Multiresolution Isosurface Rendering

Markus Steinberger*

Institute for Computer Graphics and Vision
Graz University of Technology
Graz / Austria

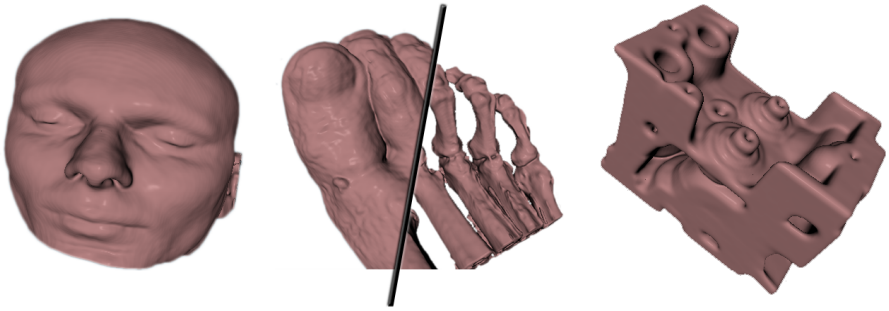


Figure 1: Perspective real time renderings of three different data sets. Left: MRI of a human head (1.5 fps; Center: composition of two different isosurface levels of a human foot (1.6 fps); Right: CT scan of two cylinders of an engine block (2.0 fps). All datasets are of size 128^3*. The system is capable of changing the isosurface level in real time, keeping all the data on the GPU.*

Abstract

In this paper we propose a new technique for isosurface rendering of volume data. Medical data visualization, e.g. relies on exact and interactive isosurface renderings. We show how to construct a multi resolution view of the data using bi-orthogonal spline wavelets and how to perform fast rendering using raycasting implemented on the GPU. This approach benefits from the properties of both: the wavelet transform and the reconstruction using three dimensional splines. The smoothness of surfaces is zoom level independent and data can be compressed to speed up rendering while still being able to show full detail quickly. Ray evaluation is implemented in model space to enable perspective rendering. Due to the fact that the isosurface is not extracted from the data beforehand, the isosurface level as well as the current resolution level can be changed without any further computation.

Keywords: Isosurface Rendering, Volume Data, Wavelet, 3D DWT, Biorthogonal Spline Wavelets

1 Introduction

The rendering of isosurfaces is a well known problem in the field of computer graphics. It deals with the task of finding a closed surface corresponding to a particular density value within a three dimensional density field, similar to the two dimensional problem of drawing contour lines on a topographical map. As there are many measurement devices producing three dimensional density fields, the use of isosurface rendering is widely spread, especially visualization of medical scan data, such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) are of big interest.

The main problem engineers are facing is the huge amount of data[1]. One of the first approaches which had been made, was extracting the isosurface from the data using the well known Marching Cubes Algorithm [7] and rendering a simple triangle mesh. Even though the output of the Marching Cubes Algorithm gives a rough and quite noisy mesh, the algorithm itself is so fast and efficient that it is still used.

1.1 Related Work

Of course a lot of authors tried to improve and alter the algorithm, e.g. in [13, 2] the reader finds approaches using different primitive shapes, improvement can also be made by, e.g. usage of a different triangulation [4]. When examining all these different algorithms from a signal processing point of view, it can be shown that all of them can be generalised to a simple reconstruction of discrete sampled data using different reconstruction filters (for evaluation of these filters see [9]).

Only the simplest meshes produced by these filters can be built using simple polygons, so a reasonable step for each pixel we want to evaluate, is to use the idea of ray tracing for calculating the intersection of the isosurface and a viewing ray. A combination of ray tracing and filter

*markus.steinberger@student.tugraz.at

[1] Just consider a small dataset of $512 \times 512 \times 512$ voxels, where each voxel is represented by a floating point number of 32 bits: 4 byte $\cdot 512^3 =$ 512 MB

evaluation has been implemented on various hardware architectures (e.g. [14, 8]). A recent work [5] focusing on interactive isosurface rendering does not extract a certain surface from the data beforehand. The advantage of this approach is clearly the ability to change the user defined isosurface level online – facing the problem of higher memory consumption.

