Master's Thesis

Highly accurate Multiresolution Isosurface Rendering

using compactly supported Spline Wavelets

Markus Steinberger

Institute for Computer Graphics and Vision
Graz University of Technology

Supervision: Dipl.-Ing. Dr. techn. Markus Grabner

Graz, April 2010

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)

Abstract

We present an interactive rendering method for isosurfaces in a voxel grid. The underlying trivariate function is represented as a spline wavelet hierarchy, which allows for adaptive (view-dependent) selection of the desired level-of-detail by superimposing appropriately weighted basis functions. Different root finding techniques are compared with respect to their precision and efficiency. Both wavelet reconstruction and root finding are implemented in CUDA to utilize the high computational performance of Nvidia's hardware and to obtain high quality results. We tested our methods with datasets of up to 512^3 voxels and demonstrate interactive frame rates for a viewport size of up to 1024×768 pixels.

Kurzfassung

In dieser Arbeit beschreiben wir eine interaktive Darstellungsmethode für Isoflächen gegeben als Volumsdatensatz. Die zugrunde liegenden trivariaten Funktionen werden als Spline Wavelet Hierarchie realisiert. Durch eine gewichtete Überlagerung der Wavelet Basis Funktionen lässt sich eine beliebige Approximation des Datensatzes herstellen. Dies erlaubt eine adaptive und betrachtungsunabhängige Festlegung des Detailgrades. Wir vergleichen verschiedene Implementierungen (u.A. Nullstellenfinder) und vergleichen diese auf Präzision sowie Geschwindigkeit. Alle relevanten Teile des Ansatzes sind in CUDA implementiert, um die volle Rechenkraft aktueller Nvidia Grafikkarten auszunutzen und hochqualitative Ergebnisse zu erzielen. Wir haben unsere Algorithmen mit Datensätze der Größe 512^3 getestet und erreichen interaktive Wiederholungsraten für Bildgrößen von 1024×768.

Contents

Chapter 1

Introduction

Contents

1.1 Volume Rendering

Volumetric datasets are a widely used representation of three-dimensional data in a variety of applications, such as medical visualization, engineering and computer games. A convenient form of volumetric data is a *voxel grid*, which is a set of samples on a regular grid in 3D-space. Due to the advance of data acquisition technologies, such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI), and more complex simulations, the size of used datasets is steadily increasing. The demanding task of rendering such datasets at interactive framerates often exceeds the processing power provided by the CPU. However, the raw processing power of current graphics cards exceeds the CPU's by far. Traditionally, this power has only been accessible by the fixed function pipeline used for triangle and fragment processing, or since the advent of programmable shading by vertex, geometry and fragment shaders. Since the release of NVIDIA CUDA, the full power of the graphics cards is accessible to the programmer by a convenient C interface, circumventing the use of complex modifications of algorithms to fit into shader limitations. Current volume rendering frameworks are often still shader based as they incorporate the use of bounding geometry, which is easily handled by graphics APIs and the graphics pipeline.
Direct volume rendering [EHK+06] is one technique that can be implemented this way and is suitable to display volumetric data in many cases. Every value in the volumetric data set is mapped to a color and opacity value and projected onto the screen. When defining data values semitransparent, this approach offers structural insight of multiple

layers within the dataset. For better understanding of the three-dimensional shape present in the dataset, local or even global lighting methods can be used. The resulting image strongly depends on the choice of mapping from data values to presentation values, which often turns out to be a demanding task, if a certain feature of the dataset is desired to be visualized.

1.2 Isosurface Rendering

However, the user is often interested in an *isosurface* instead, i.e. a surface satisfying $f(x, y, z) = c$, where $f(x, y, z)$ represents the volumetric data at the point (x, y, z) in object coordinates. Depending on the application, this can have different interpretations, e.g. the boundary between a bone and its surrounding tissue, or a region of constant pressure in a simulated combustion engine. By modifying the constant c over the range of f, the user can investigate the entire dataset and thus gain insight into its geometric and topologic properties. When rendering a single isosurface only, one point in the dataset contributes to a pixel's value on screen, therefore this task can be realized more efficiently than direct volume rendering of a whole volume. This thus gained processing time can, e.g. be spent on higher order interpolation of the data values or a more complex local lighting model.
Although isosurface rendering is a task on its own, it is often combined with direct volume rendering for applications like virtual endoscopy [SHN+06] or visualization of 3D-segmentations providing the advantage of a smoother and more exact rendering due to the higher order interpolation.
For a well-defined isosurface to exist, we have to assume an appropriate interpolation of the samples, which approximates the original function in the continuous domain. Common choices include polynomial (e.g. tri-linear or tri-cubic) interpolation functions, for which the isosurface becomes a piecewise algebraic surface, where each piece is defined as

$$f(x, y, z) = \sum_{0 \le i,j,k \le d} f_{ijk} x^i y^j z^k = 0, \tag{1.1}$$

with f_{ijk} given and $3d$ being the surface's degree[1].

[1]Note that in general the degree d of a tri-variate polynomial is defined with $0 \le i + j + k \le d$ in Equation (1.1). Since we only discuss polynomials with equal degree along each dimension, we denote the degree along one dimension by d.

1.3 Multiresolution

As mentioned above, the datasets investigated by researchers are steadily increasing. It is often neither desirable nor feasible to use the entire available data for visualization, instead, only a properly chosen subset of the original data is used. The wavelet transformation [Chu92] is well suited for this purpose, since it decorrelates data and allows the selective removal of irrelevant data, while maintaining a good approximation of the original data. Even after simplification, isosurface rendering still remains a demanding task.

Our approach further deviates from traditional rendering systems and therefore we have chosen to use CUDA as a platform that also offers the needed flexibility to cope with the complexity of advanced strategies. Furthermore, recent programmable graphics hardware [LNOM08] provides sufficient computational power to accomplish it in real-time due to the highly parallel nature of the problem.

We also present a wavelet-based hierarchical representation of volume data, from which an approximation of the input data can be reconstructed according to the current viewing parameters. This task can also be carried out efficiently by the GPU. The wavelet basis functions are written in the scaled Bernstein form, which allows simple algebraic manipulation by convolution [SR03].

Performance critically depends on proper code optimization. Current compiler sets are very good in understanding static control structures. While they can optimize these structures to a high extend, they do fail for some constructs, which may have a dramatic influence on the overall performance. To assist the optimizer, we add an advanced preprocessing mechanism, which simplifies the control flow structure.

1.4 Organization of this Work

In Chapter 2, related work on isosurface rendering, algebraic surface rendering and multi resolution strategies is reviewed. The core of our rendering approach is explained in detail in Chapter 3. We start by explaining the basis of our method, which is an algebraic surface renderer and extend the discussion to volume datasets given as voxel grids. Different data representation as well as different rootfinders are compared and their realizability for GPU-based execution is investigated. The discussion of the implementation details is complimented by simplified C code of the used algorithms, which can be found in Appendix A.

After the core rendering strategies have been discussed, we continue with explaining how spline wavelets can be used in this context to generate mixed resolution isosurface

renderings in Chapter 4. Although the renderer explained in Chapter 3 is used for multiresolution rendering, it is no prerequisite for understanding Chapter 4.
Code optimization and a short demonstration of the resulting speedup using our preprocessing step is discussed in Chapter 5.
Results and detailed performance analysis as well as a comparison to a dedicated algebraic surface renderer and a hardware supported isosurface renderer is given in Chapter 6. We also show performance measurements and a level of detail comparison for our multiresolution approach.
Chapter 7 holds concluding remarks.
A short version of this work has been presented at the IEEE/EG International Symposium on Volume Graphics 2010 [SG10].

Chapter 2

Related Work

Contents

As this thesis covers both, volume rendering of voxel grids and algebraic surface rendering and both topics are well covered in literature, there is a lot of work available, which could be mentioned here. However, we will only focus on latest and most important work.

2.1 Algebraic Surface Rendering

As a matter of fact, the idea of rendering algebraic surfaces is as old as computer graphics. Theoretically, algebraic surfaces are an elegant way for describing objects with little data. Unfortunately, describing complex objects with a single algebraic surface is infeasible, as it would be necessary to increase the degree of the underlying polynomial too far. Already in the 1980s, a first algebraic surface rendered using ray casting has been proposed [Han83]. Despite the facts that a lot of excellent researches focused on this topic and all the computational power that is available today, algebraic surface rendering is still not considered to be efficient enough to meet real-time requirements for more than a few surfaces and small degrees.

Recently, Loop and Blinn [LB06] came up with a technique called Bèzier tetrahedra for rendering piecewise algebraic surfaces of small degree. Their shader based approach can quickly calculate the univariate polynomial along a viewing ray hitting the algebraic surface defined within a tetrahedron. After this step, the root of this polynomial is found analytically. Kloetzli et al. [KOR08] adapted Bèzier tetrahedra for volume datasets. Their framework commences by transforming the data given as a voxel grid to a set of tetrahedrons, where every single tetrahedron stores an algebraic surface. Afterwards they can use the proposed method by Loop and Blinn to render the transformed data. Unfortunately, they can only render small datasets, as this complex data transformation step strongly increases the memory needed to store the volume.

Taking the structure of splines into consideration and efficiently organizing data streams, offers the opportunity to render bigger datasets using splines organized in a tetrahedral form [KZ08]. Still this approach needs a preprocessing step every single time an isovalue change occurs. Efficiently, this step can also be calculated on the GPU [KKG09]. Nevertheless, the amount of data for rendering is strongly increased.
Another dedicated algebraic surface renderer based on raycasting has been developed by Reimers and Seland [RS08]. They came up with the so-called frustum form, which is used to accelerate the computation of the univariate polynomial along the viewing ray. They found a different description for the algebraic surface within the viewing frustum, which makes it feasible to extract common computations shared by all rays. These computations can be condensed into matrix multiplications, which are pre-calculated on the CPU and distributed to the GPU processors. Especially for higher degrees and larger viewports, their work demonstrates the benefit of pre-calculation steps. Apparently, this step is only practicable for only one or just a few surfaces. Their approach uses the Bernstein form as a stable polynomial representation. When making use of the scaled Bernstein basis the manipulation of the underlying polynomials are further simplified by means of the convolution operator [SR03]. However, this has not yet been investigated in the context of isosurface rendering.

2.2 Volume Datasets

When it comes to data given as voxel grids, the well-known Marching Cubes algorithm [LC87] is still a widely used alternative to direct isosurface rendering. This algorithm as well as its follow-up methods [TPG99, ABJ05] extract the given isosurface by constructing a polygonal model. Every single voxel is checked for its intersection with the isosurface and polygons referring to the intersections are added to the mesh. Frequently a simplification algorithm is used to get rid of tiny unnecessarily added triangles. The actual rendering step is carried out by the standard graphics pipeline, as only triangles need to be considered. The three main problems of this approach are the rough and edgy surface of the resulting models, ambiguity of different cases resulting in possible cracks in the surface, and the fact that any change in the isovalue invokes the need of rerunning the whole surface extraction phase.
Consequently, most methods nowadays try to directly render the isosurface. Most methods are sill shader based, as shaders for the first time made it possible to exploit hardware features (e.g. fast texture filtering) and at the same time perform complex tasks. Using fast tri-cubic texture filtering [SH05], it is possible to render high order filtered isosurfaces of big datasets at real-time frame rates [HSS+05]. Again, raycasting is used for

rendering. The ray is coarsely but continuously sampled using the fast tri-cubic texture filtering, until two adjacent samples lie on both sides of the isovalue. The intersection between the viewing ray and the isosurface is found be refining the hit point in a binary search. Furthermore, this method does not require any preprocessing for an isovalue change and works on raw voxel data, therefore making it a good candidate for integration into a mixed direct volume and isosurface rendering system.

2.3 Wavelets for Volume Datasets

A common problem in volume rendering is the huge amount of data required for high-quality rendering. Compression is one possible solution, e.g. by using wavelets [Chu92, SDS96], which is well covered in literature for volume rendering [Wes94,GLDK95,GLDH97, KS99]. Wavelets can also be used to construct a mixed resolution representation of the underlying data. Areas of interest can be displayed in more detail, while the context is visualized with less effort. When used for isosurface rendering, special care needs to be taken at the boundaries between different resolution level to avoid cracks in the surface. Thus wavelets are often used on previously extracted surfaces and not directly on the volume dataset, as the problem can be dealt with more ease [GSG96, WKE99, BDHJ00, LHJ07].

When applied to volume datasets, linear wavelets or linear interpolation can be used to hide boundaries between resolution levels. When data is interpolated only linearly, Marching Cubes can also be used for rendering [UHP00]. The reduced quality of linear filtering is less apparent, when surface normals are constructed with C_1 continuity [KWH09]. Wavelets can further be used to analyze data and truncate unneeded coefficients, e.g. high frequency information or irrelevantly sized basis functions, as shown in [WB97].

Even current graphics cards' memory capabilities still do not provide enough memory to cope with extensive volume datasets in full size. Fortunately, in most cases it is sufficient to hold only parts of the volume in memory. Then an out-of-core memory management system must be incorporated. Recently, Crassin et al. showed a shader based approach [CNLE09] for direct volume rendering, trying to use the available memory as efficiently as possible. They keep track of used and needed data portions and only load essential data into GPU memory as well as demonstrate blending of different resolution levels.

Chapter 3

Rendering Method

Contents

Isosurface rendering is normally related to datasets only given as voxel grids. Considering the definition of an isosurface, as the surface corresponding to the equation $f(x, y, z) = c$, we can see, that the definition for an algebraic surface with $f(x, y, z) = 0$ is approximately the same as for isosurface rendering. One definition can be transformed into another by just subtracting c. The only difference is just the representation of data. For isosurface rendering, the data is given in a sampled manner and requires the application of a reconstruction filter to resemble the original object the data has been sampled from. In the case of algebraic surface rendering, on the other hand, we are given a full description of the data in terms of a polynomial.
As mentioned above, common choices for reconstruction filters for three-dimensional sampled data include tri-linear and tri-cubic filter methods. These filter methods can be described in a polynomial fashion with different polynomial degrees. Extending this polynomial view to the whole dataset, the given data is nothing but a piecewise polynomial function, where each voxel can be described by a single polynomial. This implies that the whole dataset can be viewed merely as a large number of algebraic surfaces which are limited by axis aligned bounding boxes and are placed right next to each other. The reader should keep this in mind, as the proposed rendering approach is based on exactly this view on data given as sampled grids.
For the sake of clarity tri-linear, tri-square and tri-cubic filtering as well as the underlying polynomials, shall be defined in the following. Given a polynomial of the form:

$$f(x, y, z) = \sum_{0 \leq i,j,k \leq d} f_{ijk} x^i y^j z^k = 0,$$

f_{ijk} are the polynomial coefficients and d is the maximal degree appearing along any dimension in the data. For $d = 1$, $d = 2$ and $d = 3$, one gets tri-linear, tri-square and tri-

cubic polynomials respectively. Tri-linear, tri-square and tri-cubic filtering is normally based on B-spline kernels, which can be described by tri-linear, tri-square and tri-cubic polynomials respectively.
Equipped with this basic knowledge, the reader should be able to understand the following discussion on our rendering method. We will start by describing the idea of raycasting used for rendering isosurfaces as well as algebraic surfaces, deal with common steps in the rendering pipeline of both surface types and focus on each individually. The part on algebraic surface rendering is the basis for the following discussion on isosurface rendering, as we will just extent the method described for a single algebraic surface. Therefore the part on algebraic surfaces will detail in implementation and speedup strategies for bottlenecks in the rendering of a single polynomial. The transition to grids of thousands of voxels mainly deals with data organization and speedup strategies, e.g. skipping polynomials, which do not contribute to the final image. Finally we want to extend the data organization part by introducing wavelets as a well suited mutliresolution technique for our kind of surface representation.

3.1 Rendering Pipeline

Raycasting is one common way to render algebraic surfaces or isosurfaces. For each pixel on screen, a ray is casted according to the current viewing position and traced through the given data. The first intersection between the ray and the three dimensional surface defined as $f(x, y, z) = c$ is responsible for the pixel's appearance on screen. Traditional shading methods like Phong lighting are simply based on the surface normal at the hit position, the material assigned to the surface and the light conditions. Phong lighting is normally sufficient for the viewer to understand the three dimensional structure of the surface.

Basic steps of a raycasting rendering pipeline for isosurfaces or algebraic surfaces often are as follows:

1. project the ray $\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{v}$ from screen space to object space
2. limit the search interval by a bounding structure
3. find a representation of the data along the ray $f(t)$
4. find the first root of $f(t) - c = 0$
5. evaluate the ray equation to obtain the hit point in object space $\mathbf{h} = \mathbf{r}(t_{root})$
6. calculate the normal at the given position
7. calculate lighting
8. project the position to normalized device coordinates

Depending on the application, different steps in this pipeline can be altered or split into multiple sub-stages. Still, most raycasting approaches share the basic steps listed above. Projecting the ray to object space simplified handling the data in its original description, as it removes the perspective distortion introduced by perspective rendering. Especially in the case of axis aligned bounding boxes or tree structures like octrees or kd-trees, calculations with limiting planes become increasingly simple, when an object space description is used. After the initial setup of the ray and the intersection test with a bounding structure, a description of the data along the ray needs to be found. For algebraic surfaces, this representation is often calculated explicitly in terms of a polynomial. For isosurfaces this step is often combined with rootfinding by just sampling the ray continuously. After the first hit point is found, the normal and, if needed, other quantities, e.g. curvature, need to be calculated before lighting can be performed. For mixing the raycasting output with rasterizer based rendering methods, the hit position needs to be projected to normalized clip space to determine the depth value for the z-buffer.

3.2 Algebraic Surfaces

In this section we want to present our method for rendering algebraic surfaces. It aims to introduce as little dependencies as possible between rays and circumvent precalculation to enable an easy extension to volume datasets as noted in the beginning of this chapter. When dealing with polynomial data, one can choose between different forms of data representation. Two common ways are the power form and the Bernstein form. The power

form enables fast evaluation while the Bernstein form, on the other hand, is considered to be more stable and is often used in root finding. We want to derive our algorithms for both forms, as polynomials with low degree can profit from the increased speed using the power form, and complex expressions profit from good numerical conditions and a uniform data representation through the rendering pipeline using the Bernstein form. To be able to write multiplications of polynomials as only convolutions of coefficient vectors, we use the so called scaled Bernstein form proposed in [SR03].
Our adapted rendering pipeline for algebraic surfaces consists of the following steps:

1. project the ray
$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{v} \tag{3.1}$$
from screen space to object space
2. compute the ray intersection with the bounding box of the object
3. calculate the polynomial along the ray $(f \circ \mathbf{r})(t)$
4. subtract the isovalue c and find the first root position t_{root} of the polynomial $f(t) - c = 0$
5. evaluate the ray equation to obtain the hit point in object space $\mathbf{h} = \mathbf{r}(t_{root})$
6. calculate the normal at the given position
7. calculate lighting

When implementing this pipeline for higher order polynomials, step 3 and 4 typically prove to be the bottleneck of the whole rendering. We will cover all other parts quickly, before detailing on these two steps.

