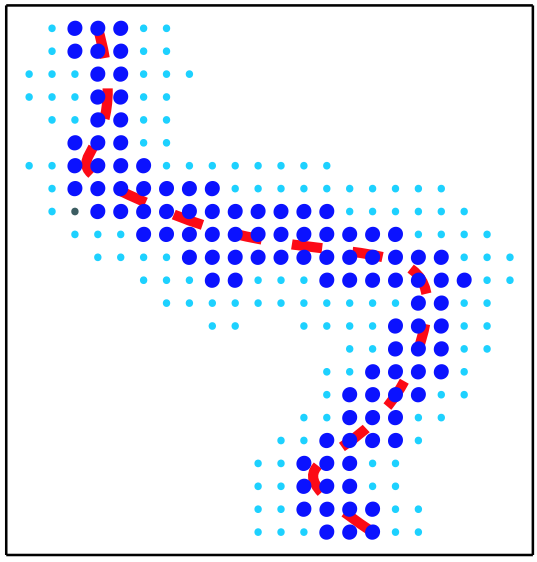


Figure 7.2: Marked data: importance of samples is set according to distance to the centerline (red). Thick points will be stored in full resolution, thin points will be stored in half resolution or higher.

Furthermore each sample point is assigned the major axis alignment of the tangent at the corresponding point on the centerline. The major axis alignment is the component of the tangent vector with the highest magnitude. The major axis alignment is needed for the *Growing Algorithm* (see Section 7.3.3).

7.3.2 Calculation of Volume Density

In this section we explain how the volume density d is calculated. The volume density defines the percentage of important points in a volume. It is needed for the *Growing Algorithm* and the *Signature Algorithm* to determine if a volume requires further subdivision and can be expanded further respectively. For easier understanding the following explanations are made for areas instead of volumes. Furthermore, the volume density refers to a specific importance level. The important points are all points which have the specified importance or higher. For example, if the importance level is medium importance then all points with medium or high importance are called important points.

In order to quickly determine the number of important points for any given area a summed area table is build. We refer to an entry of the summed area table as $D(x, y)$. The summed area table contains the number of important points from $(0, 0)$ to any point (x, y) of the whole area.

$$D_{(x,y)} = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j) \tag{7.4}$$

where $I_{(x,y)}$ returns 1 if the point at the position (x,y) is important and 0 otherwise.

The summed area table is calculated with dynamic programming with the following formula:

$$D_{(x,y)} = I(x, y) + D_{(x-1,y)} + D_{(x,y-1)} - D_{(x-1,y-1)} \tag{7.5}$$

The number of important points for the boundaries with $x = 0$ or $y = 0$ have to be calculated separately before.

The number of important points for any area from p_{min} to p_{max}, where the entry of the summed area table for each corner is given by D_i (see Figure 7.3), can be calculated as

$$D = D_3 - D_2 - D_1 + D_0 \tag{7.6}$$

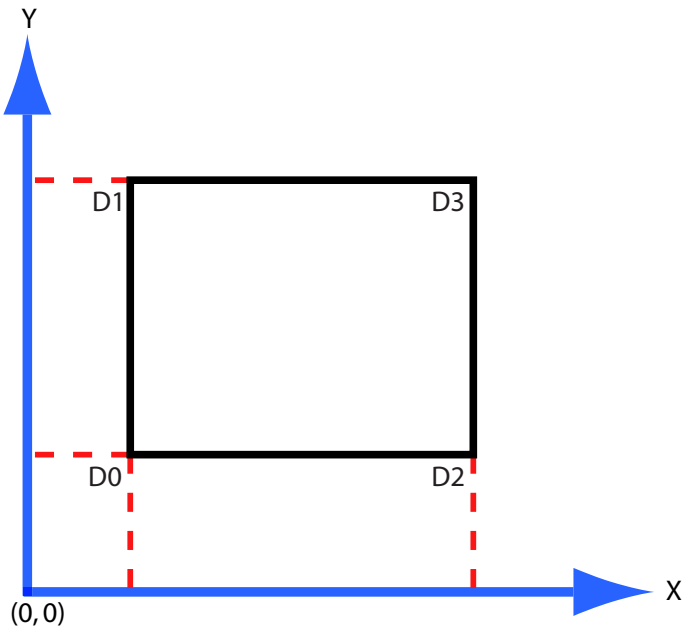


Figure 7.3: Summed area table: Table contains number of important points for all areas from origin to any point.