3.2.1 Screen Space to Object Space Transition

As the pipeline needs to be executed for every single ray, one possible execution configuration for the CUDA architecture is assigning one ray to one thread. Every single thread needs to be given its pixel coordinates, the screen dimensions and the combined model-view projection matrix to begin the first part of the pipeline. The transformation of a point p from object space to clip space is defined as:

$$\mathbf{p}_{\text{clipspace}} = \mathbf{m}_{proj} \cdot \mathbf{m}_{\text{modelview}} \cdot \mathbf{p}_{\text{object}},$$

where $\mathbf{m}_{\text{modelview}}$ and $\mathbf{m}_{\text{proj}}$ correspond to the model-view and projection matrix respectively. The clip space coordinates need to be divided by the w-coordinate to transform

the point to normalized device coordinates. To transform a point from normalized device coordinates to object space, one needs the inverse of $\mathbf{m}_{\text{proj}} \cdot \mathbf{m}_{\text{modelview}}$, which can be calculated on the CPU before the GPU kernel is started.
Again a division by w is needed:

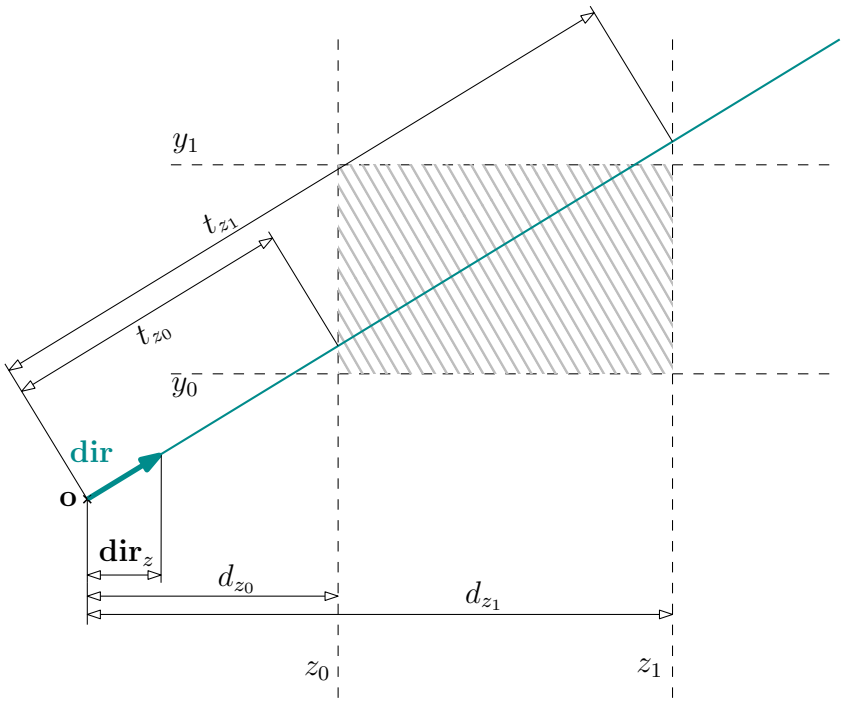
$$\mathbf{p}_{\text{intermediate}} = (\mathbf{m}_{\text{proj}} \cdot \mathbf{m}_{\text{modelview}}) \cdot \mathbf{p}_{\text{normalizeddevice}}$$
$$\mathbf{p}_{\text{object}} = \frac{\mathbf{p}_{\text{intermediate}}}{\mathbf{p}_{\text{intermediate,w}}}$$

A simple way to define a ray in object space starting from pixel coordinates is defining two points in normalized device coordinates fulfilling the ray equation and transforming both of them to object space, as described above. A given viewing ray in normalized device coordinates is defined by constant x and y coordinates. These two coordinates can be deferred from the pixel coordinates, using the screen dimensions to map them to the defined normalized device space bounds. Although the z coordinates of both points can theoretically be chosen arbitrary, setting them to the limits of the normalized device space is a convenient choice. Thus places them directly on the near clip and far clip plane. When using the transformed points in object space, one can limit the rendering result to the viewing frustum with only considering the ray segment defined by these two points – $\mathbf{p}_{\text{object,min}}$, $\mathbf{p}_{\text{object,max}}$. The ray $\mathbf{r}$ is then defined as:

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{v}$$
$$\mathbf{p} = \mathbf{p}_{\text{object,min}}$$
$$\mathbf{v} = \frac{\mathbf{p}_{\text{object,max}} - \mathbf{p}_{\text{defaultobject,min}}}{|\mathbf{p}_{\text{object,max}} - \mathbf{p}_{\text{defaultobject,min}}|}$$
$$t \in [0, |\mathbf{p}_{\text{object,max}} - \mathbf{p}_{\text{defaultobject,min}}|]$$

3.2.2 Ray Bounding Box Intersection

Given the ray segment $\mathbf{r}(t)$ limited by $t \in [t_{\min}, t_{\max}]$, this interval can easily be refined by the intersection with an axis aligned bounding box. The ray can be tested against six half-spaces, each of them defined by one face of the bounding box, leading to six intersection distances t_i, which are used to further limit t.
As the intersection algorithm for all three sets of plane pairs just differs in the coordinate index, we shall consider the xy-plane pair, which correspond to $z = z_0$ and $z = z_1$:



t_{z_0} and t_{z_1} can be computed knowing $\mathbf{v}$, $\mathbf{o}$ and the planes of the axis aligned bounding box:

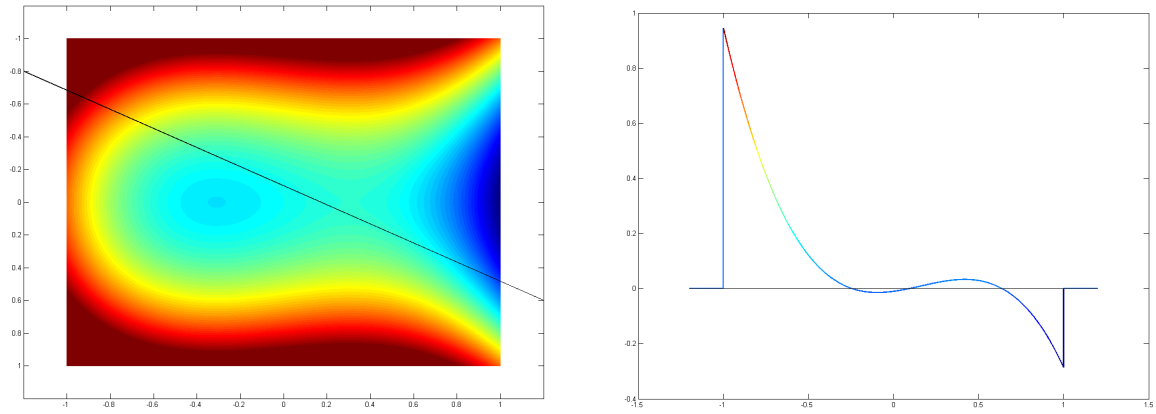
$$
\begin{aligned}
d_{z_0} &= (z_0 - \mathbf{p}_z) \\
t_{z_0} &= d_{z_0}/\mathbf{v}_z \\
d_{z_1} &= (z_1 - \mathbf{p}_z) \\
t_{z_1} &= d_{z_1}/\mathbf{v}_z
\end{aligned}
$$

A special case occurs, if the ray is parallel to one plane. To avoid a division by zero this condition must be precluded beforehand. When supporting an arbitrary viewing position, it can not be known if the ray intersects the plane defined by $z = z_0$ before $z = z_1$ or the other way around. By exchanging the two intersection points, if the right order is not given, the condition $t_{z_0} \leq t_{z_1}$ can be assured.
After applying this scheme to all three dimensions, the actual intersection interval can be determined. For the given boundaries, e.g. $t_{z_0} \leq t \leq t_{z_1}$, the ray is inside both planes. The ray is limited to the whole bounding box and the near and far plane, if t stays within all boundaries:

$$\max(t_{\min}, t_{x_0}, t_{y_0}, t_{z_0}) \leq t \leq \min(t_{\max}, t_{x_1}, t_{y_1}, t_{z_1})$$

3.2.3 Horner Scheme for Composition

The composition $(f \circ \mathbf{r})(t)$ is nothing but a description of the polynomial along the ray. For a two dimensional example, see Figure 3.1.



(a) Bi-squared area polynomial

(b) Univariate polynomial of degree 4

Figure 3.1: Two dimensional example of composition $(f \circ \boldsymbol{r})(t)$.

For simplicity and due to focus on volume data, from now on only the three dimensional case shall be considered. In this case the composition $(f \circ \mathbf{r})(t)$ transforms a trivariate polynomial (e.g. tri-linear, tri-square...) into a univariate polynomial of degree $3 \cdot d$. To circumvent this costly composition, some approaches [HSS+05] even work directly on the trivariate polynomial $f(x, y, z)$. We, on the other hand, offer an efficient approach for this step, which shows a tremendous speedup in comparison to naive implementations. To depict the complexity of the task, we want to show a straight forward formulation of the composition in power form. Therefore, we just replace every directional component x, y and z of 1.1 by its correspondent from the ray equation 3.1 resulting in $(d+1)^3$ different expressions of the form:

$$(f \circ \mathbf{r})(t) = \sum_{i=0}^{d} \sum_{j=0}^{d} \sum_{k=0}^{d} f_{i,j,k} \cdot (p_x + t \cdot v_x)^i \cdot (p_y + t \cdot v_y)^j \cdot (p_z + t \cdot v_z)^k$$

For comparison we have implemented and tested this structure for tri-cubic polynomials. To reduce the number of operations, all identical computations are precalculated before being used in the final expression. Still, this approach requires a lot of operations and an exceedingly large number of registers as to fit into a GPU based framework (see Table 3.1).

To obtain a more efficient structure, we write the composition with convolution operators [SR03]:

$$f(t) \rightarrow \mathbf{f} = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k=1}^{d} f_{ijk} \cdot \mathbf{r}_x^i * \mathbf{r}_y^j * \mathbf{r}_z^k,$$

with $\mathbf{r}_x = [p_x, v_x]$ being the coefficient sequence of the (linear) polynomial describing the x-component of the ray in equation (3.1), $\mathbf{r}_y$, $\mathbf{r}_z$ defined likewise, and $\mathbf{f}$ being the coefficient sequence of the resulting polynomial $f(t)$ of degree $3d$. The power expressions are n-folded convolutions:

$$\mathbf{r}_x^n = \underbrace{\mathbf{r}_x * \mathbf{r}_x * \cdots * \mathbf{r}_x}_{n \text{ times}},$$

which are computed incrementally, i.e. $\mathbf{r}_x^n = \mathbf{r}_x^{n-1} * \mathbf{r}_x$ (this is equivalent to the Horner scheme for polynomial evaluation), and similar for $\mathbf{r}_y^j$ and $\mathbf{r}_z^k$. Reordering leads to

$$\mathbf{f} = \sum_{i=0}^{d} \mathbf{r}_x^i * \underbrace{\sum_{j=0}^{d} \mathbf{r}_y^j * \underbrace{\sum_{k=0}^{d} f_{ijk} \cdot \mathbf{r}_z^k}_{\mathbf{f}_{ij}}}_{\mathbf{f}_i}$$

This structure separates the calculations needed for all three dimensions as good as possible. For realization on the GPU another important property of this structure can be exploited: Merely one factor $\mathbf{f}_{ij}$ and $\mathbf{f}_i$ need to be available at the same time. Thus, the memory for $\mathbf{f}_{00}$ is reused for $\mathbf{f}_{01}$ and so on, therefore, also enabling better parallelism. This evaluation scheme strongly decreases memory usage when implemented accordingly. The C code for the structure can be found in Listing A.1.

The Bernstein form of a polynomial is often used for root finding, as it shows good numerical properties [FR87]. Our approach can also be derived in the so called scaled Bernstein form proposed in [SR03]. This offers a uniform data representation throughout the rendering pipeline, better stability for the composition and also avoids basis changes from power form to Bernstein form for root finding:

$$\begin{aligned} \widetilde{\mathbf{f}} &= \sum_{i,j,k=0}^{d} \widetilde{b}_{ijk} \cdot \widetilde{\mathbf{q}}_x^{d-i} * \widetilde{\mathbf{u}}_x^i * \widetilde{\mathbf{q}}_y^{d-j} * \widetilde{\mathbf{u}}_y^j * \widetilde{\mathbf{q}}_z^{d-k} * \widetilde{\mathbf{u}}_z^k \\ &= \sum_{i=0}^{d} \widetilde{\mathbf{q}}_x^{d-i} * \widetilde{\mathbf{u}}_x^i * \sum_{j=0}^{d} \widetilde{\mathbf{q}}_y^{d-j} * \widetilde{\mathbf{u}}_y^j * \sum_{k=0}^{d} \widetilde{b}_{ijk} \widetilde{\mathbf{q}}_z^{d-k} * \widetilde{\mathbf{u}}_z^k, \end{aligned} \tag{3.2}$$

with $\widetilde{\mathbf{u}}$ being the ray component in scaled Bernstein form, and $\widetilde{\mathbf{q}} = \mathbf{1} - \widetilde{\mathbf{u}}$.

For a single dimension, the composed scaled Bernstein coefficient sequence $\widetilde{\mathbf{b}}$ can be obtained in a recursive manner (similar to the Horner scheme for the power form):

$$\begin{aligned}\widetilde{\mathbf{b}} &= \sum_{i=0}^{d} \widetilde{b}_i \widetilde{\mathbf{q}}^{d-i} * \widetilde{\mathbf{u}}^i \\ &= \widetilde{b}_d \widetilde{\mathbf{u}}^d + \widetilde{\mathbf{q}} * (\widetilde{b}_{d-1} \widetilde{\mathbf{u}}^{d-1} + \ldots + \widetilde{\mathbf{q}} * (\widetilde{b}_1 \widetilde{\mathbf{u}} + \widetilde{b}_0 \widetilde{\mathbf{q}})).\end{aligned}$$

A large amount of computational work can be saved by applying this substitution to the three sum terms in Equation (3.2). As the expression $\widetilde{\mathbf{q}}^{d-k} * \widetilde{\mathbf{u}}^k$ in the sum over k is computed repeatedly, the number of operations can additionally be reduced by precomputing and reusing it for $k = 0 \ldots d$. Against our first intuition, our simulations have shown that performing this precalculation even for the sum over j provides a faster algorithm although more operations are then needed. We assume the reason for this surprising results are the fewer dependencies of this structure, which offer the optimizer more freedom and the ability to combine more multiplications and additions to 'mad' operations. In contrast to the power basis, the Bernstein basis consists of non-zero components and therefore, the evaluations turn out to be more complex.
See Table 3.1 for an overview of the different implementations. The difference of two to three orders of magnitude in performance between the naive approach and our proposed structure, is based on the fact that our structure easily fits within the register space of current graphics cards. Although the implementation in scaled Bernstein form is approximately six times slower than the implementation in power form, this difference is responsible for only a few percent in the overall rendering time. The big speedup in this step of the algebraic surface rendering pipeline shifts the main execution time to the rootfinding step. One can also see that the scaled Bernstein form provides the lowest MSE. The difference to the power form implementation is only crucial for rays that are close to silhouettes of the surface. For more details on optimizations that should be implemented for an efficient execution on graphics cards hardware, see Chapter 5.

3.2.4 Rootfinding

After focusing on composition, we want to focus on root finding, which consumes most of the execution time. A good and stable rootfinder is essential for an exact evaluation of the intersection between viewing ray and surface.
The first intersection between viewing ray and surface corresponds to the first root, which is found along the ray. For an efficient implementation, a rootfinder should be chosen, which tries to determine only the first root, not wasting any effort on other possible roots behind the first one. Most rootfinders used in algebraic surface rendering work on

method	naive	PF	SB0	SB1	SB2
mul	1839	234	741	687	720
add	990	360	483	490	558
reg	135	27	72	60	66
time	1	3.6e-3	26e-3	23e-3	22e-3
MSE	162.3	22.6	5.15	4.78	4.78

Table 3.1: Comparison between different algorithms for trivariate to univariate composition $(f \circ \boldsymbol{r})(t)$ of random polynomial data in the tri-cubic case. Operation count is taken from the optimized machine code of the CUDA compiler. Execution time is stated relatively to slowest method, mean squared error is scaled by 10^{-12} and relative to the coefficients. PF stands for power form and SBx for scaled Bernstein form with terms $\widetilde{\boldsymbol{q}}^{d-k} * \widetilde{\boldsymbol{u}}^{k}$ precomputed for the x innermost loops. The naive approach is described in the beginning of Section 3.2.3.

the Bernstein form or a spline representation, which can be derived from the Bernstein form. The variation diminishing property is what makes the Bernstein form a common choice. The coefficients in Bernstein form can be seen as a control polygon placed at a regular interval. A root can only exist, if this control polygon crosses the zero axes.
Another representation that can be described using a control polygon are B-splines. Furthermore, if there is a crossing between the control polygon and the zero axes, a good estimate for a root can be found in this crossing. Especially when dealing with higher order polynomials, roots that are close to the silhouette of the surface can be badly conditioned, therefore, we will mainly focus on stable rootfinders in Bernstein form and B-spline form. Rootfinders in power form suffer from less stability when facing such roots and a lack of geometrical representation of the coefficients. This makes it more difficult to write efficient algorithms for the power form. Nevertheless, we shall present a root finder in power form.
The facts that most rootfinding algorithms are defined recursively and their register consumption can not be fixed at compile time, make it difficult to write an efficient implementation for graphics hardware. Since CUDA does not support recursive function calls all recursively defined algorithms need to be transfered into a iterative description. Sometimes this limits the maximum number of iterations, as an explicit stack is used. An unknown register consumption is not well suited for this architecture due to mainly two reasons: First of all, the high register count prevents the algorithm to fit into register space of the multi processor. The compiler solves the problem by moving data to local

memory[1], which is around 300 times slower. An alternative is manually placing variables in shared memory, which also limited in size. Secondly, increasing register consumption normally comes along with indexed memory access, which is only provided for shared memory and local memory. Thus the programmer's options are limited and/or the whole algorithm is slowed down. For more details on code optimization see Chapter 5.
We present four different root finding algorithms: [LR81], [MR07], a regula falsi method using De Casteljau subdivision for root isolation and a Sturm Series [LS75] based approach working in power form.

[LR81] Rootfinder

LR81 is based on the fact that the control polygon of the Bernstein form approximates the true shape of the polynomial curve, especially, if the spacing between intervals becomes increasingly small. Thus, if the connection between two consecutive control points crosses the zero axes, this interval is likely to contain a root. Furthermore, this intersection is a good estimate for the root itself and can easily be computed. Hence LR81 suggests to subdivide the control polygon using the De Casteljau algorithm to obtain two new polynomials, whereas either the left or the right polynomial should contain the root close to the end or the beginning respectively. By recursive subdivision the root can be determined to an arbitrary accuracy. For more details on the original algorithm, see [LR81].
Our GPU based version of the algorithm is implemented with constant memory usage (see Listing A.2). This is possible as the algorithm continues to subdivide only one of the two polynomials obtained by the last step. Therefore, the parent polynomial as well as the other polynomial are simply discarded and their memory cell is reused for the child polynomials. This way no index operation is required on the data and the algorithm can be implemented using only fast register space.
LR81 shows quadratic convergence speed with quadratic complexity for each step due to the de Casteljau algorithm. Even for tri-cubic data it seems to be sufficient to fix the number of iterations to 9 to 11 and 5 to 7 for algebraic surfaces and voxel data sets respectively. This limitations did not produce any visible artifacts for our test cases.
A good selection for the fixed number of iterations is crucial. If the number of iterations is too big, it will strongly slow down the whole rendering pipeline. If the the number of iterations is too low, the algorithm might even fail to identify rays which do not actually hit the surface. This can happen, as the variation diminishing property only guarantees

[1]Local memory is a CUDA terminology for thread local memory, which is physically placed in GPU main memory, which is around 300 times slower than the registers located at the multi processor.

for a root within the given interval if an uneven count of zero crossings is present in the control polygon.

[MR07] Rootfinder

MR07 is also based on incrementally refining the polynomial around the root. This algorithm though, does operate on the B-spline form, thus requiring the Bernstein form polynomial to be transformed to a B-spline description. Algorithmically, this corresponds only to an additional knot vector. For refining the root, knots are inserted based on the approximated root position inferred from the linear control polygon.
We also tried to build an efficient GPU based version of this algorithm. Unfortunately, there is no way to rid the implementation of the index operations needed when inserting knots, as the position for knot insertion depends on the polynomial, which changes for different input data and viewing position. Therefore, we placed the coefficient vector, as well as the knot vector in shared memory, to prohibit the compiler from placing it in local memory. Furthermore, these two vectors grow in size with every knot insertion. As the knot insertion can occur at an arbitrary location, parts of the vectors have to be shifted in memory. One possible countermeasure for too extensive memory growth, might be dropping knots at the front or back, depending on the current insertion position. As it is possible that the estimation of the root location emerges as a dud, this countermeasure bares the risk of discarding existential information.
When it comes to convergence and execution speed, MR07 shows linear complexity for both. Still, knot insertion is a quite demanding task, which involves divisions and copy operations, which, in combination with all other drawbacks, renders this algorithm not so well suited for GPU execution. Therefore, our optimized MR07 version is more than ten times slower than our LR81 implementation.

Regula Falsi with De Casteljau Subdivision (aRFBS)

We also implemented a Regula Falsi rootfinder combined with a binary search scheme (for guaranteed exponential convergence) using De Casteljau Subdivision for root isolation (see Listing A.3). The principle idea of this algorithm is the possibility to use a fast and simple root finder, if it can be guaranteed that only a single root exists in a given interval. For a polynomial given in Bernstein form, this condition can again be tested using the variation diminishing property.
The root isolation phase splits the given polynomial exactly between the first two zero crossings of the control polygon. Still, this split does not essentially need to produce a separated root. The split could also just reveal a dud or produce another control polygon

with two zero crossings. However, in virtually every case exhibited in our examples, one to three splits were sufficient to separate the first root.
The second phase consists of alternating a binary search step with a regula falsi step. Both need only a fast polynomial evaluation, which we implemented in scaled Bernstein basis, for increased speed. The evaluation itself is based on the Horner Scheme adapted for scaled Bernstein form for a faster polynomial evaluation. An alternative in Bernstein form would be the De Casteljau algorithm, which provides more stability but does so slower.
Although, the second phase of our aRFBS algorithm outperforms our LR81 implementation when a root has been isolated, the isolation step itself is not as well suited for the CUDA architecture as our LR81 implementation. The problem of our aRFBS root isolation step can be found in divergent branches which appear, when a ray can not isolate the root quickly, or needs more time to conclude that there is no root in the whole polynomial. In this case, a big number of threads are idle, until all of them can continue with the second phase. All other characteristics of the implementation are as well suited for GPU execution as our LR81 implementation: constant memory usage, no compile time unknown index operations and high arithmetic intensity [LNOM08] in comparison to memory access.