The density d of an area is the ratio between the number of important points and the size of the area:

$$d = D/((p_{max,x} - p_{min,x}) \cdot (p_{max,y} - p_{min,y})) \tag{7.7}$$

7.3.3 Mesh Generation

We present two different algorithms for clustering cells into axis-aligned regions. The algorithms try to create meshes that have a density d over a certain density threshold value. This guarantees that the whole AMR stores at least a certain amount of important samples in every mesh. The higher the density threshold is set, the more meshes will be generated. Thus a higher density threshold usually results in less memory consumption as long as the meshes do not get too small and produce too much overhead (for the mesh information and the sample points which are shared along the common faces of two neighboring meshes). The summed volume table is build for both

algorithms for fast computation of the volume density of any volume.

Growing Algorithm

The *Growing Algorithm* (see Table 7.3) starts at the coarsest refinement level. A bounding box of all points with an importance (low, medium, or high) is calculated and passed to the initial call of **FindBoxesEXP** as box_{in}. The algorithm creates grids so that all important points (points with an importance according to the current refinement level or higher) are covered, and stores them in $boxes_{out}$. As long as there is an important point inside box_{in} which is not covered by a box of $boxes_{out}$ yet (Line 1), this point (Line 2) is used for the initial position of a new box box_{new} (Line 3). This new box gets expanded (Line 4) and the expanded box is added to $boxes_{out}$ (Line 5). The points inside this box are cleared so that they will not be stored by further meshes and thus overlapping grids are avoided (Line 6). The boundaries of the box will be marked separately to signal these points as marked but also to allow other meshes to include these points. The procedure is repeated recursively with the next finer refinement level for every refined mesh of $boxes_{out}$ (if the mesh has a refinement level of 2 or lower).

To expand a volume, the area of the initial cell is first expanded in the plane perpendicular to the major axis alignment (see Section 4.2.1), and stored in box_{best} (Line 8). An area can only be expanded as long as it is inside box_{in} and does not contain cleared points (this avoids overlapping grids). The box gets expanded by one slice along the major axis alignment, in the direction where the new box has the highest volume density (Line 11). Afterwards, the area of the newly acquired slice is expanded again to include all adjacent important points in this slice (Line 12). These steps are repeated until the volume density of the expanded box is below a density threshold (Line 13). The volume expansion can also be seen in the 2D example of Figure 7.4.

FindBoxesEXP(box box_{in}, boxlist $boxes_{out}$)
1. While not all important points inside box_{in} are covered by $boxes_{out}$
2. *fpoint* = Find point inside box_{in} which is not already stored
3. Set box_{new} to cell of *fpoint*
4. Call ExpandVolume(box_{in}, box_{new})
5. Add box_{new} to $boxes_{out}$
6. Clear points inside box_{new}
7. End While

ExpandVolume(box box_{in}, box box_{new})
8. box_{best} = Expand Area of box_{new}
9. Repeat
10. $box_{new} = box_{best}$
11. box_{best} = Expand Volume box_{best} by one slice
12 box_{best} = Expand Area of box_{best}
13. Until density d of box_{best} < density-threshold

Table 7.3: Simplified version of our implementation of the *Growing Algorithm*.

Signature Algorithm

The *Signature Algorithm* was proposed by Berger and Rigoutsos [6] and is adopting signature-based methods used in computer vision and pattern recognition. It has become the standard approach to generate cells for an AMR because it is very efficient and fast. In Table 7.4 we provide a simplified version of our implementation of the algorithm.

First a few terms which are used in this section are described briefly. *cut-point* is used to store a point which defines the position of a cutting plane. The *tag histogram* contains the number of important points on all slices perpendicular to the points on every axis. For example, the entry for slice number i parallel to the yz plane is given by

$$S_{yz}(i) = \sum_{y=1}^{m} \sum_{z=1}^{n} I(i, y, z) \tag{7.8}$$

where m is the size of the box in y-direction and n is the size of the box

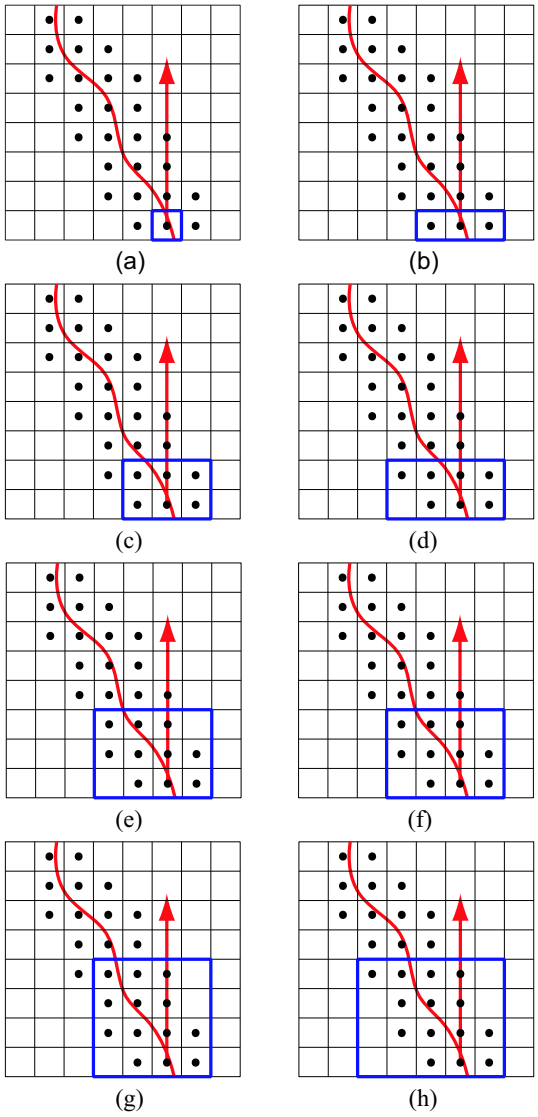


Figure 7.4: Basic steps of volume expansion: The blue box shows the current expanded volume. The red arrow indicates the major axis alignment of the centerline. (a) first an important point that is not covered by a box is selected. In (b), (d), (f), (h) the box is expanded perpendicular to the major axis alignment so that all important points of the current slice are covered. In (c), (e), (g) the box is expanded one step along the major axis alignment. The volume expansion is stopped after (h) because the density is below a threshold.

in z-direction. Further the Laplacian second derivative

$$\Delta_{yz}(i) = -2 \cdot S_{yz}(i) + S_{yz}(i-1) + S_{yz}(i+1) \tag{7.9}$$

is calculated for every entry in the histogram.

An *inflection point* occurs at a sign change of two neighboring Δ entries. The biggest *inflection point* is the *inflection point* with the highest absolute difference between the two neighboring Δ entries.

Figure 7.5 shows an example for the *tag histogram*. The $\sum$ entry contains the number of important points for every slice (in this 2D example a slice is either a row or column). The Δ entry contains the Laplacian second derivative. Each sign change of two neighboring entries of Δ indicate an *inflection point*. The biggest *inflection point* occurs between the 4^{th} and the 5^{th} column (the absolute difference between the two neighboring Δ entries is 8) and is taken as the splitting index (blue line).

FindBoxesBR(box box_{in}**, boxlist** $boxes_{out}$**)**
1. Set box_{new} = bounding box of box_{in}
2. If density d of box_{new} < density-threshold
3. Calculate tag histogram for each dimension in box_{new}
4. If $\exists$ *zero histogram value* in box_{new}
5. Set *cutpoint* to assoc. zero value cell index
6. Else
7. Set *cutpoint* to assoc. inflection cell index
8. End If
9. Split box_{new}into box_{left}, box_{right} at *cutpoint*
10. Call FindBoxesBR(box_{left}, $boxes_{out}$)
11. Call FindBoxesBR(box_{right}, $boxes_{out}$)
12. Else add box_{new} to $boxes_{out}$

Table 7.4: Simplified version of our implementation of the Berger-Rigoutsos Algorithm.

The algorithm (see Table 7.4) starts at the coarsest refinement level and creates meshes so that all important points are covered. A bounding box of the whole volume is passed to the initial call of **FindBoxesBR** as box_{in}. The procedure generates a list of non-overlapping boxes ($boxes_{out}$) where each