Binary Search based on Sturm Series

Root finding is not only limited to polynomials in Bernstein form, and as we are able to formulate all other parts of the pipeline using power form, we also implemented a rootfinder in power form. We apply the Sturm Series [LS75], which can be used to count multiplicities of roots in a given interval. We subdivide the given search interval in a binary manner to confine the root further and further (see Listing A.4). This simple algorithm is converging exponentially with constant memory consumption. The Sturm Series is only computed one time at the beginning of the algorithm and evaluated once during every iteration step. Therefore, and due to little branching, the algorithm is especially fast for an increasing iteration count. On the other hand, the numerical problems, which are natural for power form implementations, become increasingly severe for more iterations. The second down side of the algorithm is its high memory consumption, as the complete Sturm Series needs to be kept in memory during the whole execution.

Comparison

For a comparison between all four rootfinders see Table 3.2. Although, aRFBS is the fastest algorithm on average, we suggest to use our LR81 implementation, as it shows

the least mean squared error and least maximal error, whilst being nearly as fast as the aRFBS approach. The relatively high maximal error has been tolerated for this comparison, as all algorithms were setup to achieve approximately the same mean squared error and our MR07 implementation reached the boundaries of shared memory.

method	LR81	MR07	aRFBS	Sturm
reg	42	$30 + 9d + 2i$	43	73
time	0.076	1.0	0.062	0.077
mse	0.003	0.034	0.016	0.003
max err	0.66	0.85	0.99	1.0

Table 3.2: Performance of different root finders for degree 9 (tri-cubic): Execution time and maximum error are relative to worst result indicated by 1.0. Mean squared error is relative to the search interval. Note: only a few iterations (i) for all methods were considered, as our [MR07] implementation reached the limit of shared memory.

3.2.5 Normal Estimation and Lighting

For algebraic surfaces, the normal, which is very often needed for local lighting models, can easily be evaluated by calculating the gradient of the algebraic surface at the hitpoint **h**:

$$\mathbf{n} = \nabla f(x,y,z) = \left[\frac{\partial f(x,y,z)}{\partial x}, \frac{\partial f(x,y,z)}{\partial y}, \frac{\partial f(x,y,z)}{\partial z} \right]^T \Bigg|_{\mathbf{h}}$$

When comparing the normal evaluation with other parts of the pipeline, this step proves to be quickly computable. The needed data coefficients are normally present in texture cache, as they have been used for the composition step of the same data. The derivative calculations are yet again accelerated using the Horner scheme.

For our implementation we simply used Phong lighting and calculated the local lighting model in a shader applying deferred shading. If there is just one isosurface/material present in the scene, it is considerable faster to calculate the lighting model in CUDA and just transmit the resulting color and depth values to OpenGL, as the current CUDA architecture needs to copy buffers whenever they are interchanged between OpenGL and CUDA.

Of course, our implementation is not limited to shading techniques that solely rely on the normal. It is also easy to access curvature or other local features, as long as they can be deduced from the algebraic definition. Thus, it is no problem to produce images as proposed in [HSS+05].

3.3 Rendering Thousands of Trivariate Polynomials

As the size of common voxel data sets is steadily increasing, it is common to face datasets of 512^3 voxels and more. Therefore, the need for good empty space skipping strategies and a multi resolution approach naturally arises. Still, there will be thousands of voxels active at the same time. For a smooth isosurface, higher order interpolation is required, e.g. tri-cubic interpolation is needed for smooth reflections. From an algebraic point of view, this corresponds to a trivariate polynomial for each voxel, which is influenced by $(d+1)^3$ data values (influence coefficients **IC**).
Our approach shows good frame rates for this scenario, as every ray can work through the whole rendering pipeline independently from all others. In contrast, other methods like [RS08] are designed for a single polynomial and precalculate common terms for all rays. Such approaches do not scale well for an increasing number of polynomials, as the precalculation step is too time consuming when only a few rays – that actually hit this one voxel – can benefit from it.

3.3.1 From a Voxel Grid to Algebraic Surfaces

To transform a given voxel grid into an algebraic surface description, every trivariate polynomial needs to be calculated from its surrounding sample values. This calculation can either be carried out before the renderer is started or on-the-fly, as an additional step in the rendering pipeline. Every explicit polynomial description needs $(d+1)^3$ coefficients, therefore calculating the polynomial description of every single voxel beforehand increases the memory consumption for the volume dataset by a factor of $(d+1)^3$. For a few voxels this might be a good choice, given that enough rays hit the same voxels, thus not requiring too much additional bus transfer.
For big volumes, calculating the polynomial description on-the-fly is the only option to stay within memory bounds of current graphics cards. In this case, it is possible to work directly with the original data representation, as it is normally just desired to interpolate the given data.
In our case, however, this transformation corresponds only to a basis transformation. Tri-linear filtering and its natural extensions like tri-cubic filtering, are based on the B-spline polynomials. From another point of view, this means that the polynomial of a certain voxel is given in the B-spline basis by the data values surrounding the voxel. The basis transform from the one dimensional B-spline basis to either the one dimensional scaled Bernstein basis or power basis, corresponds to a multiplication of the data vector with a matrix of size $(d+1) \times (d+1)$.

The one dimensional interpolation polynomials or B-spline basis is given by the columns of the B-spline to power form transition matrices:

- cubic $d = 3$:

$$M_{BSpline \to Power,3} = \frac{1}{6} \cdot \begin{pmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

  where the transformation to power form looks like that:

$$\begin{pmatrix} x^3 & x^2 & x & 1 \end{pmatrix} \cdot M_{BSpline \to Power,3} \cdot \mathbf{c},$$

  with $\mathbf{c}$ being the B-spline coefficient vector. All other matrices are stated in the same manner.

- square $d = 2$:

$$M_{BSpline \to Power,2} = \frac{1}{2} \cdot \begin{pmatrix} 1 & 1 & 0 \\ -2 & 2 & 0 \\ 1 & -1 & 1 \end{pmatrix}$$

- linear $d = 3$:

$$M_{BSpline \to Power,1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

The one dimensional scaled Bernstein basis is given by the formula:

$$\widetilde{b_{i,d}}(x) = x^i(1-x)^{d-i} = u^i v^{d-i}$$

or explicitly, by the matrices:

- cubic $d = 3$:

$$\widetilde{B_3} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 1 & 0 & 0 \\ 3 & -2 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{pmatrix}$$

- square $d = 2$:

$$\widetilde{B_2} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix}$$

- linear $d = 1$:

$$\widetilde{B_1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

The transformation matrices from the one dimensional B-spline basis to scaled Bernstein basis can be calculated from the above stated matrices:

$$M_{BSpline \to scaledBernstein} = \widetilde{B}^{-1} \cdot M_{BSpline \to Power}$$

- cubic $d = 3$:

$$M_{BSpline \to scaledBernstein,3} = \frac{1}{6} \cdot \begin{pmatrix} 1 & 4 & 1 & 0 \\ 0 & 12 & 6 & 0 \\ 0 & 6 & 12 & 0 \\ 0 & 1 & 4 & 1 \end{pmatrix}$$

- square $d = 2$:

$$M_{BSpline \to scaledBernstein,2} = \frac{1}{2} \cdot \begin{pmatrix} 1 & 1 & 0 \\ 0 & 4 & 0 \\ 0 & 2 & 1 \end{pmatrix}$$

- square $d = 1$:

$$M_{BSpline \to scaledBernstein,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Aiming at matrices containing the highest number of ones, we omit the scaling factors e (1/6 or 1/2), thus reducing the number of required operations. Then, for a one dimensional basis transformation only six and eight multiplications are needed for transitions to scaled Bernstein basis and power basis in the cubic case respectively. When we consider linear polynomials, we see that both basis are identical.
The N-dimensional basis transformation can be constructed from $N \cdot (d+1)^{N-1}$ one dimensional basis transforms, as the N-dimensional basis itself is just a tensor product of the one dimensional basis.
This basis transformation can efficiently be integrated either in our Horner Scheme for composition, which also shows this separation of single dimensions, or be calculated in shared memory by a set of threads – logically forming a packet of rays – hitting the same polynomial. For few polynomials/voxels hit by many rays, the second approach shows higher frame rates, whereas many voxels, however, are rendered more quickly using the first approach, as the coherence between rays is not as high.

3.3.2 Rendertree

For handling big datasets, knowledge of the underlying data and its location in memory is required. We store all meta information in an octree: Every node stores a conservative estimate of the minimum and maximum value appearing in its subtree to quickly

identify regions, which can not contain the isosurface. During traversal of the octree, this information is used to enable a conservative empty space skipping strategy. Each leaf in the octree corresponds to a group of b^3 adjacent voxels/polynomials, also called brick. b forms a trade-off factor between more efficient empty space skipping and memory requirement of the tree structure.
A fully filled tree, with $b = 1$ needs more than the double amount of memory as the data itself. Even more memory is needed, if the brick's location in memory or a pointer to the children is needed. Therefore, we choose $b = 1$ only for small volumes and increase b for bigger volumes to reduce the storage overhead of the octree structure.
The minimum and maximum values at leaf nodes can be estimated using the convex combination property of the B-spline basis[2]and are propagated towards the root of the tree.
During rendering, we use a GPU-enabled version of the parametric octree traversal algorithm proposed in [RUL00]. To circumvent the usage of an explicit stack in shared memory, the current octree coordinate is stored in a single 32 bit data word. Every three bits encode the chosen subvoxel when descending the tree. Thus, 30 bits suffice to support octrees of depth 10, which is ample for data dimensions beyond 1024^3, if brick dimensions are chosen appropriately.

[2]The global extrema of the polynomial function $f_v(\mathbf{x})$ in the interval $[0,1]^3$ are bound by the extrema of the B-spline influence coefficients $vecIC$:

$$\min\left(f_v(\mathbf{x})\right) \geq \min\left(\mathbf{IC}_1(v), \mathbf{IC}_2(v), \cdots, \mathbf{IC}_{(d+1)^3}(v)\right)$$

$$\max\left(f_v(\mathbf{x})\right) \leq \max\left(\mathbf{IC}_1(v), \mathbf{IC}_2(v), \cdots, \mathbf{IC}_{(d+1)^3}(v)\right)$$

3.3.3 Pipeline

Our adapted rendering pipeline for isosurface rendering enabled by algebraic surfaces includes the following steps:

1. project the ray $\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{v}$ from screen space to object space
2. compute the ray intersection with the octree boundaries
3. traverse the tree checking only these nodes, which can possibly contain a hit
4. synchronize threads as soon a brick containing possible hits has been identified
5. check the brick for a hit
   (a) identify the first/next hit voxel within the brick
   (b) calculate the trivariate polynomial scaled by a constant factor $e \cdot f(x, y, z)$[3]from the B-spline coefficients
   (c) calculate the polynomial along the ray $e \cdot (f \circ \mathbf{r})(t)$
   (d) subtract the isovalue scaled by the factor e extracted from the basis transform
   (e) find the first root position t_{root} of the polynomial $e \cdot f(t) - e \cdot c = 0$[4]
   (f) if a hit is identified, calculate object space hit point and normal, continue at 6
   otherwise, go to next voxel (5a) or, if the whole brick did not contain a hit, continue with octree traversal (3)
6. calculate lighting

3.3.4 Execution Configuration

The execution of CUDA kernels is controlled by setting up a grid of thread blocks. For current graphics cards, a so called warp is formed by 32 threads, which all are provided with the same command stream. This means, if threads within a warp take different branches, all branches, even those which are only taken by one thread, have to be executed. All threads, which did not choose this branch, remain idle during this time. Therefore, it is important to choose a good execution configuration, with as many

[3]The constant factor e is introduced because of the acceleration techniques applied to the basis transformation, see Section 3.3.1.

[4]The zero crossing of the polynomial $f(t) - c$ are equal to the zeros of $e \cdot (f(t) - c)$.

coherent threads as possible. We have chosen to assign one thread per ray and form blocks for 8×8 pixel areas (8×4 pixels correspond to one warp).
Additionally, it is desired to start as little blocks as possible. It would be a waste of resources starting threads for all pixels on screen, if the projection of the objects bounding box can be evaluated easily. Thus, we project the object space bounding box (octree bounds) to screen space and start threads only for the rectangle which is defined by the minimum and maximum coordinates of the projected bounding box corners. Figure 3.2 indicates how many threads have to be started for an arbitrary scene and how fast the whole block can finish.

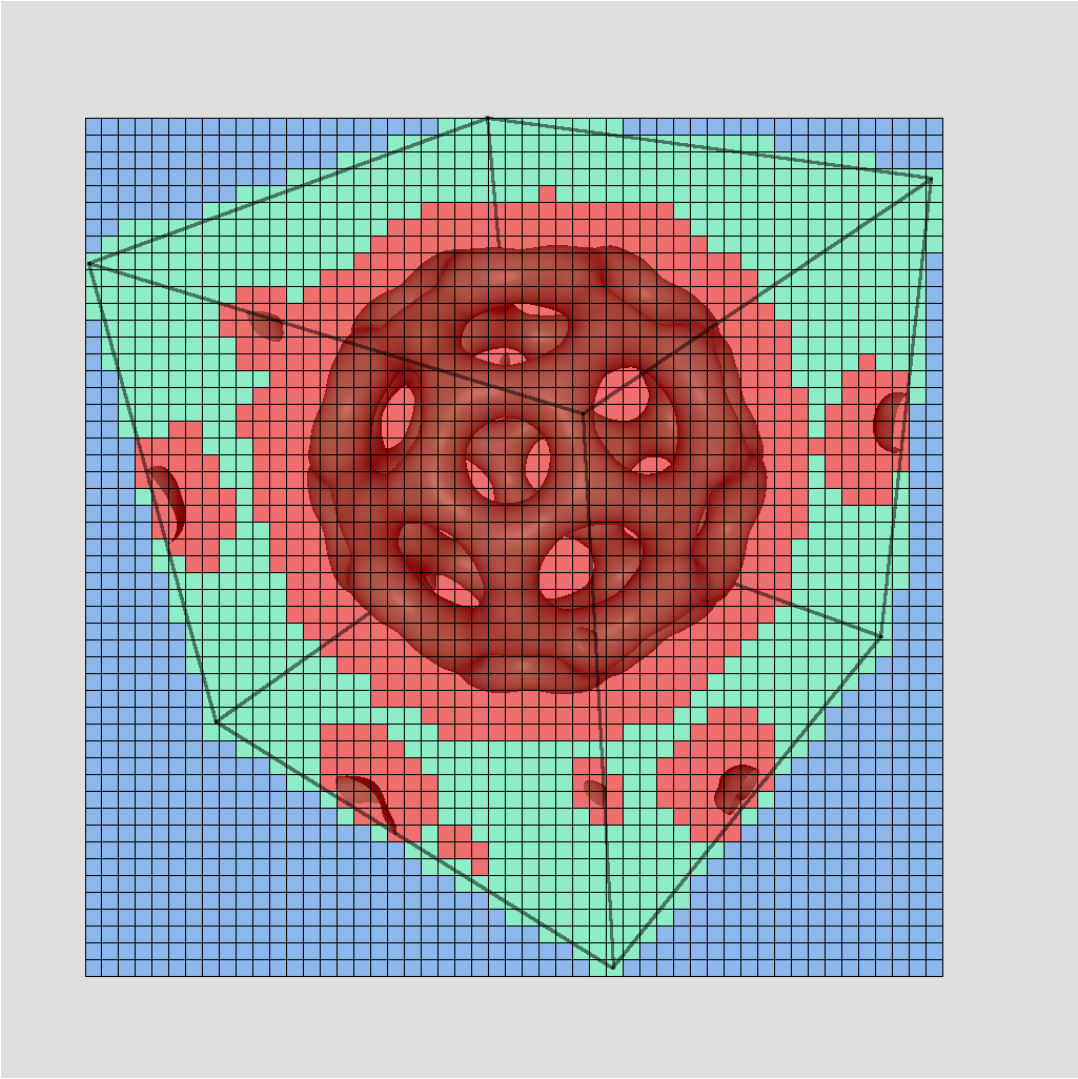


Figure 3.2: Execution configuration for a rendering of the Bucky dataset: one thread is started for each pixel that lies within a rectangle that is defined by the projection of the bounding box on the screen. In this particular example, 752 blocks (29%) finished immediately, 799 blocks (31%) never identified a possible hit and only in 1050 blocks (40%) rootfinding has been performed on a polynomial level. Blue: blocks that finish immediately after failing the bounding box check (step 2 in the pipeline); green: blocks for which no thread identifies a voxel that can possibly contain a hit; red: blocks for which rays exist that need to start rootfinding for at least one voxel along the ray

Chapter 4

Wavelet-based Multiresolution

Contents

When facing bigger datasets, multiresolution techniques often are considered. Especially when a mixed resolution rendering technique can be developed, renderings of big volumetric datasets can benefit the most. Parts being close to camera and therefore being responsible for a big portion of the image, should be rendered in their highest resolution, while for distant parts it is not necessary to include every detail, as the thus excluded detail might only be at sub pixel size. For such small features, reducing the resolution even addresses aliasing as raytracing is only sampling the isosurface on a regular basis and does not consider the extend of the pixel from where the ray originates. In our case, less detail also increases rendering speed: If the resolution is decreased to a coarser level, less but bigger sized voxels need to be rendered. As our method scales approximately linearly with the number of voxels, a coarser representation greatly accelerates the whole rendering.

Especially mixing different resolutions is known to be a complex task. The borders between different resolutions need to be handled with care, as not reveal differences in resolution. For isosurface rendering, this task is even more demanding, as inconsistencies stand out as holes in the surface. One possible choice for building a multiresolution data description are wavelets.

Analyzing the data and building differently sized representation of the data are two different steps in a wavelet framework: The analysis step is called decomposition and can be build from recursively applying filtering and downsampling to the current resolution level of the data. Additionally, to the lower resulting representation, so called detail coefficients are produced that correspond to the information needed to reconstruct the higher resolution from the lower resolution data. The reconstruction step works exactly the other way around: the highest available resolution level is upsampled, filtered and

added to the upsampled and filtered detail coefficients. The result of this step is the next higher resolution representation of the data.
The wavelet decomposition is an offline preprocessing step, which only needs to be done once per dataset and is therefore not time-critical. When only a small set of different resolutions of the data should be available, they can also be precomputed and stored for direct access during rendering. Wavelet reconstruction for a mixed resolution framework needs to be invoked whenever the desired resolution is changed and therefore has to be optimized carefully. The amount of data needed to be able to reconstruct all resolution levels corresponds exactly in size to the original number of data coefficients.
Most proposed isosurface renderers, which use wavelets for data simplification do not consider the type of wavelets during rendering. During reconstruction of resolution levels borders between resolution levels are not considered, they are taken care of in an individual step. This is also the reason why linear interpolation is mostly chosen for these approaches: mixing two resolutions is trivial for linear interpolation, as a set of higher resolution coefficients just needs to 'smear-in' the first set of lower resolution coefficients.
In contrast to such approaches, we haven chosen wavelets which can be directly rendered with our framework: Polynomial spline wavelets [CDF92]. The scaling function of these wavelets is the B-spline basis function. A single B-spline basis function is described as a single coefficient in the volume dataset (see Section 3.3.1 and Figure 4.1). Using the appropriate degree of interpolation enables us to render an exact representation of the scaling function and hence a wavelet function, which is constructed from a superposition of scaling functions. In other words this means: constructing and rendering a higher resolution level with all detail coefficients disabled (no additional information is added), produces exactly the same result as the lower resolution level.
Therefore, the continuity property of B-spline wavelets is transferable directly to renderings spanning multiple levels of resolution. This guarantees that the reconstruction (and hence the isosurface) is continuous and no complex additional border handling between resolutions needs to be considered (for more details see Section 4.4). The non-standard scheme for decomposition and reconstruction [SDS96] is employed in order to obtain a localized influence of a single wavelet coefficient and thus, a useful approximation of the input data at each level.