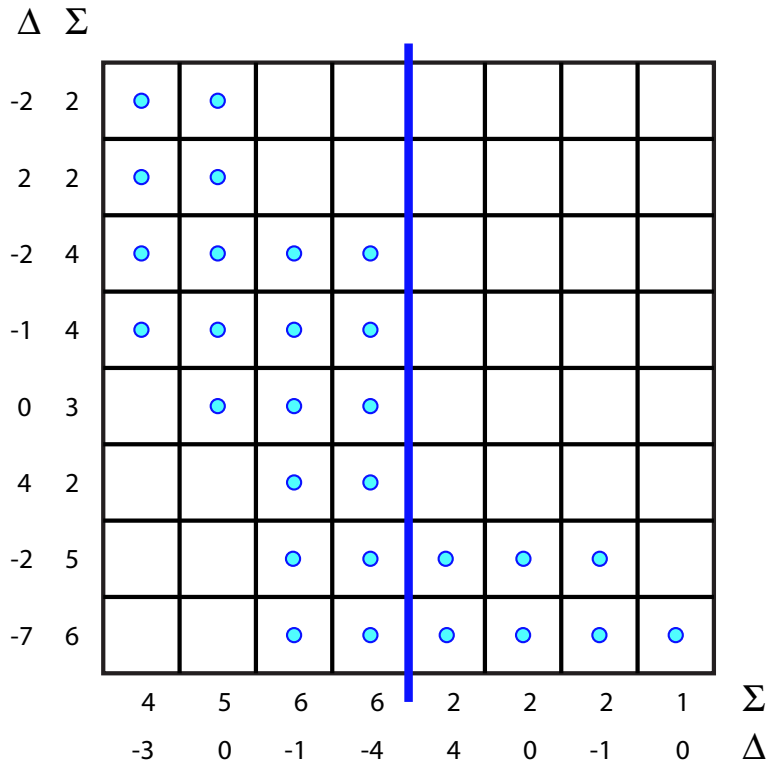


Figure 7.5: Histogram values: The Sum $\sum$ and the discrete Laplacian Δ are calculated for every slice. An *inflection point* occurs at a sign change in Δ. The biggest inflection point is taken as a splitting index (blue line).

of the boxes has a density which is bigger than a certain density-threshold (Line 2) and all important points are covered by the boxes. First the bounding box of box_{in} is calculated and stored in box_{new} (Line 1). If this bounding box has a density over a certain density-threshold it will be added to the output list $boxes_{out}$ (Line 12). If the density of the box is not high enough a tag histogram will be computed (Line 3).

If the *tag histogram* contains an entry with a zero value (Line 4) the *cutpoint* will be set to the index of this entry (Line 5). If more than one zero value entry exists the *cutpoint* will be set to the entry with the highest minimum distance to the bounding box. If no such entry is found the *cutpoint* will be set to the biggest *inflection point* in the histogram (Line 7). Afterwards the current bounding box box_{new} will be split at the *cutpoint* into two

boxes (Line 9). The function **FindBoxesBR** will be called for each of these two new boxes (Lines 10 and 11). One step of the algorithm can be seen in Figure 7.6.

Once all boxes are generated the algorithm will be repeated for every box of the list $boxes_{out}$ with the next finer refinement level (if the current refinement level is greater than one).

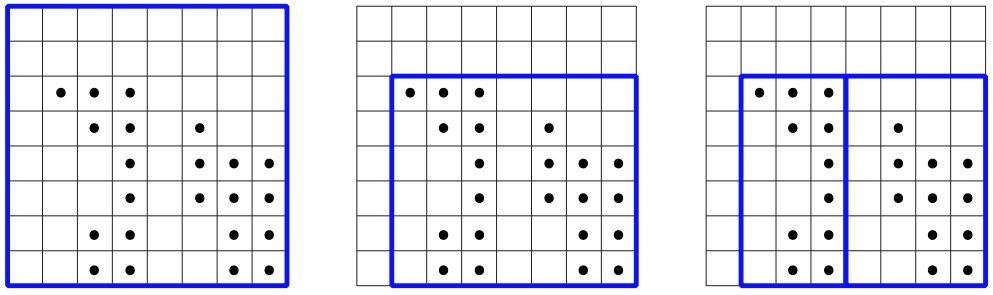


Figure 7.6: One step of the *Signature Algorithm*: (a) the blue box marks the initial box. (b) the bounding box is set. (c) the box is split at an empty slice into 2 new boxes.

7.3.4 Visualization

The *Adaptive Meshes* can be visualized with many volume visualization techniques, such as CPR and DVR. All visualization techniques need to reconstruct a value at an arbitrary position. Therefore the meshes which contain the eight surrounding voxels of the current position have to be identified because each mesh only stores the points which are not stored already in a coarser mesh. Afterwards those eight voxels have to be interpolated to reconstruct the value at the current position.