4.1 Used Wavelets

As volumetric datasets are naturally three-dimensional, a three-dimensional wavelet transformation is required. Three-dimensional wavelets can be constructed from the

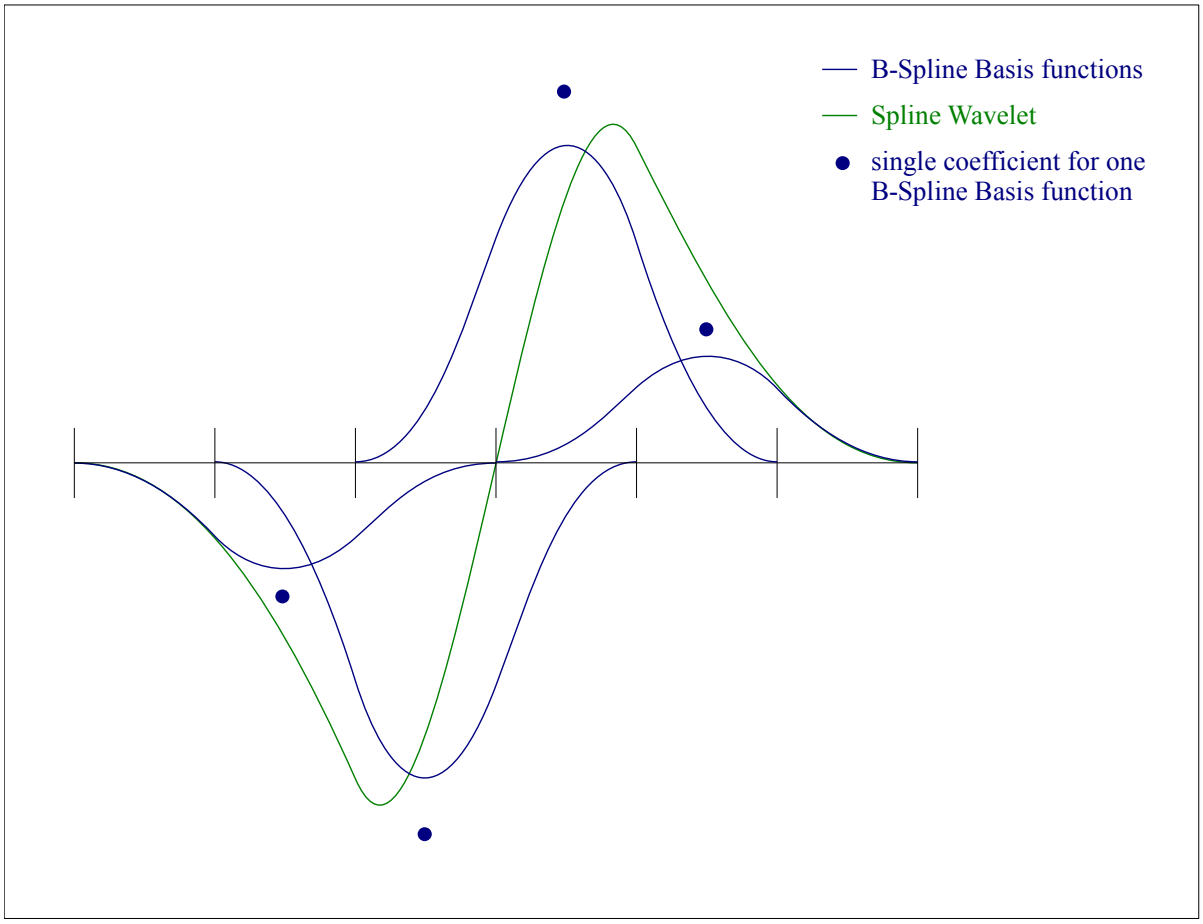


Figure 4.1: The spline wavelet function $\psi_{3,1}$ is constructed from 4 B-spline basis functions and can thus be rendered with 4 coefficients.

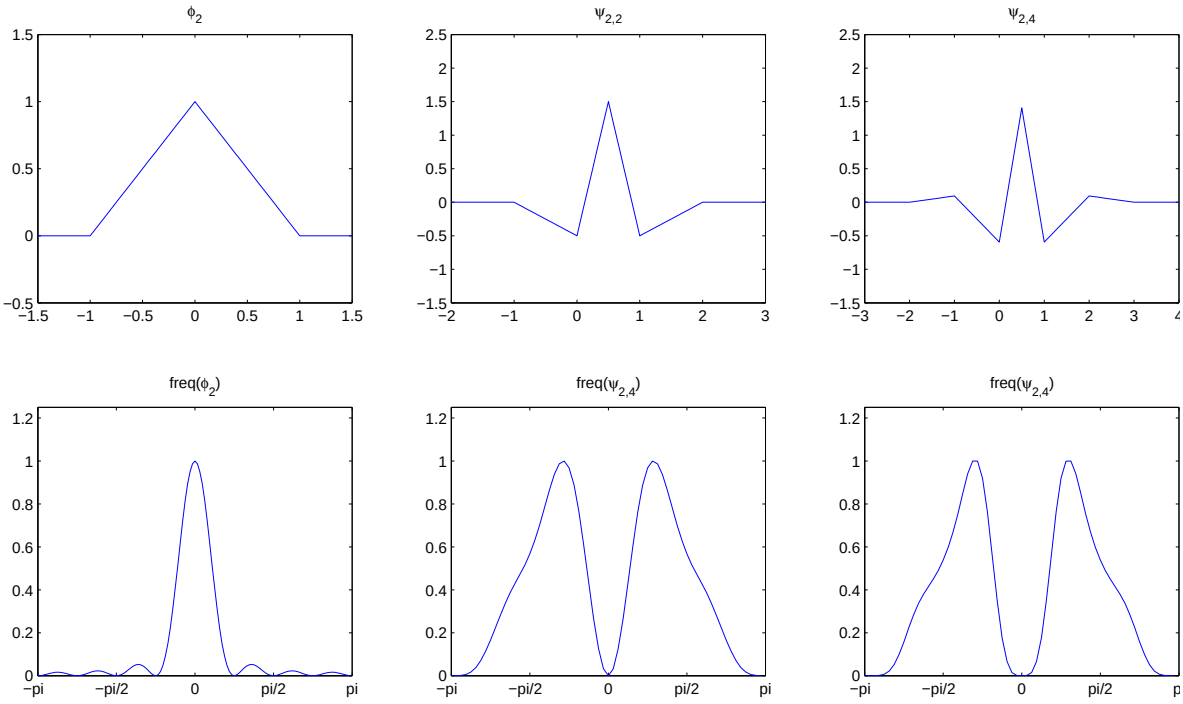
degree	filter size		vanishing	polynomial pieces	
	low	high	moments	low	high
linear ($\psi_{2,2}$)	3	5	2	2	6
linear ($\psi_{2,4}$)	3	9	4	2	10
square ($\psi_{3,1}$)	4	4	1	3	6
square ($\psi_{3,3}$)	4	8	3	3	10
cubic ($\psi_{4,2}$)	5	7	2	4	10
cubic ($\psi_{4,4}$)	5	11	4	4	14

Table 4.1: Size of support of wavelet reconstruction low/high pass filters (one dimensional).

one dimensional wavelet transformation by a simple tensor product. In the one dimensional case one has a scaling function ϕ and a wavelet function ψ. When constructing the three-dimensional functions, one ends up with one scaling function which is essentially the tensor product of three one dimensional ϕ, one along each direction. All other seven possible combinations of tensor products between ϕ and ψ, along all directions, form seven wavelet functions.

One three-dimensional decomposition step thus results in seven times as many detail coefficients as scaling coefficients, whereas every different wavelet has associated an equal number of detail coefficients. The reconstruction step is changed accordingly. The therefore needed three-dimensional filtering is separable, which means that the whole data can be filtered along one dimension after another. This is essential for a fast implementation.

We consider wavelets for tri-linear to tri-cubic B-splines to support all so far available rendering methods. Note that higher orders would also be possible when accepting lower frame rates. We also experimented with different numbers of vanishing moments, whereas we concluded that the least possible number of vanishing moments seems to be the best choice. A higher number of vanishing moments implies a bigger support of the wavelets and thus increases the time needed for reconstruction tremendously and in addition small voxels are more quickly inserted to the data. An overview of the number of vanishing moments for the used wavelets can be found in Table 4.1 and the one dimensional versions of them are depicted in Figure 4.2, 4.3 and 4.4, including a listing of the associated low and high pass filter coefficients used for reconstruction.

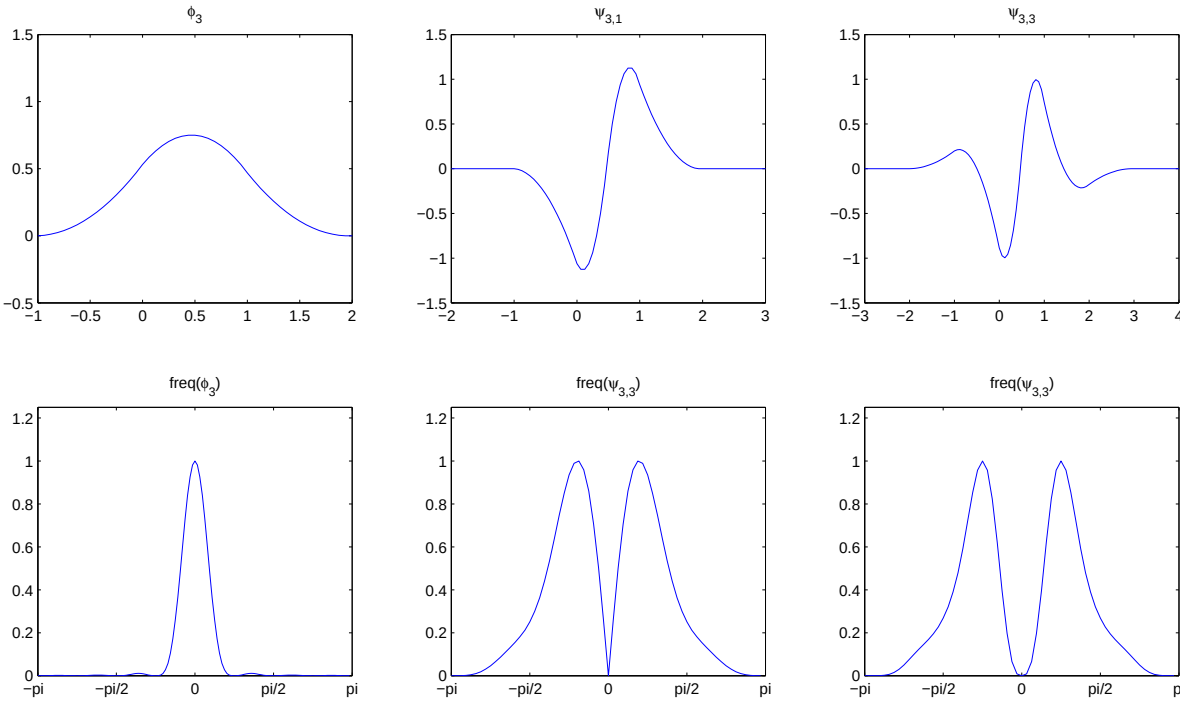


(a) scaling and wavelet functions

linear	
decomposition	
ϕ_2	$\frac{1}{4} \cdot [-1, 2, -1]$
$\psi_{2,2}$	$\frac{1}{8} \cdot [-1, 2, 6, 2, -1]$
$\psi_{2,4}$	$\frac{1}{128} \cdot [3, -4, -16, 38, 90, 38, -16, -4, 3]$
reconstruction	
ϕ_2	$\frac{1}{2} \cdot [1, 2, 1]$
$\psi_{2,2}$	$\frac{1}{4} \cdot [-1, -2, 6, -2, -1]$
$\psi_{2,4}$	$\frac{1}{64} \cdot [3, 4, -16, -38, 90, -38, -16, 4, 3]$

(b) filter coefficients

Figure 4.2: (a) One dimensional plot of the linear scaling function ϕ_2, and two associated wavelet functions $\psi_{2,2}$ and $\psi_{2,4}$ as well as the corresponding frequency coverage of the given mother functions. (b) One dimensional filter coefficients for the used scaling functions and wavelet functions.

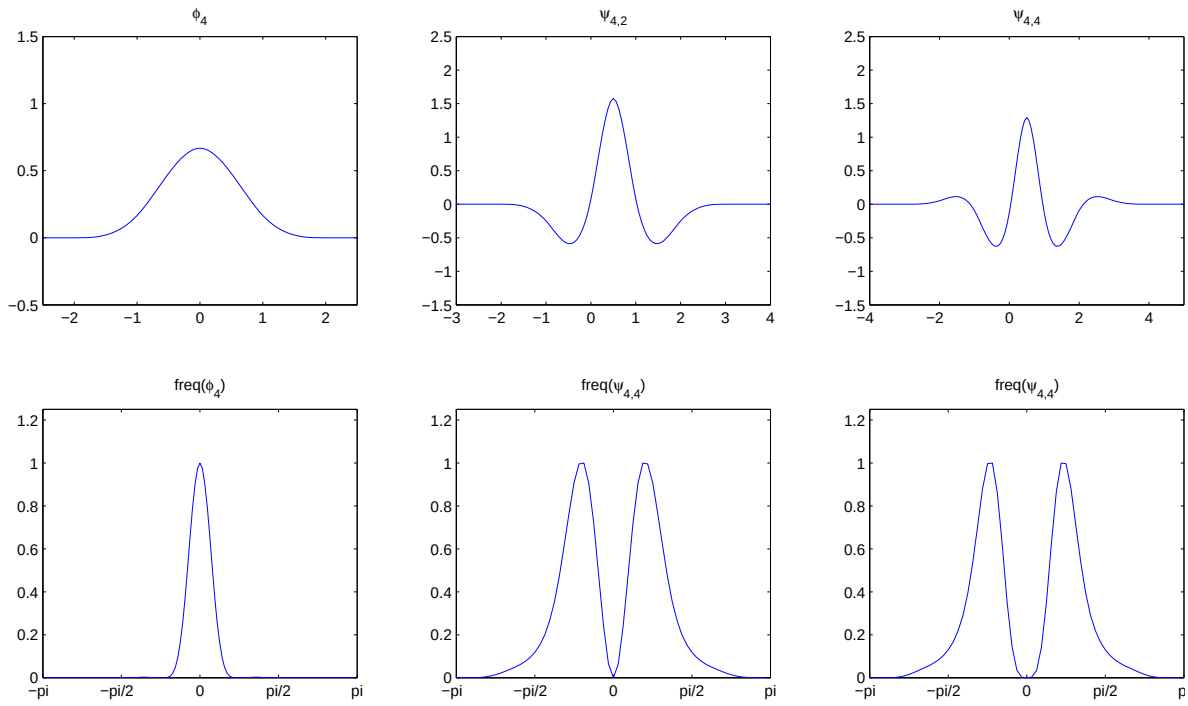


(a) scaling and wavelet functions

square	
decomposition	
ϕ_3	$\frac{1}{8} \cdot [1, -3, 3, -1]$
$\psi_{3,1}$	$\frac{1}{4} \cdot [-1, 3, 3, -1]$
$\psi_{3,3}$	$\frac{1}{64} \cdot [3, -9, -7, 45, 45, -7, -9, 3]$
reconstruction	
ϕ_3	$\frac{1}{4} \cdot [1, 3, 3, 1]$
$\psi_{3,1}$	$\frac{1}{2} \cdot [1, 3, -3, -1]$
$\psi_{3,3}$	$\frac{1}{128} \cdot [-3, -9, 7, 45, -45, -7, 9, 3]$

(b) filter coefficients

Figure 4.3: (a) One dimensional plot of the linear scaling function ϕ_3, and two associated wavelet functions $\psi_{3,1}$ and $\psi_{3,3}$ as well as the corresponding frequency coverage of the given mother functions. (b) One dimensional filter coefficients for the used scaling functions and wavelet functions.



(a) scaling and wavelet functions

cubic	
decomposition	
ϕ_4	$\frac{1}{16} \cdot [1, -4, 6, -4, 1]$
$\psi_{4,2}$	$\frac{1}{32} \cdot [3, -12, 5, 40, 5, -12, 3]$
$\psi_{4,4}$	$\frac{1}{256} \cdot [-5, 20, -1, -96, 70, 280, 70, -96, -1, -20, -5]$
reconstruction	
ϕ_4	$\frac{1}{8} \cdot [1, 4, 6, 4, 1]$
$\psi_{4,2}$	$\frac{1}{16} \cdot [-3, -12, -5, 40, -5, -12, -3]$
$\psi_{4,4}$	$\frac{1}{128} \cdot [5, 20, 1, -96, -70, 280, -70, -96, 1, 20, 5]$

(b) filter coefficients

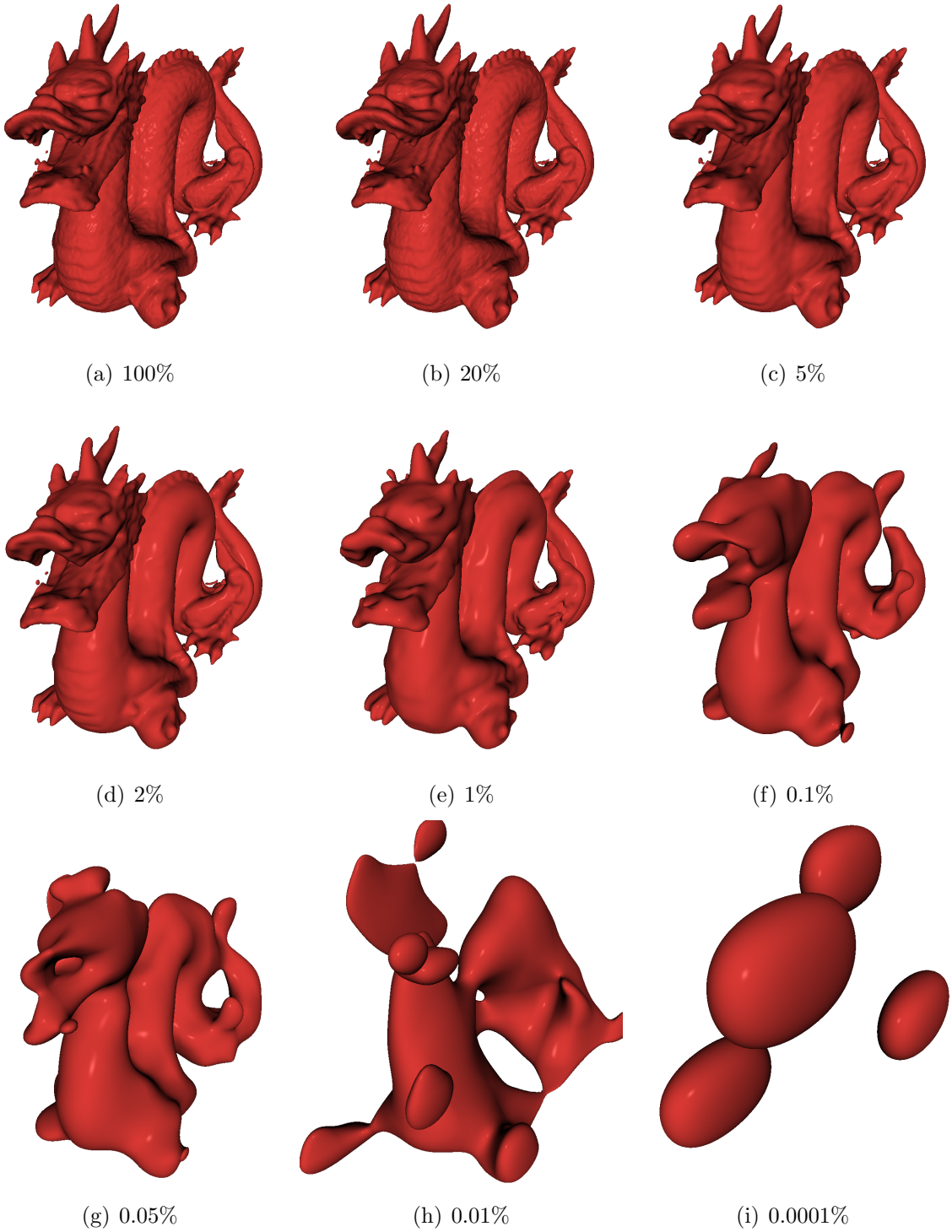
Figure 4.4: (a) One dimensional plot of the linear scaling function ϕ_4, and two associated wavelet functions $\psi_{4,2}$ and $\psi_{4,4}$ as well as the corresponding frequency coverage of the given mother functions. (b) One dimensional filter coefficients for the used scaling functions and wavelet functions.

4.2 Selection of Wavelets Coefficients

The selection of the wavelet coefficients used for reconstruction strongly depends on the scenario. Traditional wavelet applications are noise reduction, for which mostly high frequency bands are discarded, and compression, which discards coefficients with low magnitude. Although our work does not focus on compression, we want to show that our rendering framework can also use wavelets for compression, which is one of the most common applications for wavelets. A lossy three-dimensional data compression can be carried out, when selecting the biggest coefficients only (see Figure 4.5).
Although there are less coefficients used to describe the dataset in the compression scenario, this does not essentially aid our rendering framework for increase in frame rates. At a finer level, big coefficients are not removed, while still producing a bigger number of small voxels, which need more time to render.
Considering the overlap of adjacent wavelets, which is given by their support, one can e.g. remove 99% of wavelet coefficients from the last level and still not reduce the number of voxels needed for rendering: The support for one of the used tri-cubic wavelets is 7^3, which means activating exactly one wavelet coefficient adds 7^3 voxels to the dataset. If the distance between activated wavelet coefficients is chosen with exactly the size of the support along each direction, a whole level of voxels is created for only 1% of activated data. In this case, one could also render the whole dataset with the same number of voxels (thus with the same framerates), but in full detail.
To avoid that situation from happening, as little transitions as possible between different resolution levels should be included when advancing from neighbor to neighbor within the dataset. This idea is also accompanied by a view dependent selection of wavelet coefficients. Regions close to the camera should be present with their full resolution, whilst the resolution of distant regions can decrease gradually (see Figure 4.6). In particular, we collapse an octree node if $\frac{s}{c} < \varepsilon$, where s is the node diameter, c is its distance from the camera, and ε is a user-defined threshold. This approach is very simple and there exist more sophisticated approaches which have been proposed for multiresolution meshes, e.g. based on silhouettes and specular highlights [XV96, Hop97]. These ideas are out of scope for this work.

4.3 Integration

To support a multiresolution description during rendering, the current description of the data needs to be embedded into our octree structure, which is used for rendering. A chosen approximation of the original data can be seen as linear combination of a subset

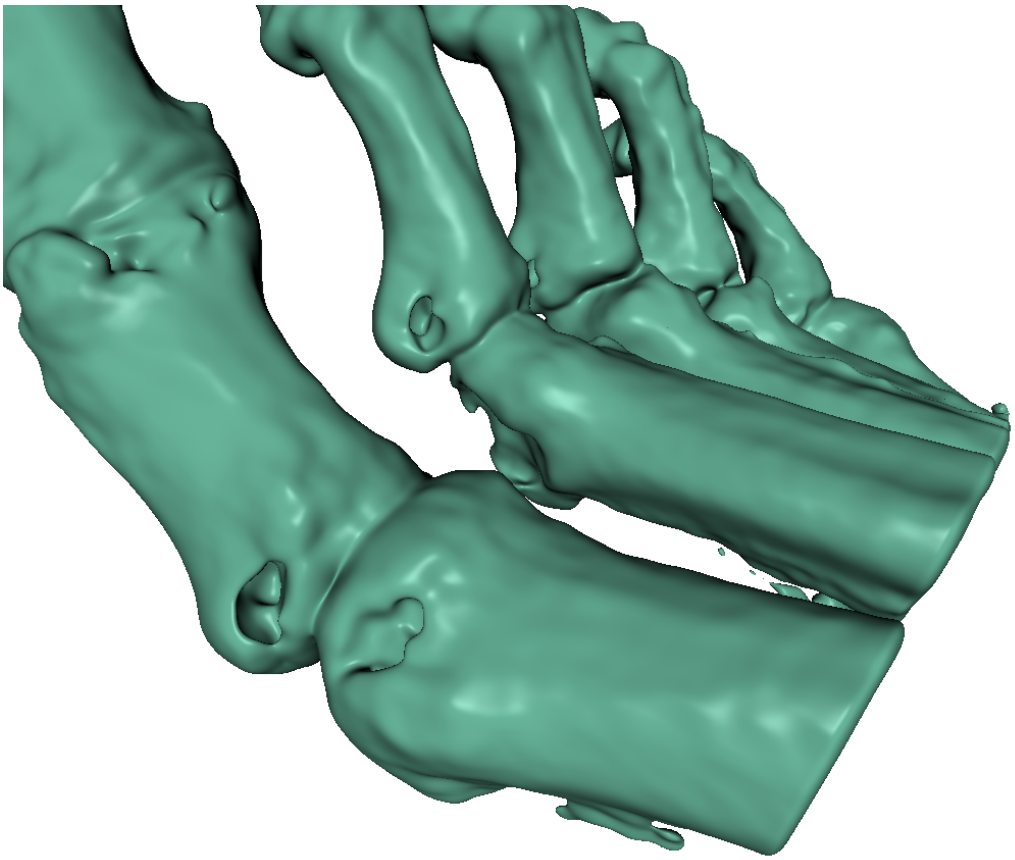


(a) 100% (b) 20% (c) 5%

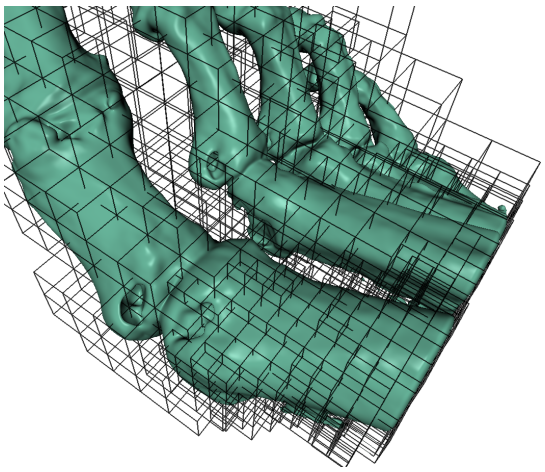
(d) 2% (e) 1% (f) 0.1%

(g) 0.05% (h) 0.01% (i) 0.0001%

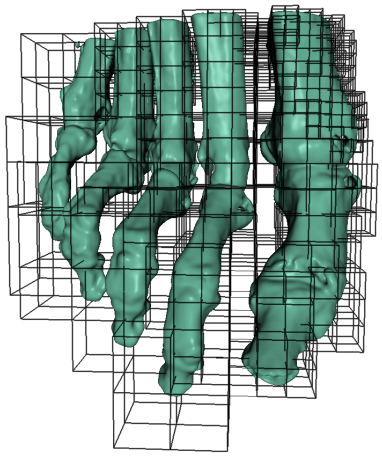
Figure 4.5: Dragon dataset: wavelet reconstruction involving data from different scales, with coefficient selection based on greatest magnitude. The images demonstrate that a small subset of the whole volume dataset already yields good results.



(a) scene with view-dependent LOD



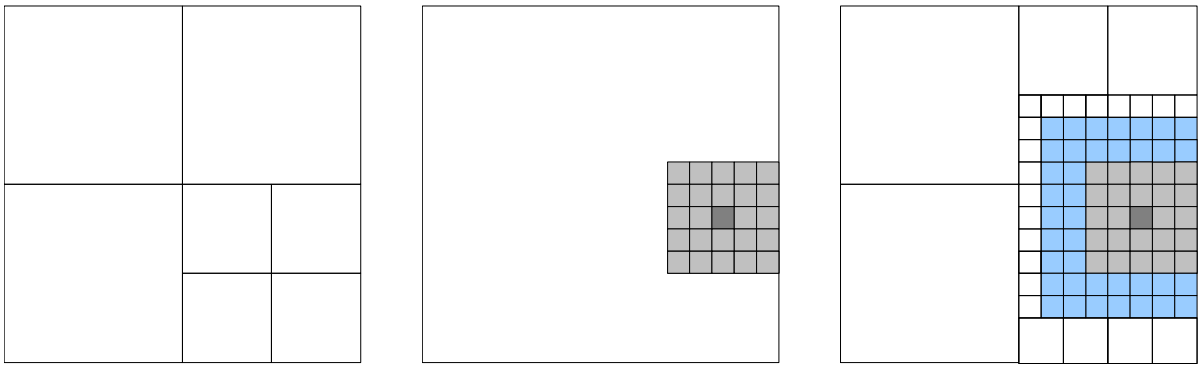
(b) octree nodes superimposed



(c) different view of (b) with frozen LOD

Figure 4.6: View-dependent multiresolution scene (a) spanning several levels of the wavelet hierarchy (indicated by the black cubes in (b) and (c)).

of the available basis functions (i.e. wavelets), weighted by the corresponding detail coefficients (obtained by the decomposition step). The levels of the octree form a set of nested vector spaces (much like the wavelets themselves [SDS96]). Hence, we can expand octree nodes (starting from the root) as needed, until the vector space spanned by the currently expanded nodes is powerful enough to hold the desired approximation. Note: the octree itself just keeps track of polynomial pieces and their organization in memory, whereas the underlying data itself is given as a set of scaled and dilated B-spline functions, which are used to represent exactly the chosen subset of activated wavelets (i.e. for which the detail coefficients are not set to zero).



(a) before insertion (b) wavelet (c) after insertion

Figure 4.7: Expanding the spline octree (a) for insertion of a wavelet (b), resulting in a new octree (c) which can represent the newly added detail; dark grey: single activated detail coefficient, light grey: influenced spline coeffcients, blue: additional polynomial pieces needed for rendering the spline coefficients (light grey)

Figure 4.7 illustrates a single step of this procedure. However, a straightforward implementation would be inefficient for two reasons. First, distributing spline coefficients within a wavelet's support (light grey square in Figure 4.7(b)) to all influenced data coefficients involves a scattering memory access pattern. Though possible on recent GPUs by means of atomic operations, such an approach suffers from performance penalties due to bus locking. Instead, we proceed by traversing the hierarchy and gathering contributions of wavelets, which influence the data coefficient under consideration. This entails another problem: the processing step for a single data coefficients needs to know whether the surrounding wavelets are activated or not. Again, this either involves a scattering access pattern or the error metric needs to be evaluated for all surrounding wavelets for the computation of a single data coefficient. We will deal with this problem in the Section 4.4.

Nevertheless, the problem gets even more demanding for three-dimensional data. The number of wavelet coefficients influencing a single node grows cubicly with the support of the one dimensional wavelet (e.g. 7^3 non-zero influence coefficients for the tri-cubic case). More efficiently, one can use the separability of the B-spline wavelet basis and sequentially perform the transformation in z-, y-, and x-direction. This reduces the number of influence wavelet coefficients to three times the number needed in the one dimensional case (e.g. 3×7 coefficients have to be considered in the tri-cubic case).

4.4 Efficient Wavelet Reconstruction

Concerning bigger data sets, the wavelet reconstruction can get rather time consuming, thus we try to use different speedup strategies. In general, frame-to-frame coherence can be exploited if camera movements are not too drastic. When a higher resolution becomes necessary, wavelet reconstruction does not need to start at the root level. The highest available resolution level can be chosen as a starting point. When advancing wavelet reconstruction towards higher resolution, we also keep low resolution data in memory, therefore reduction of the desired level of detail is trivial.

Still, the most important point is the granularity at which the wavelet reconstruction is carried out. Although possible, it is not feasible to keep track of the activation and the influence of every single wavelet (see Figure 4.7 and Section 4.3). Especially communication of states and evaluation of the view dependent error metric form an undeniable overhead, when implemented on a per spline coefficient basis. Furthermore, only tasks which can be subdivided into a sufficient number of subtasks of identical structure can be implemented with maximum performance on SIMD hardware, which is only possible, if a bigger set of adjacent wavelets display the same activation pattern. We therefore operate on chunks of $k \times k \times k$ nodes (we choose $k = 8$) and activate a whole chunk of wavelet coefficients at once (such that all nodes within the chunk can be processed efficiently in a similar way).

An activated chunk needs to be reconstructed fully. Chunks which are not activated, do not add any additional data to the reconstruction and their detail coefficients are thus set to zero. If all detail coefficients contributing to a chunk are set to zero, the output corresponds exactly to the data at the next lower resolution level, sampled with twice as many points along each dimension. Therefore, we can omit reconstruction of all chunks, which are neither activated nor needed to maintain continuity. Since the support of some wavelets in an activated chunk extends into neighboring chunks, these neighbors also have to be reconstructed and used for rendering to maintain continuity. As the polynomial description of a voxel is constructed during rendering from its neighboring

data entries and resolutions are only switched at chunk borders, we also have to compute one or two data coefficients in high resolution chunks next to the resolution borders. For a detailed description on which chunks need to be activated see Figure 4.8.

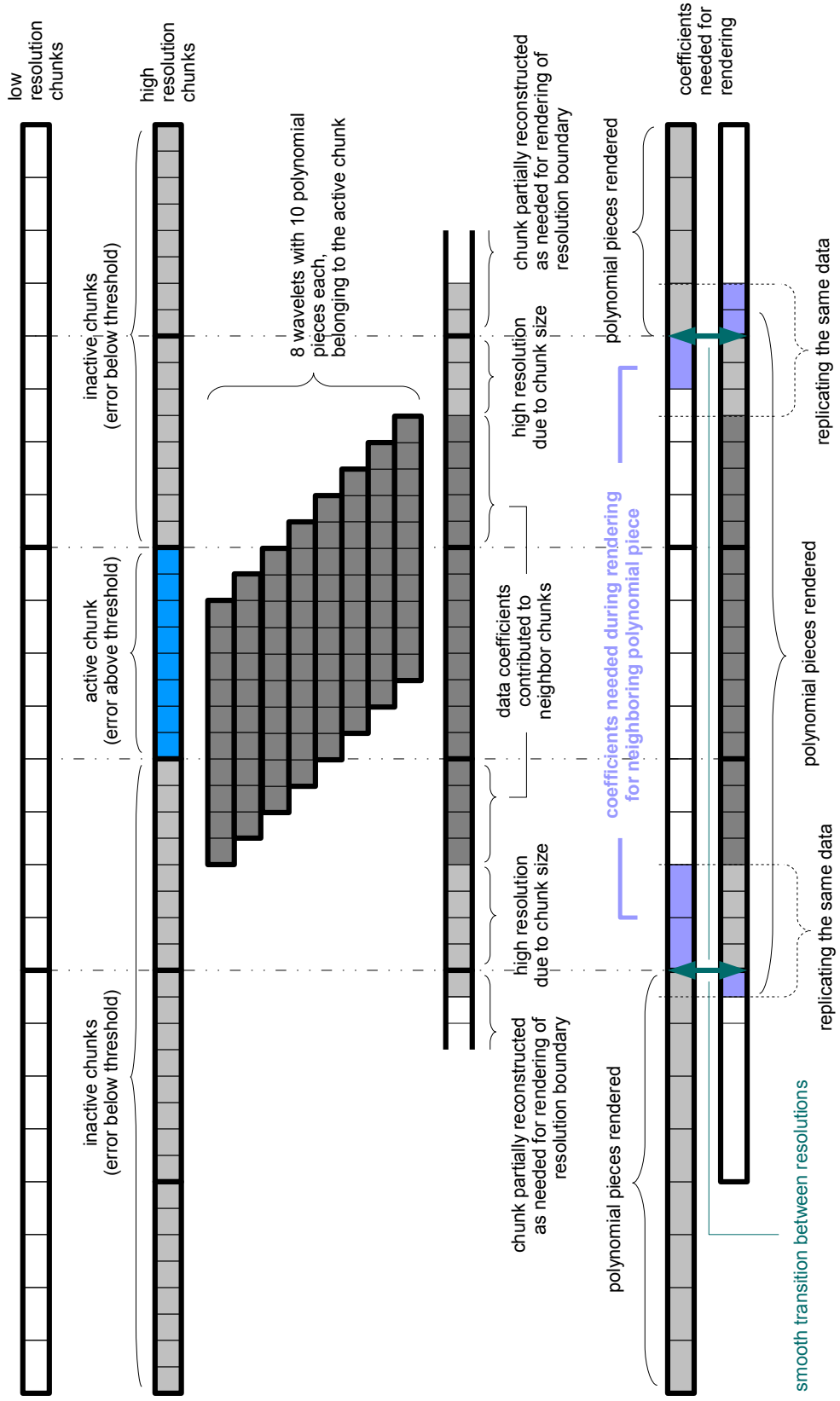


Figure 4.8: Influence of an activated chunk on data coefficients, surrounding chunks, polynomial pieces and transition between resolution levels.

Three different CUDA kernels are used for the reconstruction of a single resolution level. One kernel is used for the convolution along each dimension. Whereas the second and the third kernel's tasks only consist of convolution and are equal apart from different operation directions, the first kernel also covers additional responsibilities, namely:

- evaluating the error metric for the chunk under consideration
- evaluating the error metric for surrounding chunks to correctly handle the influence of the possibly activated wavelets within these chunks
- storing information of whether this chunk needs to be reconstructed or not
- updating the octree structure to correctly render the resulting approximation of data (i.e. setting the right references to data locations for all nodes coinciding with the current chunk)
- loading the lower resolution data
- loading detail coefficients if the chunk is activated

For more details on all three kernels, their execution and remapping of thread duties see Algorithm 1, 2 and 3.
The CUDA kernels for all three dimensions each consist of blocks of $k \times k$ threads. Each thread computes the one-dimensional convolution with the wavelet filter kernels (low pass and high pass). One thread block is responsible for a beam of $k \times k$ voxels along the current convolution direction. Reconstruction is applied to one chunk after another and deactivated chunks are simply skipped (see Figure 4.9). At each reconstruction step it is assured that all needed data resides in register space before the convolution is carried out. Data fetched for the reconstruction of the previous chunk is reused, which also enables storing the altered data at exactly the same memory location as the input data. Thus, a single read instruction from global memory per sample is sufficient to compute the convolution. Fast memory transactions can be guaranteed, if threads access a continuous region in memory (coalescing). A chunk laid out for access along a certain dimension shows suboptimal access patterns along a different dimension. Therefore, data is reordered in shared memory, before being written back to global memory. Banking conflicts in shared memory are avoided by additional padding, much like described in [HSO07]. This method is only applicable for conflict free access patterns within two dimensional data. Due to the execution order of $z \rightarrow y \rightarrow x$ only data vectors between the x and y dimension (second and third kernel) have to be interchanged and therefore this simple padding method can also be applied here.

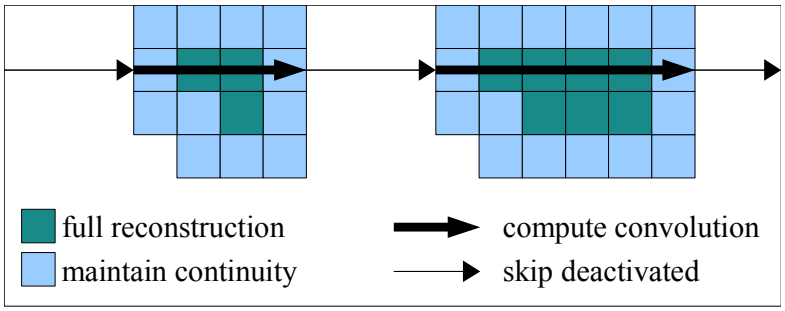


Figure 4.9: Processing scheme for selective wavelet reconstruction, the arrows indicate the execution of a single thread block in the CUDA kernel. Boxes do not refer to octree nodes, but to chunks of $k \times k \times k$ nodes as explained in Section 4.4.

Algorithm 1 Efficient wavelet reconstruction kernel z part 1

each thread computes its 2D-thread id (t_0, t_1)
while not all chunks in threadblock's associated beam have been checked **do**
  assign all threads to individual unchecked chunks
  for $x = -2 \ldots 2$,$y = -2 \ldots 2$ **do** {neighborhood of the chunk}
    c_0 = lower resolution chunk active
    c_1 = error criteria of the chunk fulfilled
    c_2 = isovalue within min/max range of associated octree node
    if c_0 and c_1 and c_2 **then** {chunk activate}
      store a bit indication chunk's activation type in shared memory
    end if
  end for
  store the activation of the current chunk in global memory
  synchronize
  for $z = -2 \ldots 2$ **do** {z neighborhood of the chunk}
    gather chunk activity from shared memory and update current chunks activity accordingly
  end for
end while
synchronize

Algorithm 2 Efficient wavelet reconstruction kernel z part 2

every thread is now responsible for a one dimensional data vector
for $i = 0 \dots N/k$ **do** {entire range of 1D convolution}
  load chunk activity from shared memory (broadcast)
  if chunk i active or neighbor active **then**
    load low pass data from lower resolution
    if chunk i active **then**
      load detail coefficients
    end if
    if left chunk $(i-1)$ active **then**
      load detail coefficients of left chunk
    end if
    if right chunk $(i+1)$ active **then**
      load detail coefficients of right chunk
    end if
    for $x = 0 \dots k-1$ **do** {compute convolution}
      each thread performs the convolution (i.e. computes the dot product of the fetched data and filter kernels)
      each thread stores the item in global memory(coalesced access) (t_0, t_1, z) (conflict-free access)
    end for
  end if
end for

Algorithm 3 Efficient wavelet reconstruction kernel y/x

reserve 3 chunks of data in register space: `pre`, `main`, `post`
keep track of loaded chunks $n_{loaded} = 0$
for $i = 0, \ldots, N/k$ **do** {entire range of 1D convolution}
  if chunk i active **then**
    if $n_{loaded} == 2$ **then** {`post` and `main` can be reused}
      `pre` = `main`
      `main` = `post`
    else if $n_{loaded} == 1$ **then** {`post` can be reused}
      `pre` = `post`
      load chunk `main` (i) from global memory (coalesced access)
      reorder chunk `main` using shared memory to grant every thread with the right data (conflict-free access)
    else if $n_{loaded} \leq 0$ **then** {none to reuse}
      load chunks `pre` ($i-1$), `main` (i) from global memory (coalesced access)
      reorder chunks `pre`, `main` using shared memory to grant every thread with the right data (conflict-free access)
    end if
    load chunk `post` ($i+1$) from global memory (coalesced access)
    reorder chunk `post` using shared memory to grant every thread with the right data (conflict-free access)
    $n_{loaded} = 2$
  else
    $n_{loaded} = n_{loaded} - 1$
  end if
  for $j = 0 \ldots k-1$ **do** {compute convolution}
    each thread performs the convolution (i.e. computes the dot product of the fetched data and filter kernels)
    each thread stores the result in shared memory (conflict-free access)
  end for
  for $j = 0, \ldots, k-1$ **do** {reorder results}
    each thread fetches a single item from shared memory (conflict-free access)
    each thread stores the item in global memory(coalesced access)
  end for
end for

Chapter 5

Code Optimization

According to [LNOM08] high arithmetic intensity is necessary within the context of CUDA, as full utilization of computational resources naturally is essential for high performance. If the equations in Section 3.2.3 are implemented in a straightforward fashion, the produced code includes several nested loops with the trip count being known in advance. The design of the CUDA compiler is as such as to detect static program flow constructs and regard these within the optimization of the code (i.e. unroll loops). However, this is not realized in this particular case, because inner loop limits depend on outer loop indices.
Although the code demonstrated in Listing A.1 has already been unrolled manually for all three dimensions to demonstrate the needed steps, the `convolution_inplace` calls contain a loop depending on the loop index of the current dimension. Therefore the compiler also fails in unrolling for these constructions. The problem gets even more severe, if an innermost loop can not be unrolled, as this also prohibits outer loops from unrolling.
Hence, the resulting program performs dynamic looping, which, however, is very inefficient for two reasons. First, a certain overhead is associated with loop counter manipulation and conditional branching. Second, and most important, since indirect addressing is not supported in register space, the indexed quantities like polynomial coefficients are stored in local memory, which has significantly higher latency and lower bandwidth than the on-chip register file. To overcome this problem, we added a code transformation step to our build framework which is invoked before the CUDA compiler. We thereby "flatten out" the entire static control flow of the program, which generates 81 times faster code for the tri-cubic composition and subsequent root finding than with dynamic branching. Without this optimization, this part of the algorithm is the bottleneck of the system, consuming $88 \pm 5\%$ of total frame time tri-cubic rendering. After optimization, it only contributes $30 \pm 5\%$ to the frame time and overall performance has been improved by a factor of approximately 17. The remaining $70 \pm 5\%$ consist mainly of traversal and data transfer time.
The developed preprocessor consists of three steps:

1. Parsing steps:

- load input files
- generate C++ code for step 2 from marked regions
- pass through standard C++ code to remain unchanged

2. Compile and Run step:
   - compile the produced intermediate file
   - run the generated executable
   - produce a new source file containing the passed through original code and the generated code
3. include the generated file to the project

The parsing step 1 understands the following pragmas and sets a preprocessor definition to enable to programmer to write code that can also be compiled without the preprocessing step.