Resampling

To get the value at a certain position $(x_{cur}, y_{cur}, z_{cur})$ the eight closest voxels have to be found. First, the finest available mesh which contains this position has to be identified. Starting at this mesh, which has the refinement level

$reflevel$, the positions of the eight closest voxels are:
$p_0 = (x_{left}, y_{left}, z_{left})$
$p_1 = (x_{right}, y_{left}, z_{left})$
$p_2 = (x_{left}, y_{right}, z_{left})$
$p_3 = (x_{right}, y_{right}, z_{left})$
$p_4 = (x_{left}, y_{left}, z_{right})$
$p_5 = (x_{right}, y_{left}, z_{right})$
$p_6 = (x_{left}, y_{right}, z_{right})$
$p_7 = (x_{right}, y_{right}, z_{right})$

where the left and right coordinates can be calculated as follows:

$x_{left} = \lfloor x_{cur}/reflevel \rfloor \cdot reflevel$
$x_{right} = x_{left} + reflevel$
$y_{left} = \lfloor y_{cur}/reflevel \rfloor \cdot reflevel$
$y_{right} = y_{left} + reflevel$
$z_{left} = \lfloor z_{cur}/reflevel \rfloor \cdot reflevel$
$z_{right} = z_{left} + reflevel$

A voxel (x, y, z) is stored in a mesh if the current mesh is the coarsest mesh or if $x \ mod \ plevel \neq 0 \vee y \ mod \ plevel \neq 0 \vee z \ mod \ plevel \neq 0$ where *plevel* is the refinement level of the parent mesh. Starting at the finest available mesh for the resampled position all eight voxels are tested if they are stored in this mesh. The value of the voxels which are contained in this mesh are retrieved. As long as some voxel values are missing this procedure is repeated recursively for the parent mesh.

The value at the position $(x_{cur}, y_{cur}, z_{cur})$ can now be calculated as a trilinear interpolation of the eight closest voxels.

7.4 Results

7.4.1 Dataset

For testing purposes we used a CTA dataset with a resolution of 512x512x264. The coarsest mesh of our AMR data structure has a refinement level of four. Therefore we extended the dataset to the size of 513x513x265 so that the coarsest mesh can be aligned to the volume boundary. Since every sample value needs two bytes the total size of the volume is 139.5 MB.

7.4.2 Mesh Density

We examined different density threshold values to find the best settings for creation of the adaptive mesh structure in respect to the memory consumption. More meshes are generated because the meshes are smaller to achieve the desired volume density. This also means more total overhead since every mesh has to store internal information (bounding box, pointers to children, etc.) and neighboring meshes share the points at their common faces. Thus it is desirable to keep the total number of meshes small.

The points inside the radius to the centerline are marked with high importance (they need to be stored in full resolution) and the points inside two times the radius to the centerline are marked with medium importance (they need to be stored at least in half resolution).

Algorithm Comparison

With the *Growing Algorithm* the best result was achieved with 60% volume density. The memory size of this AMR structure is only 432 KB. With the *Signature Algorithm* the memory size could only be reduced to 570 KB with a volume density of 50% (see Figure 7.7).

Figure 7.8 shows that the *Signature Algorithm* generates much more meshes than the *Growing Algorithm*. The *Growing Algorithm* takes advantage of the knowledge of the underlying data while the *Signature Algorithm* is a general approach for AMR and needs more meshes to achieve the same mesh density.

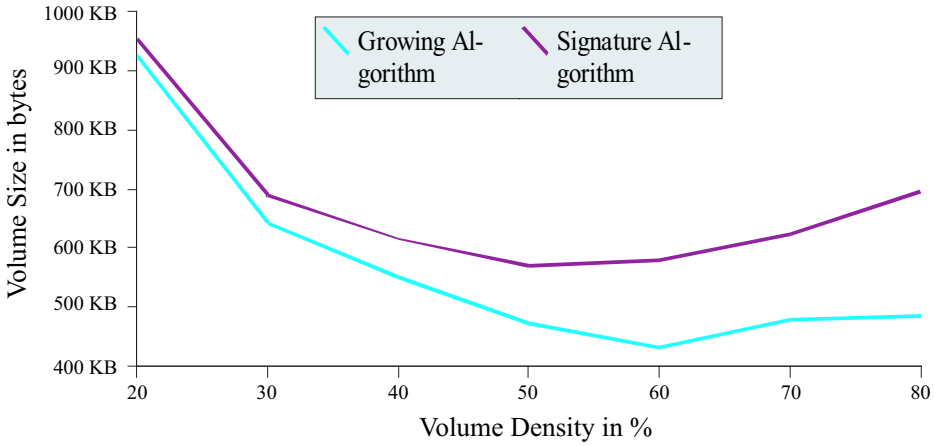


Figure 7.7: Volume size with different density settings.