- `meta_refer_to_input_file` turns references to the original source on and off
- `meta_begin` , `meta_end` encloses code that is to be compiled and executed by the preprocessor
- `meta_early_begin` , `meta_early_end` encloses code that can also include `#include` statements and function definitions
- `meta_include` includes files that are compiled and executed by the preprocessor
- `meta_parse` opens the given file, parses it and adds the resulting output to the current parsing location
- ${, } inline synonym of `meta_begin` and `meta_end`

Our preprocessor would convert the statement:

```
1  float out = 0.0f;
2  #pragma meta_begin
3  for(int i = 0; i < 3; ++i)
4  for(int j = 0; j <= i; ++j)
5  {
6  #pragma meta_end
7          out += in[${i}][${j}];
8  #pragma meta_begin
9  }
10 #pragma meta_end
```

to an unrolled:

```
1 float out = 0.0f;
2   out += in[0][0];
3   out += in[1][0];
4   out += in[1][1];
5   out += in[2][0];
6   out += in[2][1];
7   out += in[2][2];
```

This framework offers a nearly unlimited amount of options, as it does not only support loop unrolling, but also C++ code, which is executed at compile time. We also use the framework to easily support different interpolation kernels, switch methods for octree traversal or hardcode wavelet coefficients into the GPU code.

Chapter 6

Results

Contents

This chapter is split into three parts: Firstly we will show results for algebraic surface rendering and compare our approach to a dedicated algebraic surface renderer [RS08]. We will proceed with isosurface rendering of volume datasets and compare our approach to [HSS+05], which is – as far as we know – still the reference for isosurface rendering of volume datasets. Finally, we will consider the performance of our wavelet reconstruction strategies in comparison to rendering times and show the influence on image quality. All performance measurements were carried out on an Intel Core i7 and a single NVIDIA GeForce 285 running on both Microsoft Windows Vista and Linux.

6.1 Frustum Form vs. Our Approach

[RS08] showed that it is possible to reduce the complexity of per ray calculations for multivariate to univariate composition, as needed in algebraic surface raycasting, by extracting common computations and pre-calculate them on the CPU. The polynomial describing the data is sampled along the frustum planes and plugged into a tensor product equation. This equation is efficiently solved by a sequence of matrix multiplications and leads to the Frustum Form that is transmitted to the GPU. The transition from Frustum Form to Bernstein coefficients describing the polynomial along the ray, involves evaluations only along two coordinate axis and can therefore be carried out efficiently. [RS08] were kind enough to provide us with their original implementation, thus enabling us a comparison to our work.
Figure 6.1 shows a comparison for different view port sizes as well as different polynomial degrees. One can clearly see that our approach outperforms the Frustum Form, especially for lower polynomial degrees. In these cases our composition scheme (see Section 3.2.3)

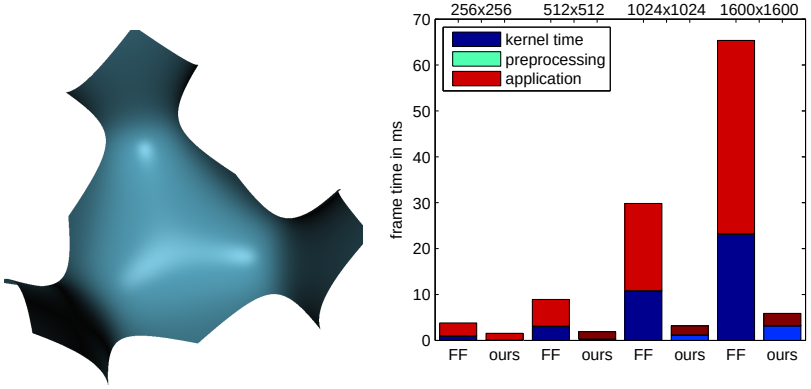
Stage	tri-linear	tri-squared	tri-cubic
Setup and Load	8.390	0.903	0.187
to Object Space	6.936	0.743	0.141
BBox Check	6.090	0.676	0.139
Composition	3.941	1.823	1.029
Rootfind	71.017	93.551	97.489
Evaluate	1.546	1.732	0.972
Lighting	2.079	0.571	0.042

Table 6.1: Time (in %) spent in each stage of our algebraic surface rendering pipeline. One can clearly see that our composition step is efficient in comparison to the time needed for the root finder ($6(d+1)$ iterations of our [LR81] implementation).

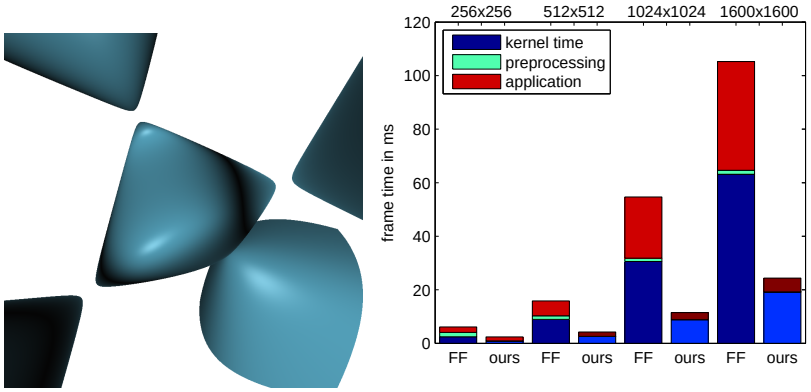
as well as the step from Frustum Form to polynomial coefficients along the ray involve a small amount operations. We believe the main reason for our approach outperforming the Frustum Form is to be found in optimization. For increasing degree and view port size, the Frustum Form approach is getting closer to our approach, which reflects that the Frustum Form should theoretically be faster for dedicated algebraic surface rendering.
Another important point is the preprocessing step that needs to be calculated and transmitted to the GPU for the Frustum Form, whereas our approach does not need any additional CPU to GPU communication. As one can see, this preprocessing step (turquoise) is slowing down the whole approach only if the algebraic surface is responsible for a small screen area or in this case small view port. This demonstrates that the Frustum Form is not well suited for rendering many algebraic surfaces as we carry it out for volume datasets. When rendering a big number of voxels, a single voxel is only responsible for a small portion of the screen. Therefore, only few rays hit this voxel and a preprocessing step like the one used in the Frustum Form approach, carried out for every single voxel would simply be too slow.
Table 6.1 shows how much time is spent on each stage of our algebraic surface rendering pipeline (see Section 3.2). The table demonstrates that our composition step is efficient enough to shift the bottleneck to the root finding phase when rendering a single algebraic surface. Note that the same relations do not hold for rendering volume datasets.

6.2 Tri-cubic Texture Sampling vs. Our Approach

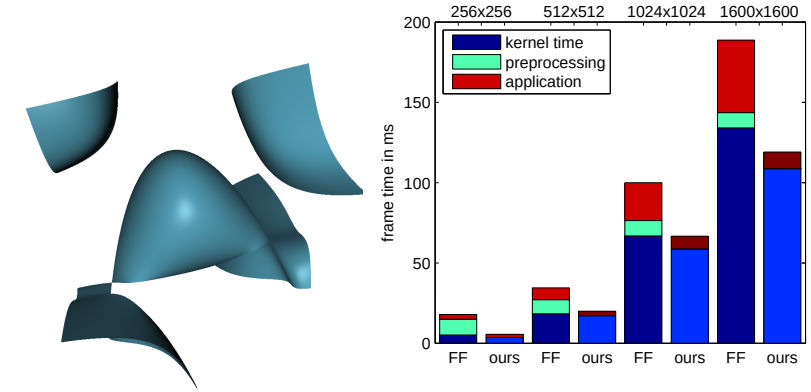
Efficient tri-cubic texture sampling is the key for interactive isosurface rendering as proposed in [HSS+05]. Simple tri-cubic texture sampling is enhanced by hardware support,



(a) tri-linear: $x + y + z + xyz - 1$



(b) tri-square: $4x^2 + 4y^2 + 4z^2 + 16xyz - 1 + x^2y^2z^2$



(c) tri-cubic: $x^2 + y^2 + z^2 - x^3y^3z^3 - x^2y - 1$

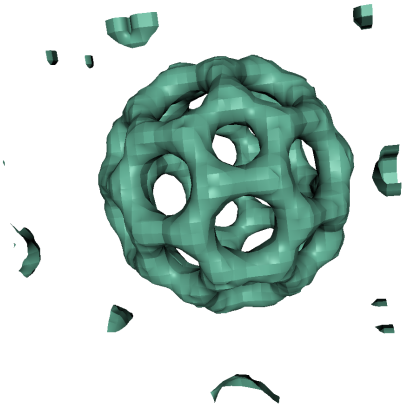
Figure 6.1: Comparison between the frustum form (FF) approach [RS08] and our implementation for raycasting a single algebraic surfaces. Both approaches were configured to work with $6(d+1)$ iterations of an [LR81] rootfinder.

using tri-linear texture sampling as proposed in [SH05]. Using this technique, a tri-cubic texture lookup is constructed from eight tri-linear texture lookups, whereas their texture coordinates are inferred from the tri-cubic sample position and have to be weighted and combined accordingly. This strategy can not only be used for tri-cubic interpolation, but also for the calculation of derivatives of the tri-cubicly interpolated data.
Their isosurface renderer does not explicitly work on any polynomials. Samples are simply drawn along each viewing ray, until a transition from below isovalue to above isovalue or vice-versa is found. This limits the position of the first root to an interval equal to the step length used for drawing samples along the ray, which is further refined in a binary manner: A sample is drawn exactly in the middle of the last two samples. Using the same idea as before the interval containing a transition from below isovalue to above is further refined. This step is simply called hitpoint refinement.
The chosen step length is crucial for both, an exact result as well as fast rendering. For best performance, the step length should be chosen as high as possible, depending on the maximal frequency occurring in the data, as not to miss any root close the silhouettes of the surface. This also leads to the idea of adaptive sampling, whereas different step length are used depending on how close the current sample is to the isovalue. For some data sets and especially during interaction, one can choose step lengths greater than one voxel.
Our approach, on the other hand, needs to check every voxel that can possibly contain a hit – not discarded by empty space skipping. We can also benefit from low frequency data by using the wavelet transformation (see Section 4) to reduce the number of small voxels. Omitting the time needed for traversing the octree data structure, the rendering performance of our implementation scales linearly with the number of tested voxels.
Another important point is empty space skipping. [HSS+05] constructs proxy geometry from the voxel dataset, which equals the union of the boundaries of all voxels that can possibly contain the isosurface. This geometry needs to be rendered twice. During the first pass only back-faces are drawn to the framebuffer. The second pass is only carried out for front faces, which provides the starting point for all rays. The whole ray traversal as described above is carried out in the fragment shader of the second pass. The framebuffer of the first pass is bound as a texture to define the exit point of each ray. The proxy geometry is efficiently constructed only if the isovalue has been altered. This strategy results in approximately the same volume that needs to be tested, as yielded by our octree structure. The advantage of using this constructed proxy geometry is the fact that it can efficiently be applied to the rendering process as the rasterizer converts it into entry and exit points for ray traversal. Our approach, on the other hand, does not need a construction step after isovalue changes and is not limited to a single entry

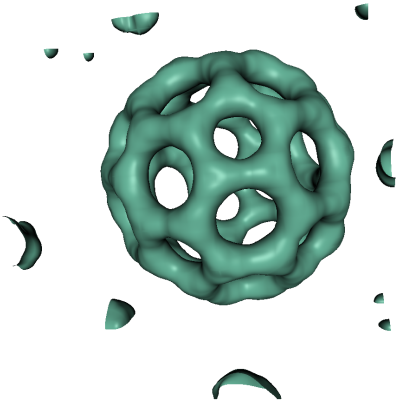
and exit point. Imagine a dataset of a human head, where the user is interested in its bone structure. An empty space skipping strategy relying only on a single entry and exit point, will lead to a ray segment that is active throughout the whole head. Handling such scenarios with bounding geometry involves techniques like depth peeling or more advanced methods [KGB+09]. Our octree based method, however, will skip the volume inside the skull automatically, as the minimum and maximum estimates stored in the octree are evaluated for every node during rendering.
We tested our approach with various examples, showing interactive frame rates for large datasets (see Table 6.2 and Figures 6.2, 6.3 6.4, 6.5). The images demonstrate the effect of different reconstruction filters. For the chosen lighting setup, tri-squared interpolation already yields very good results. Every figure also contains a rendering which shows all active bricks as to demonstrate the number of voxels which have to be considered for rendering. We compared our approach to our implementation of fast tri-cubic sampling using the same octree structure for empty space skipping. For the comparison we tried to configure the fast tri-cubic sampling implementation to meet good visual quality (see figures). To make a fair comparison possible, all these performance measurements have been carried out with the original data and multiresolution reconstruction disabled. As shown in Table 6.2, tri-cubic sampling outperforms our method in terms of rendering speed. On the other hand, tri-cubic texture sampling shows considerable artifacts due to the limited precision of the texture interpolation hardware (9 bits [NVI08]). In volume rendering applications and in particular for tasks such as virtual endoscopy, closeup views are often desired. In this case a big part of the screen area is covered by only a few voxels. For this, 9 bits are often not sufficient for high quality renderings, introducing artifacts which may be distracting or misleading (see Figure 6.6). The quality of our approach, on the other hand, is independent of the voxel sizes and it is even faster for bigger voxels.
To conclude the performance analysis of our technique applied to volume rendering, we want to analyze the time spent in every phase of the rendering pipeline (see 3.3.3). Table 6.3 shows both, the absolute and the relative execution times. A few further points should be addressed in the following:

- Especially, concerning bigger data sets, most time is spent for octree traversal (60-90%). This demonstrates, that our implementations for composition and root finding are fast enough as not to limit the rendering times for this approach.

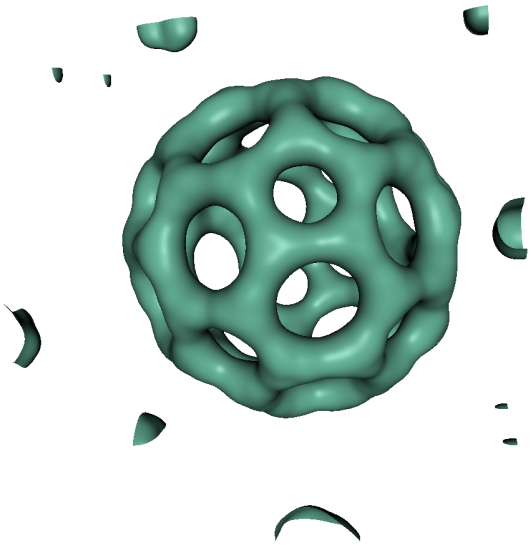
- One can also see that synthetic datasets, e.g. the distance field represented by the dragon dataset is rendered faster than "real-life" datasets, such as the foot or vertebra dataset. The traversal time, which should scale approximately logarith-



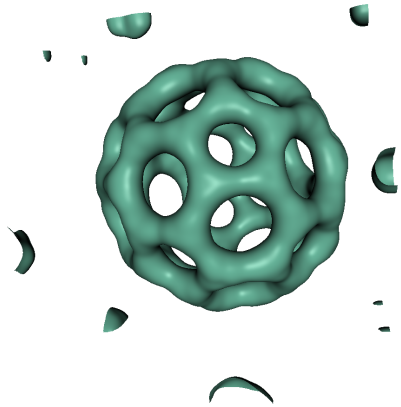
(a) tri-linear



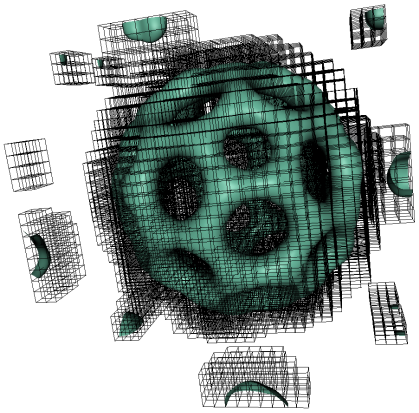
(b) tri-squared



(c) tri-cubic

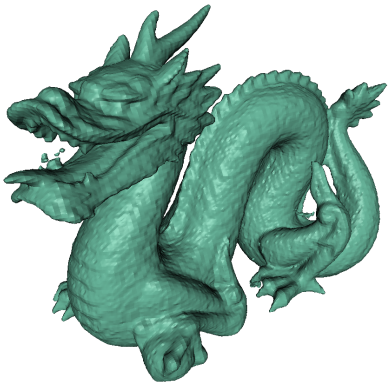


(d) sampled

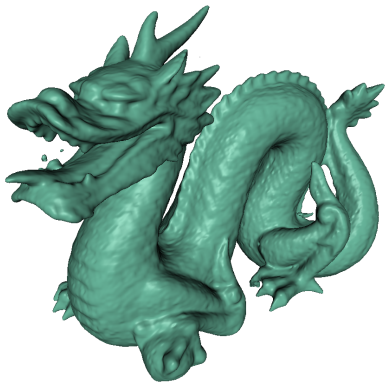


(e) bricks (1^3) displayed

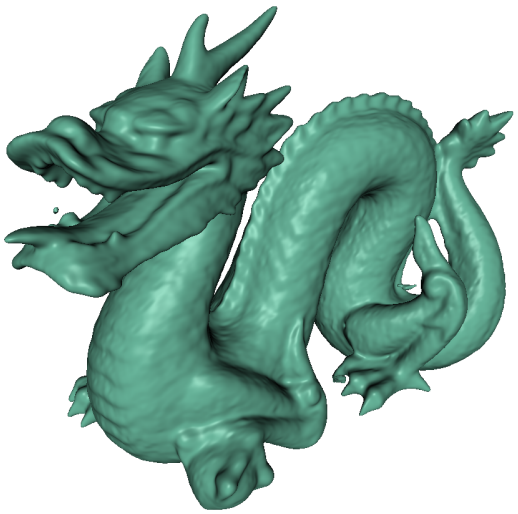
Figure 6.2: Bucky Ball 32^3



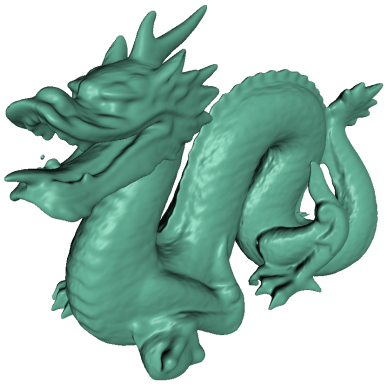
(a) tri-linear



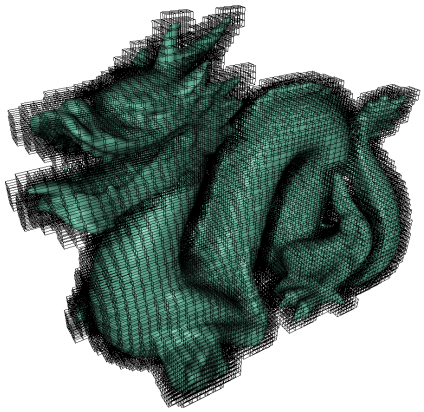
(b) tri-squared



(c) tri-cubic

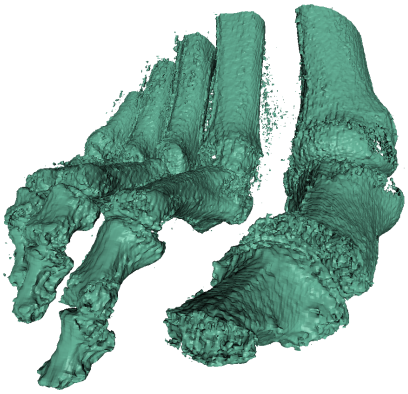


(d) fast tri-cubic sampling

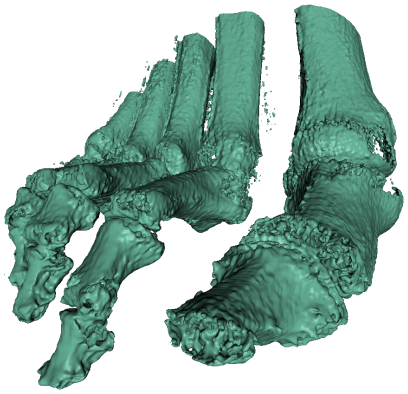


(e) bricks (2^3) displayed

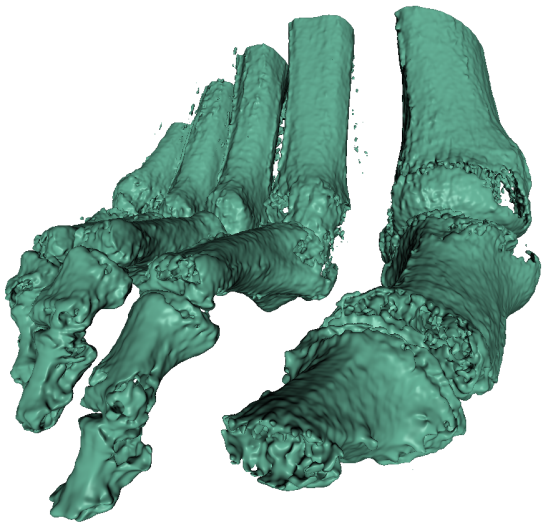
Figure 6.3: Dragon 128^3



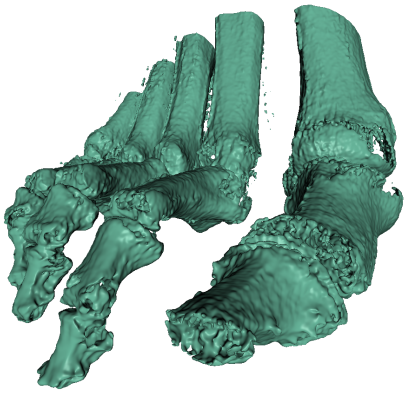
(a) tri-linear



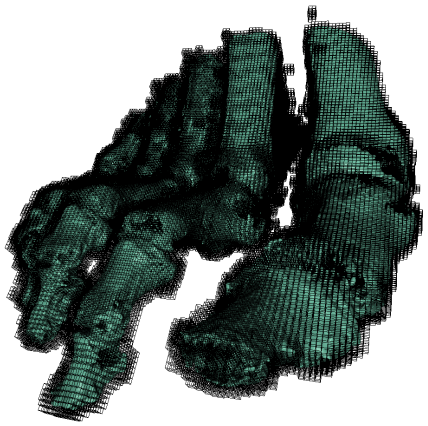
(b) tri-squared



(c) tri-cubic

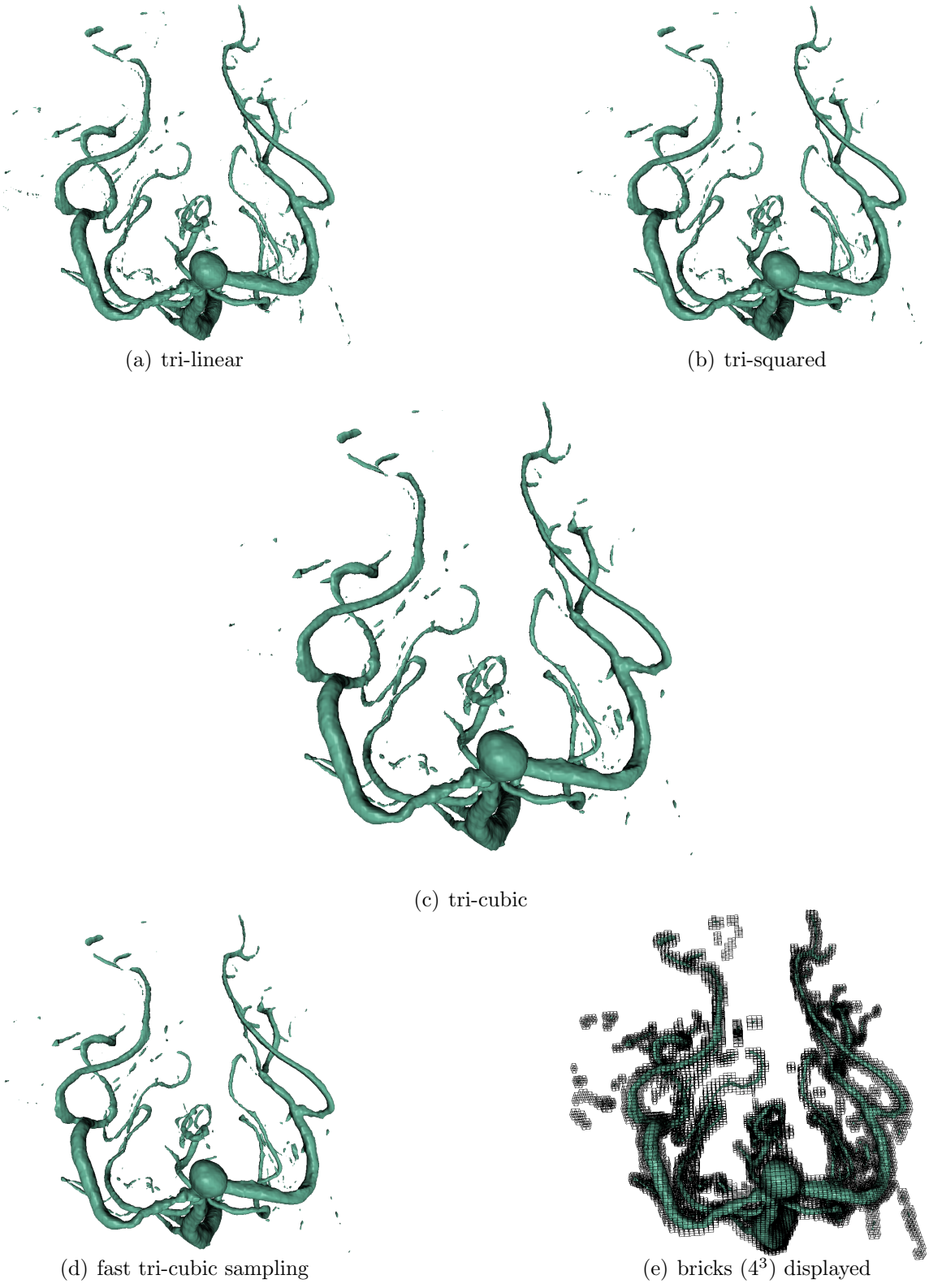


(d) fast tri-cubic sampling



(e) bricks (2^3) displayed

Figure 6.4: Foot Dataset 256^3



(a) tri-linear

(b) tri-squared

(c) tri-cubic

(d) fast tri-cubic sampling

(e) bricks (4^3) displayed

Figure 6.5: Vertebra 512^3

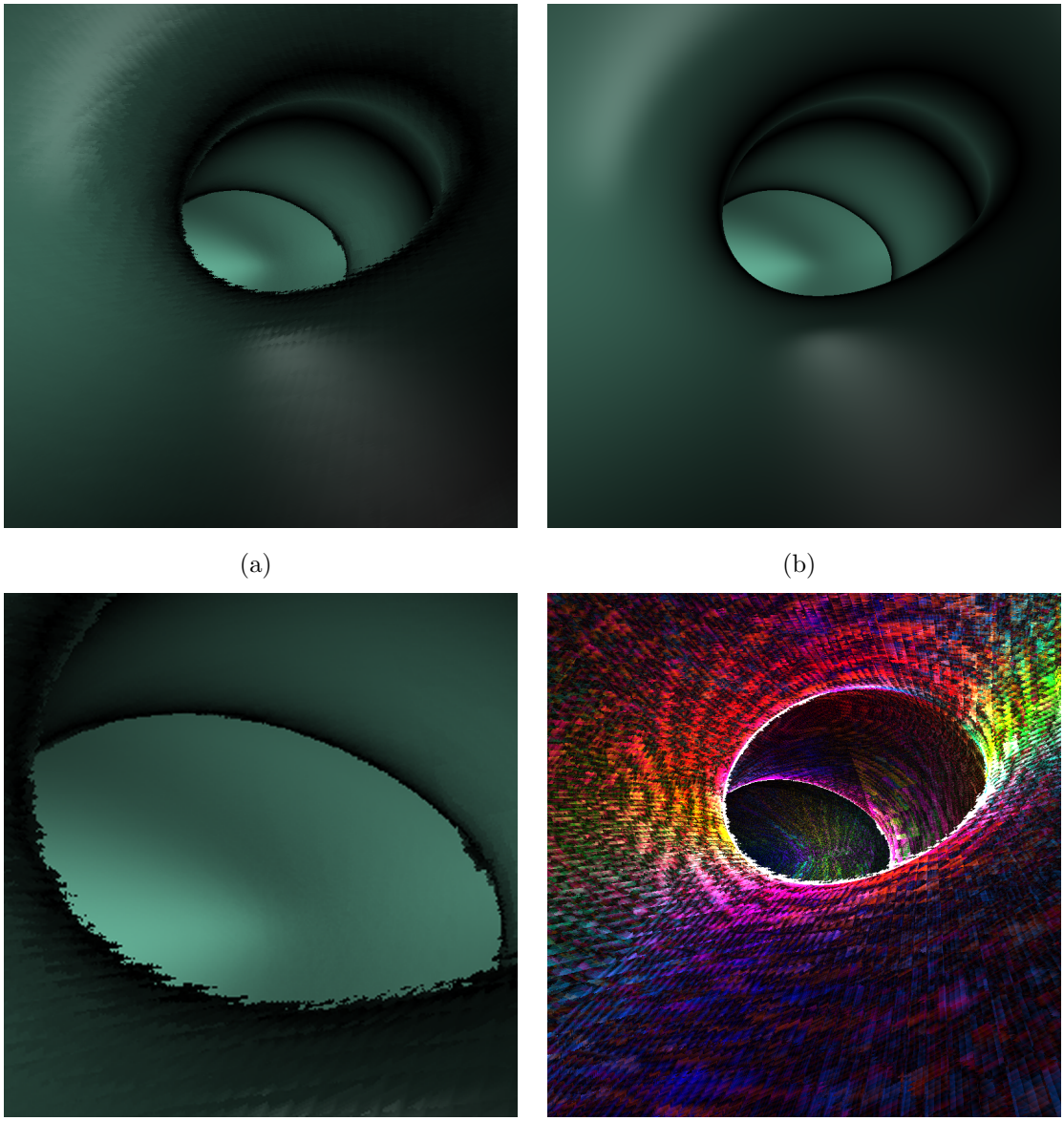
	Bucky	Dragon	Foot	Vertebra
tri-linear	31.2 fps	9.8 fps	4.6 fps	6.6 fps
tri-square	16.3 fps	6.9 fps	2.8 fps	5.3 fps
tri-cubic	7.1 fps	4.6 fps	1.5 fps	3.5 fps
sampling	29.8 fps	18.3 fps	12.0 fps	4.7 fps

Table 6.2: Performance measurement for datasets from Figure 6.2-6.5: Viewport size 1024×768*; first three methods correspond to our approach using different degree for interpolation; sampling is our implementation of [SH05] using our octree structure for emtpy space skipping. The relatively low frame rates for sampling approach occur as it has been configured to reach high visual quality – still some difference to the true solution is visible.*

mically with the size of the dataset, is double as high for the foot dataset as it is for the dragon dataset.

- The traversal time is decreased for lower order of interpolation. The reason for this might be the more exact estimation of minimum and maximum values within leaf nodes.

- While for algebraic surface rendering the ratio of the time investment of composition to root finding favors the composition part, this ratio inverses for isosurface rendering of volume datasets (see Table 6.1). In case of algebraic surface rendering, only a single polynomial is processed. In case of voxel datasets, it occurs more frequently that the polynomial along the ray is computed and the root finder can trivially reject it, as the control polygon does not show any zero crossing. Furthermore, we reduced the number of iterations for the root finder to three, five and seven for tri-linear, tri-square and tri-cubic interpolation respectively.



(a) (b)

(c) (d)

Figure 6.6: Vertebra dataset (see Figure 6.5); inside view of an artery close to an aneurysm found in a human head: Fast tri-cubic sampling [SH05] (a) produces artifacts due to low number of bits used in interpolation hardware (see (c) for a closeup view), which may be both distracting and misleading for diagnosis. (d) shows the difference of normals between (a) and (b).

(a)

	Bucky			Dragon			Foot			Vertebra		
	lin	squ	cub	lin	squ	cub	lin	squ	cub	lin	squ	cub
setup	0.40	0.39	0.41	0.40	0.40	0.39	0.40	0.39	0.40	0.42	0.42	0.42
traverse	21.96	32.95	40.47	54.06	92.94	114.9	100.2	205.0	224.0	98.64	173.6	214.1
compostion + load	5.46	21.92	75.55	2.51	11.88	41.44	3.85	22.22	67.16	4.33	20.11	64.23
rootfind	0.62	6.30	22.29	0.67	4.52	21.58	1.26	6.73	23.48	0.77	4.53	17.24
normal calc + store	0.21	2.32	10.56	0.41	2.90	17.78	0.71	4.34	22.58	0.37	2.50	15.74

(b)

	Bucky			Dragon			Foot			Vertebra		
	lin	squ	cub	lin	squ	cub	lin	squ	cub	lin	squ	cub
setup	1.40	0.62	0.28	0.69	0.35	0.20	0.38	0.17	0.12	0.40	0.21	0.13
traverse	76.65	51.58	27.11	93.11	82.51	58.61	94.16	85.89	66.35	94.37	86.31	68.69
compostion + load	19.07	34.31	50.61	4.33	10.55	21.13	3.62	9.31	19.89	4.14	9.99	20.60
rootfind	2.15	9.86	14.93	1.16	4.02	11.00	1.18	2.82	6.95	0.74	2.25	5.53
normal calc + store	0.74	3.63	7.07	0.71	2.57	9.06	0.67	1.82	6.69	0.35	1.24	5.05

Table 6.3: Breakdown of time (in ms: (a); in % (b)) needed in the different rendering stages for different data sizes and degrees.

6.3 Multiresolution Performance

Our method used for multiresolution wavelet selection can even run on a per frame basis with an overhead of less than 10% whilst increasing framerates up to 100% and more. See Table 6.4 and Figure 6.7 for our method applied to the Dragon dataset. Due to the exact representation in terms of piecewise polynomials and the uniform layout of the pipeline, even closeups of regions, which are covered by different resolutions, do not show any artifacts or holes in the surface. Figure 6.8 details on a multiresolution border, to demonstrate this smooth transition between different resolution levels. Although, our method has been designed with the idea of mixing resolutions within a dataset, it can also be directly applied for datasets being displayed at greater distances. In this case the resolution of the whole dataset is decreased automatically and composed by coefficients of one or two resolution levels.

	full	6.7(b)	6.7(c)	6.7(d)	6.7(e)	6.7(f)
reconst.	7.23	8.72	9.25	6.11	6.92	3.58
rendering	225	219	161	115	89.3	61.4
MSE	0	24	45	57	176	370
chunks	4681	3421	1877	585	295	73
chunks (%)	100%	73%	40%	12%	6%	1.6%

Table 6.4: Performance measurement for view dependent wavelet selection of datasets from Figure 6.7. Reconstruction and rendering times are given in ms, MSE is scaled by 10^3 *and calculated over all colored pixels. Fully reconstructed Dragon dataset* 128^3 *consists of a total of* 4681 *chunks.*

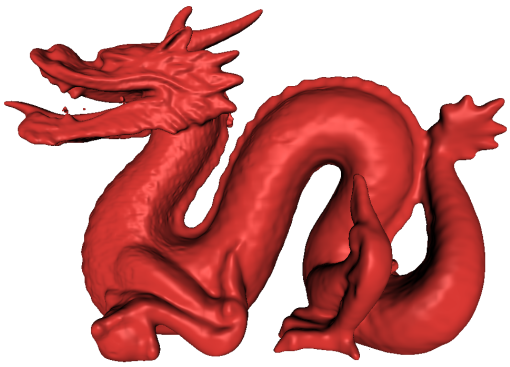
One problem of this approach is rooted in the idea of decreasing the resolution of the volume dataset and not the isosurface directly. As the data is rendered using a direct isosurface renderer, this is the only possible target for decrease in resolution. Unfortunately, which voxels are replaced by a lower resolution representation and the way the isosurface extends within in these nodes is not linked. The expressibility of a single voxel is limited due to the limited expressibility of the underlying B-spline function, which is also expressed in its polynomial representation. Accordingly, a limited number of voxels can neither represent an arbitrary complex function nor surface. This behavior is, on the one hand, desired for smoothing parts of the surface, when decreasing the resolution level. On the other hand, this also introduces problems, if the resolution is decreased too aggressively. As soon as the topology of the underlying isosurface exceeds the expressibility of the chosen subset of voxels, topological changes can be introduced (see Figure 6.9).

(a) 100% (b) 73.1%

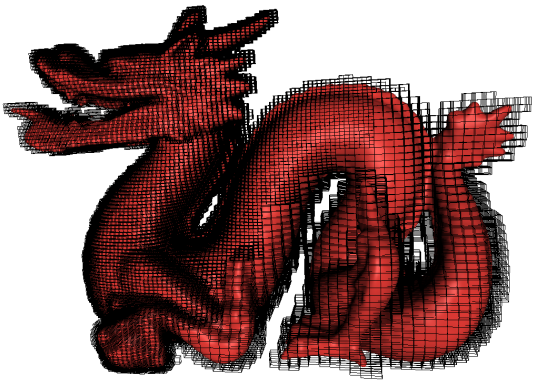
(c) 40.0% (d) 12.5%

(e) 6.3% (f) 1.6%

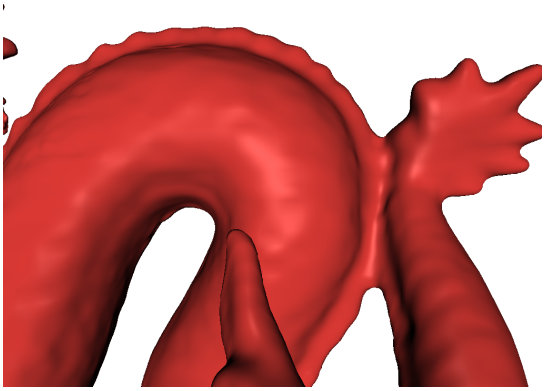
Figure 6.7: Dragon dataset: View dependent Wavelet Reconstruction greatly reduces rendering time (see Table 6.4), while retaining good visual quality.