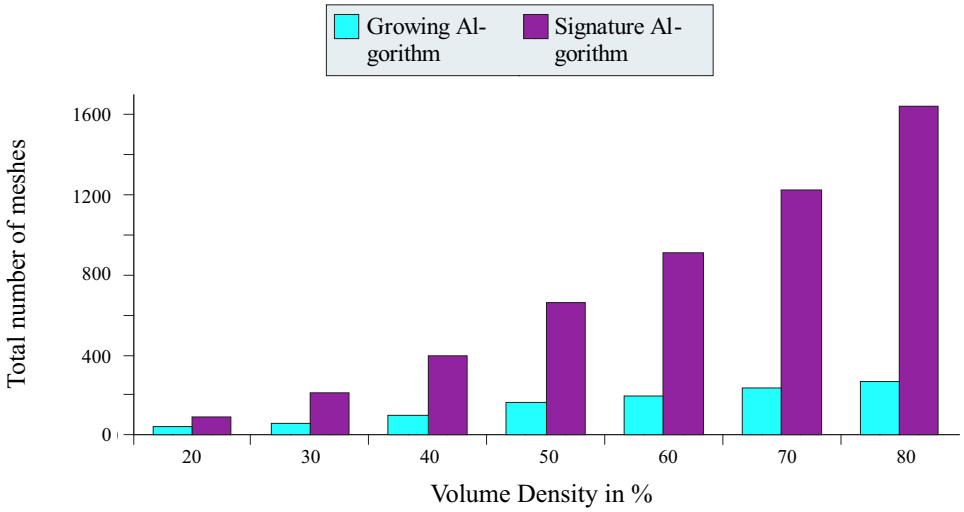


Figure 7.8: Total number of meshes for *Growing Algorithm* and *Signature Algorithm*.

On a Pentium 4 1800MHz with 512MB of RAM the creation time of the AMR data structure with a volume density of 60 % is 1.3 sec with the *Growing Algorithm* and 9.9 sec with the *Signature Algorithm*. The rendering

time for a MIP with the AMR data structure with 60 % volume density is 2.7 sec. For the whole volume the rendering time for a MIP is 13 seconds.

7.4.3 Image Quality

We compared the image quality of a CPR image rendered from the AMR data structure to the same image rendered from the whole volume (i.e., all sample values). The difference image in Figure 7.9 shows the difference of these two images. A white pixel means that this pixel is completely equal in both images. As a pixel in the difference image gets darker the difference between the rendered images grows. We used different settings for marking the data points. The points are marked dependent on their distance to the centerline (see Section 7.3.1).

In Figure 7.9 the adaptive meshes store all points inside the radius to the centerline in full resolution and all points inside two times the radius to the centerline at least in half resolution. The remaining points will be stored at least in a low resolution (every 4^{th} point in each axis). As can be seen a lower resolution leads to more artifacts. Obviously points inside meshes with full resolution are resampled without artifacts.

The strong artifacts on the upper and right side of the difference images are due to the fact that the volume is enlarged for the AMR structure. Points sampled at these borders are interpolated in the adaptive Meshes while the same points are outside of the original volume. This effect is not crucial since it only occurs at the boundaries of the volume.

Further strong artifacts occur at the boundaries of certain objects which have a different data value range than its neighborhood. Since samples are missing, the interpolation is visible due to the rapid change of the data values. Except those objects very few artifacts are visible.

7.4.4 Data Structure Comparison

The total size of our dataset is 139.5 MB. For comparison with other data structures the points have been marked as described in Section 7.4.2. The

easiest way to reduce the volume size is to use one bounding box to include all important samples which results in a total size of 16 MB.

We also implemented an Octree data structure where each node is subdivided if it is not homogeneous (a node is homogeneous if it only contains important samples or is empty). As in the adaptive meshes, neighboring nodes share sample points at their common faces. This produces overhead and makes it necessary to stop the subdivision of an octree node at a certain subdivision level. The smallest size of this data structure was 1.5 MB with a maximum of eight levels.

With our AMR implementation the size could be reduced to 432 KB which is only about 0.3 % of the original volume size (see Table 7.5 for a comparison of the data structures).

data structure	memory space
Mesh (total volume)	139.5 MB
Mesh (bounding box of aorta)	16 MB
Octree	1.5 MB
AMR	432 KB

Table 7.5: Memory consumption of CTA-dataset for several data structures.

Kähler et al [18] have also implemented an AMR which uses a *Signature Algorithm* to generate the meshes. They optimized their data structure for speed while our data structure is optimized for memory size. They store the samples in textures which results in a loss of memory usage because the texture size has to be a power of 2 in modern graphic cards. Sample points are only stored in leaf nodes of the AMR hierarchy while our data structure allows storage of samples in intermediate nodes. We can store areas in different resolutions without producing any overhead for the data because each grid only stores samples which are not already contained in coarser grids.