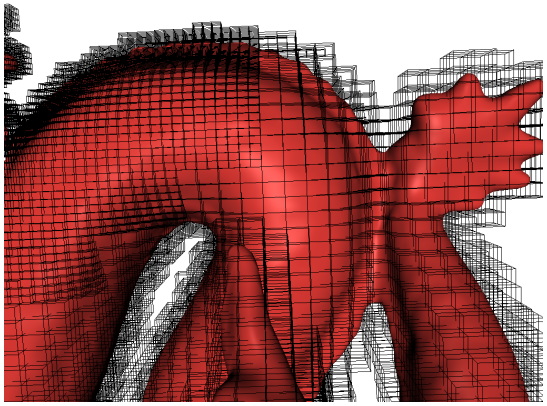
(a) different view on 6.7(b)



(b) with bricks displayed

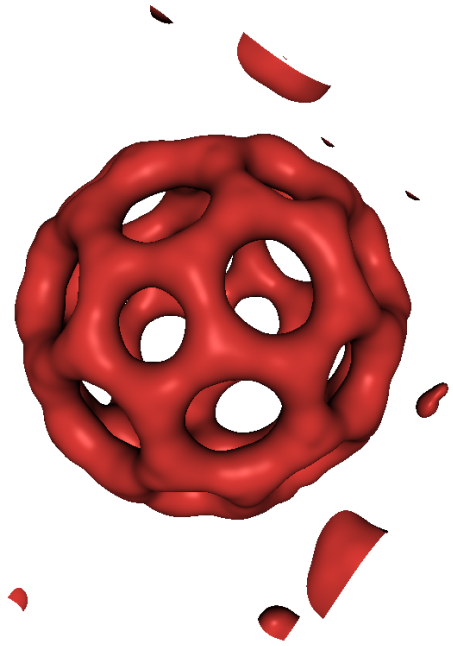


(c) closeup on the transition

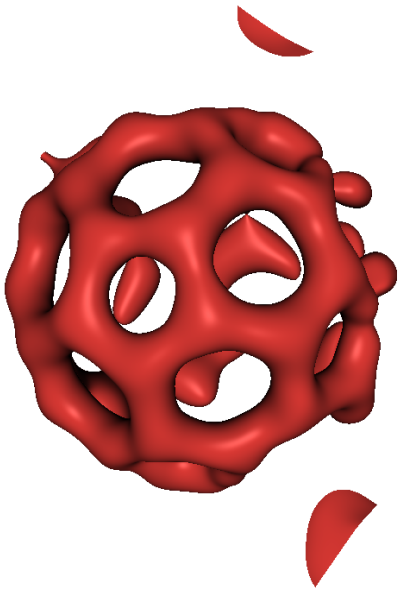


(d) with bricks displayed

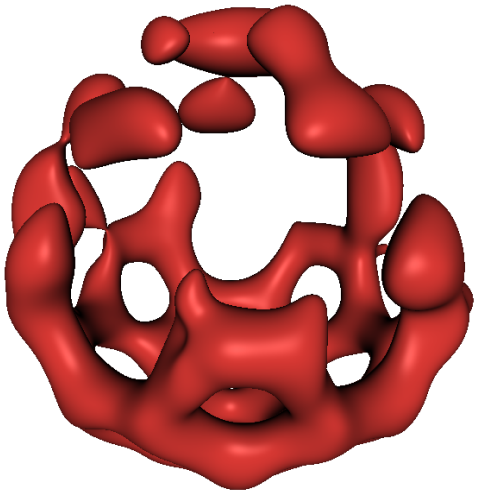
Figure 6.8: Close-up on the transition from one resolution to another of rendering 6.7(b): Our methode implicitly handles borders between different resolution levels, showing no artifacts or holes in the surface.



(a) full resolution rendering



(b) aggressive(6%) multiresolution rendering



(c) back view of the frozen setup

Figure 6.9: Bucky 32^3: an aggressive multi resolution setup introduces topological changes in the surface, due to the limited expressibility of the lower resolution data. Note: on the other hand, the same principle is desired for smoothing the surface.

Chapter 7

Conclusions and Future Work

We demonstrated that rendering of higher order algebraic surfaces is applicable to volume datasets of bigger extend, if implemented to utilize the full power of current graphics hardware. The actual ray/surface intersection can be performed in, e.g., the power form or the scaled Bernstein form, which results in simpler expressions at the same numerical stability as the frequently used (classical) Bernstein form. Moreover, all static flow control constructs are resolved before the source code is compiled to aid the optimizer in producing efficient machine code. Our rendering algorithm does not need a special data representation or preprocessing step, since an appropriate basis transformation can be applied on-the-fly. The proposed method is more accurate (though slower) than a texture hardware based intersection algorithm [SH05], and it is more efficient than the dedicated algebraic surface renderer proposed by [RS08].

Three-dimensional spline wavelets themselves are nothing but piecewise algebraic functions, which can directly be rendered by our rendering framework. This enables us to build a multiresolution description of the volume dataset and render an arbitrary approximation of the data. A view-dependent error metric is used to decrease the resolution of distant parts, to speed up rendering while high visual quality is retained in the foreground. Even transitions between different resolution levels are rendered seamlessly without any additional effort. Any desired level of continuity can be obtained by choosing wavelets of appropriate order. Cubic wavelets (for C^2-continuity) already provide good results in terms of visual quality. An octree is well suited to keep track of the underlying data and resolution levels.

Performance measurements have shown that our optimizations have shifted the main effort from composition and rootfinding to octree traversal. As this strategy of empty space skipping applied to fast approaches like [SH05] also limits their frame rates dramatically, we would want to compare fast initial node finding for our octree, different methods for empty space skipping and apply further optimizations to improve overall rendering performance.

Data organization in an octree opens up a number of new opportunities for additional features. Since irrelevant subtrees are pruned during traversal, it is not necessary to upload the entire tree to GPU memory at once, merely those parts required for the

current frame have to be available. If transmission cannot keep up with the rate at which data is needed for visualization, e.g. due to disk I/O (out-of-core rendering) or network delay (progressive transmission), the visual quality of the rendered images gradually degrades.

Acknowledgements

I would like to thank Johan Seland (SINTEF ICT, Norway) for providing the implementation of their method [RS08] in source code form and his assistance with setting up their implementation.

I am thankful to Markus Grabner for all his ideas, guidance and help during all the work on this topic and throughout different projects, which overall resulted in this work.

Special thanks to my friends Tobi and Georg for proof reading all my English work.

And as this work also marks the end of my Master studies, I would also like to thank all the people supporting me through the last five years of studying. Especially my parents Helga and Arnold, who always offered me everything I needed and made me realize all the possibilities in life. Eva, my love, who enlightens me every day. And, last but not least, all my friends, who did both for me, make me focus on technical things, as well as dragging me away from them, if I needed it.

List of Figures

List of Tables

Bibliography

[ABJ05] John C. Anderson, Janine Bennett, and Kenneth I. Joy. Marching Diamonds for Unstructured Meshes. In *IEEE Visualization 2005*, pages 423–429, October 2005.

[BDHJ00] Martin Bertram, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. In Thomas Ertl, Bernd Hamann, and Amitabh Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 389–396. IEEE, IEEE Computer Society Press, October 2000.

[CDF92] A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Comm. Pure Appl. Math.*, XLV:485–560, 1992.

[Chu92] Charles K. Chui. *An Introduction to Wavelets*, volume 1 of *Wavelet Analysis and its Applications*. Academic Press, San Diego, CA, USA, 1992.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In Eric Haines, Morgan McGuire, Daniel G. Aliaga, Manuel M. Oliveira, and Stephen N. Spencer, editors, *Proceedings of the 2009 Symposium on Interactive 3D Graphics, SI3D 2009, February 27 - March 1, 2009, Boston, Massachusetts, USA*, pages 15–22. ACM, 2009.

[EHK+06] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. A K Peters, Ltd., 2006. ISBN 1-56881-266-3.

[FR87] Rida T. Farouki and V. T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4(3):191–216, 1987.

[GLDH97] M. H. Gross, L. Lippert, R. Dittrich, and S. Häring. Two methods for wavelet-based volume rendering. *Computers and Graphics*, 21(2):237–252, 1997.

[GLDK95] M. Gross, L. Lippert, A. Dreger, and R. Koch. A new method to approximate the volume rendering equation using wavelet bases and piecewise polynomials. *Computers & Graphics*, 19:47–62, 1995.

[GSG96] Markus H. Gross, Oliver G. Staadt, and Roger Gatti. Efficient triangular surface approximations using wavelets and quadtree data structures. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):130–143, 1996.

[Han83] Pat Hanrahan. Ray tracing algebraic surfaces. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 83–90, New York, NY, USA, 1983. ACM.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus H. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Comput. Graph. Forum*, 24(3):303–312, 2005.

[KGB+09] Bernhard Kainz, Markus Grabner, Alexander Bornik, Stefan Hauswiesner, Judith Muehl, and Dieter Schmalstieg. Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs. In *Siggraph Asia 2009 Proceedings*. ACM SIGGRAPH, 2009.

[KKG09] Thomas Kalbe, Thomas Koch, and Michael Goesele. High-quality rendering of varying isosurfaces with cubic trivariate c1-continuous splines. In *ISVC '09: Proceedings of the 5th International Symposium on Advances in Visual Computing*, pages 596–607, Berlin, Heidelberg, 2009. Springer-Verlag.

[KOR08] John Kloetzli, Marc Olano, and Penny Rheingans. Interactive volume isosurface rendering using BT volumes. In *SI3D '08: Proceedings of the 2008*

symposium on Interactive 3D graphics and games, pages 45–52, New York, NY, USA, 2008. ACM.

[KS99] Tae-Young Kim and Yeong Gil Shin. An efficient wavelet-based compression method for volume rendering. *Computer Graphics and Applications, Pacific Conference on*, 0:147, 1999.

[KWH09] Aaron M. Knoll, Ingo Wald, and Charles D. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, 3 2009.

[KZ08] Thomas Kalbe and Frank Zeilfelder. Hardware-accelerated, high-quality rendering based on trivariate splines approximating volume data. *Comput. Graph. Forum*, 27(2):331–340, 2008.

[LB06] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25(3):664–670, 2006.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[LHJ07] Lars Linsen, Bernd Hamann, and Kenneth Joy. Wavelets for adaptively refined $\sqrt[3]{2}$ subdivision meshes. *International Journal of Computers and Applications*, 29(3):223–231, 2007.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March/April 2008.

[LR81] Jeffrey M. Lane and R. F. Riesenfeld. Bounds on a polynomial. *BIT Numerical Mathematics*, 21:112–117, 1981.

[LS75] M. Lal and R. Singh, H. ard Panwar. Sturm test algorithm for digital computer. *Circuits and Systems, IEEE Transactions on*, 22(1):62–63, Jan 1975.

[MR07] Knut Mrken and Martin Reimers. An unconditionally convergent method for computing zeros of splines and polynomials. *Math. of Comp.*, 76:845–865, 2007.

[NVI08] NVIDIA. Nvidia cuda programming guide 2.0. Technical report, NVIDIA, 2008.

[RS08] Martin Reimers and Johan Seland. Ray casting algebraic surfaces using the frustum form. *Computer Graphics Forum*, 27(2):361–370, 2008.

[RUL00] J. Revelles, C. Urea, and M. Lastra. An efficient parametric algorithm for octree traversal. In *Journal of WSCG*, pages 212–219, 2000.

[SDS96] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgann Kaufmann, San Francisco, CA, 1996.

[SG10] Markus Steinberger and Markus Grabner. Wavelet-based Multiresolution Isosurface Rendering. In *Eurographics/IEEE VGTC Symposium on Volume Graphics*, Munich, Germany, 2010. Eurographics Association.

[SH05] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. In Matt Pharr, editor, *GPU Gems 2*, chapter 20, pages 313–329. Addison Wesley, 2005.

[SHN+06] H. Scharsach, M. Hadwiger, A. Neubauer, S. Wolfsberger, and K. Bühler. Perspective isosurface and direct volume rendering for virtual endoscopy applications. *Eurographics/IEEE-VGTC Symposium on Visualization*, pages 315–322, 2006.

[SR03] Javier Sánchez-Reyes. Algebraic manipulation in the Bernstein form made simple via convolutions. *Computer-Aided Design*, 35(10):959–967, 2003.

[TPG99] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics*, 23:583–598, 1999.

[UHP00] Tushar Udeshi, Randy Hudson, and Michael E. Papka. Seamless multiresolution isosurfaces using wavelets. Technical Report ANL/MCS-P801-0300, Argonne National Laboratory, 2000.

[WB97] Pak Chung Wong and R. Daniel Bergeron. Performance evaluation of multiresolution isosurface rendering. In *DAGSTUHL '97: Proceedings of the Conference on Scientific Visualization*, page 322, Washington, DC, USA, 1997. IEEE Computer Society.

[Wes94] Rüdiger Westermann. A multiresolution framework for volume rendering. In *Symposium On Volume Visualization*, pages 51–58. ACM Press, 1994.

[WKE99] R. Westermann, L. Kobbelt, and T. Ertl. Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *The Visual Computer*, 15(2):100–111, 1999.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of IEEE Visualization '96*. IEEE, IEEE Computer Society Press, October 1996.

Appendix A

Code Examples

This chapter offers different simplified code snippets from the implementation, which should assist the reader in understanding the underling algorithmic and structural ideas.

Listing A.1: C code of our Horner Scheme for multivariate to univariate composition: tri-cubic case; power form. Note: only 16 registers for input and 11 temporary registers needed; inplace convolution for b of size two does not need any additional registers.

```
1  void add(real_t *a, real_t *b, int size);
2  real_t get_f(int i, int j, int k);
3
4  void convolution_inplace(real_t *a, real_t *b, int size_a)
5  {
6    //size_b == 2
7    a[size_a] = a[size_a-1]*b[1];
8    for(int i = size_a-1; i > 0; --i)
9      a[i] = b[0]*a[i] + a[i-1]*b[1];
10   a[0] *= b[0];
11 }
12
13 void composition(real_t r_x[2], real_t r_y[2], real_t r_z[2], real_t result[10])
14 {
15   //result needs to be filled with zeros already
16   real_t *f_t = result;
17   real_t f_i[7];
18   real_t f_ij[4];
19   for(int i = 3; i >= 0; ++i)
20   {
21     //set f_i to zero
22     for(int t = 0; t < 7; ++t)
23       f_i[t] = 0;
24     //calculate next f_i
25     for(int j = 3; j >= 0; ++j)
26     {
27       //set f_ij to zero
28       for(int t = 0; t < 4; ++t)
29         f_ij[t] = 0;
30       //calculate next f_ij
31       for(int k = 3; k >= 0; ++k)
32       {
33         f_ijk = get_f(i,j,k);
34         //add f_ijk to f_ij
35         add(f_ij,f_ijk,1);
36         //convolution with r_x - implementing Horner Scheme
37         if(k != 0)
38           convolution_inplace(f_ij,r_z,4-k);
```

```
39        }
40        //add f_ij to f_i
41        add(f_i,f_ij,4);
42        //convolution with r_y - implementing Horner Scheme
43        if(j != 0)
44          convolution_inplace(f_i,r_y,7-j);
45      }
46      //add f_i to f_t
47      add(f_t,f_i,7);
48      //convolution with r_x - implementing Horner Scheme
49      if(i != 0)
50        convolution_inplace(f_t,r_x,10-i);
51    }
52  }
```

Listing A.2: Simplified C code of our LR81 implementation for the tri-cubic case. Note: Algorithm has constant memory consumption and does not need any pointer or index operation that could not be evaluated at compile time, which makes this implementation well suited for GPU execution.

```
1  bool findCrossing(real_t poly[10], int& crossing);
2  void copy(real_t *dst, real_t src*, int length);
3  void deCasteljauSplit(real_t parent[10], real_t left[10], real_t right[10], real_t at)
4  {
5    real_t u = 1.0-at;
6    left[0] = parent[0]; right[9] = parent[9];
7    for(size_t i = 1; i <= 9; ++i)
8    {
9      left[i] = u*parent[0] + t*parent[1];
10     right[9-i] = u*parent[9-i] + t*parent[10-i];
11     for(size_t j = 1; j < 9-i; ++j)
12       parent[j] = u*parent[j] + t*parent[j+1];
13     parent[0] = left[i];
14     parent[9-i] = right[9-i];
15   }
16 }
17 real_t approxRoot(real_t poly[10], int crossing)
18 {
19   //linearly interpolate between points
20   real_t k = poly[crossing] - poly[crossing+1];
21   return = poly[crossing] / (10.0*k) + crossing/10.0;
22 }
23 int lr81(real_t poly[10], real_t& rootpos)
24 {
25   real_t *left = poly;
26   real_t right[10], temp[10];
27   int crossing, firstsplit;
28   real_t at, ActRoot(1), CurRoot(1), OldCRoot(1), Scale(1);
29
30   //no root at all
31   if(!findCrossing(poly, crossing))
32     return 10;
33
34   real_t at = approxRoot(poly, crossing);
35   firstsplit = crossing;
```

```
36    /* ... snipped update of Scale and Root
37    */
38    copy(temp, poly);
39    for(int iteration = 0; iteration < MAX_ITERATION; ++iteration)
40    {
41      deCasteljauSplit(temp, left, right, at);
42      if(findCrossing(left, crossing))
43      {
44        at = approxRoot(left, crossing);
45        /* ... snipped update of Scale and Root */
46        copy(temp, left);
47      }
48      else if(findCrossing(right, crossing))
49      {
50        at = approxRoot(right, crossing);
51        /* ... snipped update of Scale and Root */
52        copy(temp, right);
53      }
54      else
55      {
56        //no root
57        return 10;
58      }
59    }
60    rootpos = ActRoot;
61  }
```

Listing A.3: Simplified C code of the first and second phase of our aRFBS implementation for the tri-cubic case.

```
1   int findFirstCrossings(real_t poly[10], real_t& crossing1, real_t& crossing2);
2
3   bool aRFBS(real_t poly[10], real_t& rootpos)
4   {
5     real_t *left = poly;
6     real_t temp[10], right[10];
7     int crossings;
8     real_t crossing1, crossing2;
9     real_t ActSplit(1), CurSplit(1), OldCSplit(1), Scale(1);
10    crossings = findFirstCrossing(left, crossing1, crossing2);
11    if(crossings == 0)
12      return false;
13
14    //begin phase one
15    if(crossings > 1)
16    {
17      crossing1 = 0.5*(crossing1 + crossing2);
18      /* ... snipped update of Scale and Root */
19      copy(temp, left, 10);
20      while(crossings > 1)
21      {
22        deCasteljauSplit(temp, left, right, crossing1);
23        crossings = findFirstCrossing(left, crossing1, crossing2);
24        if(crossings == 0)
25        {
26          crossings = findFirstCrossing(right, crossing1, crossing2);
```

```
27          if(crossings > 1)
28          {
29            copy(temp, right, 10);
30            crossing1 = 0.5*(crossing1 + crossing2);
31            /* ... snipped update of Scale and Root */
32          }
33          else if(crossing == 0)
34            return false;
35          else
36            copy(left, right, 10);
37        }
38        else if(crossing > 1)
39        {
40          copy(temp, left, 10);
41          crossing1 = 0.5*(crossing1 + crossing2);
42          /* ... snipped update of Scale and Root */
43        }
44      }
45    }
46    phase2(left, rootpos);
47    return true;
48  }
49
50  void bernstein2ScaledBernstein(real_t poly[10]);
51  real_t evalScaledBernsteinHorner(real_t poly[10], real_t where);
52  void phase2(real_t poly[10], real_t& rootpos)
53  {
54    //begin phase two
55    bernstein2ScaledBernstein(poly);
56    real_t a = 0;
57    real_t b = 1;
58    real_t fa = poly[0];
59    real_t fb = poly[9];
60    //init c with first zero crossing
61    real_t c = crossing1;
62    real_t fc, u;
63
64    fc = evalScaledBernsteinHorner(left, c);
65    if(abs(fc) < RES)
66    {
67      rootpos = c;
68      /* ... snipped update of Scale and Root */
69     return true;
70    }
71
72    if(fc*fa > 0.0)
73      a = c, fa = fc;
74    else
75      b = c,  fb = fc;
76
77    for(size_t iteration = 1; iteration < iterations; ++iterations)
78    {
79      if(iteration % 2)
80        //regula falsi step
81        c = (a*fb-b*fa)/(fb-fa);
82      else
```

```
83        //binary step
84        c = 0.5*(a+b);
85      fc = evalScaledBernsteinHorner(left, c);
86
87      if(fc*fa > 0.0)
88        a = c, fa = fc;
89      else
90        b = c,  fb = fc;
91    }
92  }
```

Listing A.4: Simplified C code for our Sturm Series based Rootfinder for the tri-cubic case.

```
1   void calcStrumSeries(real_t** sturmChain);
2   unsigned sturmComputeSignChanges(real_t** sturmChain, real_t pos)
3   {
4     unsigned changes = 0;
5
6     //init values
7     real_t value[10];
8     for(int i = 0; i < 10; ++i) value[i] = 0;
9
10    //evaluate sturm chain for pos using horner scheme
11    for(int i = 0; i < 10; ++i)
12      for(int j = 9-i; j >= 0; --j)
13      {
14        if(j != 0)
15          value[i] = (value[i] + sturmChain[i][j])*pos;
16        else
17          value[i] += sturmChain[i][j];
18      }
19
20    //check sign changes
21    for(int i = 1; i < 10; ++i)
22      changes=changes+(value[i-1]*value[i] < 0);
23    return changes;
24  }
25
26  bool sturmbinary(real_t poly[10], real_t& rootpos)
27  {
28    //parts of the sturm series
29    real_t st1[9], st2[8], st3[7], st4[6], st5[5], st6[4], st7[3], st8[2], st9[1];
30    real_t *sturmChain[10] = { poly, st1, st2, st3, st4, st5, st6, st7, st8, st9 };
31    calcStrumSeries(sturmChain);
32
33    //signchanges
34    unsigned sc_a,sc_b;
35    real_t pos_a(0), pos_b(1);
36
37    sc_a = sturmComputeSignChanges(sturmChain,pos_a);
38    sc_b = sturmComputeSignChanges(sturmChain,pos_b);
39    if (sca == scb)//equal number of sign changes -> no root
40        return false;
41
42    for(unsigned it = 0; it < ITERATIONS; ++it)
```

```
43    {
44      real_t pos_m = 0.5*(pos_a+pos_b);
45      unsigned sc_m = sturmComputeSignChanges(sturmChain,pos_m);
46      if (sc_m == sc_a) //no zero in first half
47        //second half will be subdivided further
48        pos_a = pos_m, sc_a = sc_m;
49      else  //zeros in first half
50        //first half will be subdivided further
51        pos_b = pos_m, sc_b = sc_m;
52    }
53    return true;
54  }
```