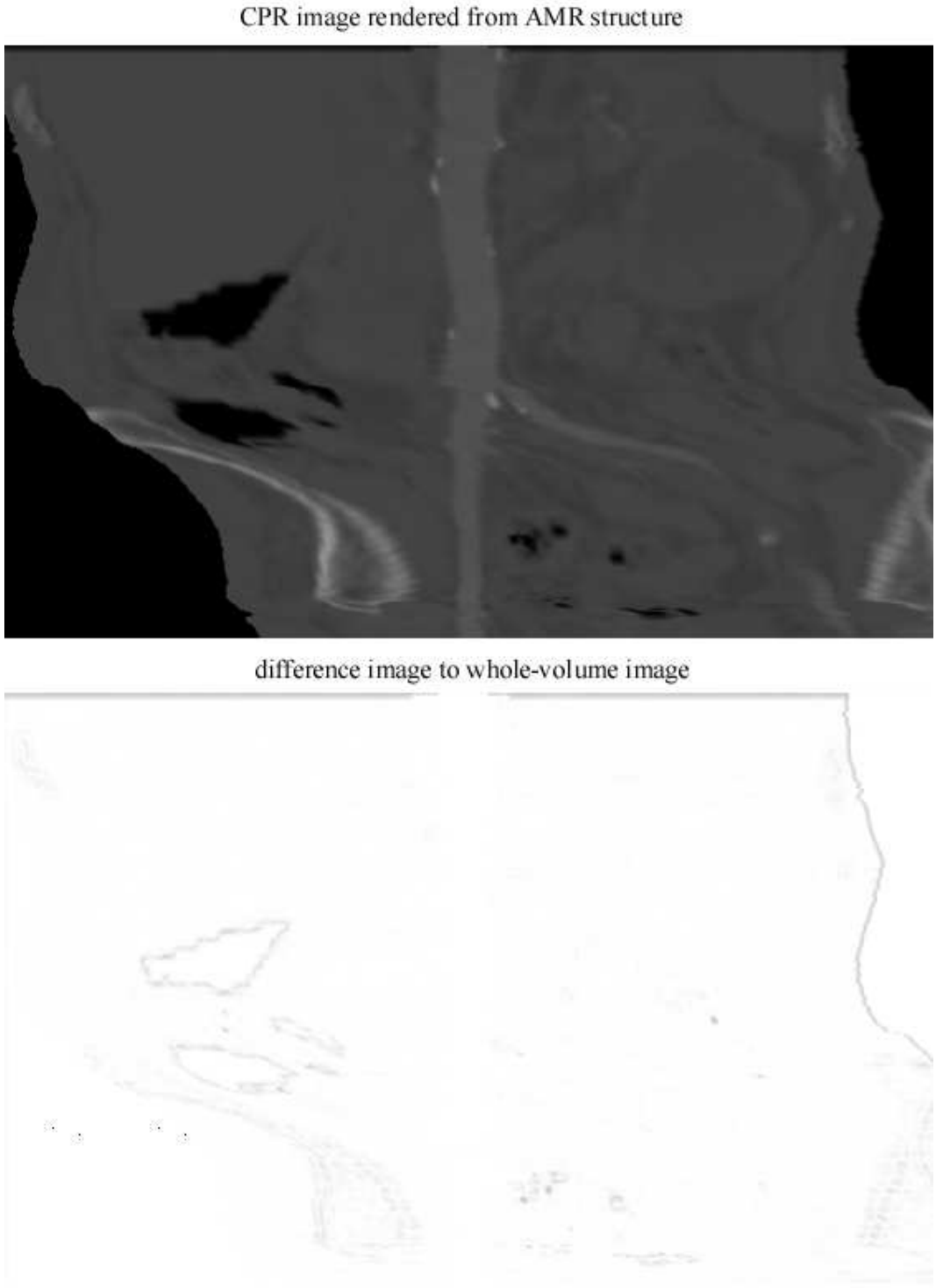


Figure 7.9: Points inside the radius to the centerline are stored in full resolution, points inside two times the radius to the centerline are stored at least in half resolution. Remaining points are stored at least in low resolution.

Acknowledgements

hAS aNYONE sEEN MY
cAPSLOCK kEY?.

Spike Donner

This thesis was written at UC Davis, CA and was only possible due to the encouragement of the master, Prof. Eduard Gröller, and his relations to the outstanding IDAV visualization group in Davis. Many thanks to Prof. Ken I. Joy for supervising me while I was in Davis, giving me the possibility to participate in many meetings and seminars and making a cubicle available for me. I sincerely thank my supervisor Sören Grimm for his encouragement and support while writing this thesis and giving me regularly feedback. Many thanks to Chris Co for his tremendous support and helping me with almost any problem I encountered.

I'm also very grateful for the financial support of my parents as well as the scholarships from the *NÖ Landesakademie* and *TU-Wien*.

Bibliography

[1] http://www.rx-groeninge.be/wat_is_nieuw.htm.

[2] J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[3] M. Bentum. *Interactive Visualization of Volume Data*. PhD thesis, University of Twente, 1995.

[4] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. 82:64–84, May 1989. Lawrence Livermore Laboratory Report No. UCRL-97196.

[5] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. 53:484–512, March 1984.

[6] M. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Trans. Sys. Man. & Cyber.*, 21(5):1278–1286, September/October 1992. NYU Technical Report 501, April, 1990.

[7] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.

[8] K. R. Castleman. *Digital Image Processing*. Prentice Hall, Englewood Cliffs, NJ, USA, 1996.

[9] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the reconstruction of surfaces from tomograms. *Medical Physics*, 15(3):320–327, 1988.

[10] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 307–316, New York, NY, USA, 1981. ACM Press.

[11] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74. ACM Press, 1988.

[12] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics (2nd ed. in C): principles and practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[13] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[14] M. E. Goss. An adjustable gradient filter for volume visualization image enhancement. In *Proceedings of Graphics Interface '94*, pages 67–74, Banff, Alberta, Canada, 1994.

[15] P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proc. of SIGGRAPH-93: Computer Graphics*, pages 165–174, Anaheim, CA, 1993.

[16] G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing*, 9(1):1–21, January 1979.

[17] K. H. Höhne and R. Bernstein. Shading 3d-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging*, 5(1):45–47, March 1986.

[18] R. Kähler, M. Simon, and H.-C. Hege. Interactive volume rendering of large data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):341–351, 2003.

[19] A. Kanitsar. Advanced visualization techniques for vessel investigation. Master's thesis, University of Technology Vienna, Institute of Computergraphics and Algorithm, March 2001.

[20] A. Kanitsar, D. Fleischmann, R. Wegenkittl, P. Felkel, and Meister E. Gröller. CPR - Curved Planar Reformation. In *IEEE Visualization 2002*, pages 37–44, October 2002.

[21] A. Kanitsar, D. Fleischmann, R. Wegenkittl, D. Sandner, P. Felkel, and E. Gröller. Computed tomography angiography: a case study of peripheral vessel investigation. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 477–480. IEEE Computer Society, 2001.

[22] E. Keppel. Approximation complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.

[23] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458. ACM Press, 1994.

[24] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 285–288. ACM Press, 1991.

[25] M. Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, 1988.

[26] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the*

14th annual conference on Computer graphics and interactive techniques, pages 163–169. ACM Press, 1987.

[27] M. Galanski M. Prokop. *Spiral and Multislice Computed Tomography of the Body*. Stuttgart - New York: Thieme Verlag, 2003.

[28] S. R. Marschner and R. Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Visualization*, pages 100–107, 1994.

[29] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A Practical Evaluation of Four Popular Volume Rendering Algorithms. In *Proc. of ACM Symposium on Volume Visualization*, 2000.

[30] L. Mroz. *Real-Time Volume Visualization on Low-End Hardware*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2001.

[31] L. Mroz and H. Hauser. Rtvr: a flexible java library for interactive volume rendering. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 279–286. IEEE Computer Society, 2001.

[32] N.R. Pal and S.K. Pal. A review on image segmentation techniques. *Pattern Recognition*, 26(9):1277–1294, 1993.

[33] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[34] T. Porter and T. Duff. Compositing digital images. 18(3):253–259, July 1984.

[35] H. Qu and A. E. Kaufman. O-buffer: A framework for sample-based graphics. *IEEE Trans. Vis. Comput. Graph.*, 10(4):410–421, 2004.

[36] P. Sabella. A rendering algorithm for visualizing 3d scalar fields. *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, 22(4):51–58, 1988.

[37] C. Schlick. A fast alternative to phong's specular shading model. In *Graphics gems IV*, pages 385–387. Academic Press, 1994.

[38] P. Shirley and A. A. Tuchman. Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24, pages 63–70, 1990.

[39] I. Sobel. An isotropic 3x3x3 volume gradient operator. Unpublished manuscript, May 1995.

[40] M. Sramek. Interactive segmentation of tissues for medical imaging. *Czech Pattern Recognition Workshop '93*, pages 164–171, 1993.

[41] J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 95–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[42] C. Upson and M. Keeler. V-buffer: Visible volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 59–64. ACM Press, 1988.

[43] L. Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376. ACM Press, 1990.

List of Figures

List of Tables