Another method we want to mention here is an approach trying to truncate unneeded data using discrete wavelet transformation of the given data and thus reducing the dataset by means of discarding both high frequency information as well as irrelevantly sized basis functions [15]. Unfortunately, this approach also uses Marching Cubes to get displayable meshes.

Changing the point of view from sampled data representation to an algebraic form, one should consider the approaches given in [10]. This paper deals with the problem of rendering three variate polynomials using the so called Frustum Form. Using this approach, higher frame rates are possible as the special shape of the viewing frustum can be exploited to precompute data for all casted rays. We are looking forward to testing some of these methods for fast and stable root finding.

1.2 Contribution

We will combine a lot of the aforementioned ideas and taking them a step further to show - as far as we know - a completely different approach of rendering isosurfaces. Our main goal is to provide a method of interactive isosurface rendering, that is capable of both: displaying the given data quickly when further away from the actual point of interest as well as displaying all details, when taking a close-up.
Our approach will combine ideas from the discrete and the continuous wavelet transformation, three dimensional splines and numeric root solving. After some pre-processing, we will show how to construct a render tree structure and placing it on the graphics card for doing highly parallel raytracing. Therefore, we use state of the art technology as CUDA for computing the intersection of each viewing ray with the isosurface.

1.3 Outline

In the first part of this paper, we will briefly review the basic ideas of wavelets, focusing on bi-orthogonal spline wavelets and how these ideas can be exploited for isosurface rendering. We will demonstrate how to reduce the amount of memory needed, whilst still being able to switch quickly to full detail rendering. Chapter 3 will focus on traversing the render tree and evaluating the isosurface polynomial along a viewing ray, while chapter 4 will show our final results as well as focusing on stability problems and on how to prevent image artefacts. Finally, the paper will conclude with discussing open questions and provide ideas for future work that involve multiresolution wavelets.

2 Multiresolution Volume Data Representation

A common approach for building a multiresolution view on data is the representation of the data using differently sized basis functions. We will show in Chapter 3, that compactly supported bi-orthogonal spline wavelets [1] are well suited for our system. An introduction to wavelets can be found in [3, 12]. In simple words, one could describe the wavelet transformation as the approach of constructing an arbitrary signal using scaled and dilated instances of just two functions: the wavelet function $\psi(x)$ and the scaling function $\varphi(x)$. When the wavelet transformation is carried out in a discrete sampled domain, we talk about the discrete wavelet transformation (DWT). When we consider a continuous domain on the other hand, we talk about the continuous wavelet transform (CWT). When applying an appropriate DWT, we get an alternative representation of the data, which needs as much storage as the original data. Furthermore, it supports a perfect reconstruction of the data, where every value of the DWT data corresponds to either an instance of scaling function, or the wavelet function (see figure 2).

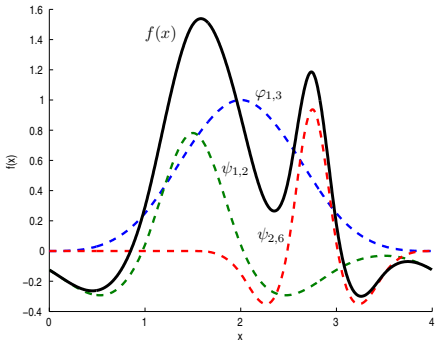


Figure 2: $f(x)$ constructed from scaled and dilated versions of scaling function $\varphi(x)$ and wavelet function $\psi(x)$: $f(x) = \varphi_{1,3} + \psi_{1,2} + \psi_{2,6}$

Using all this information, it is clear that we are able to construct a DWT of the given input data, followed by discarding all data samples which do not contribute enough information for our final render and finally getting the data back to a displayable format.

2.1 3D Discrete Wavelet Transform

Implementing the DWT can be done in a recursive manner. Starting with the given signal, the signal is filtered using high pass filter (with impulse response h) as well as low pass filter (with impulse response g) followed by down sampling of the factor two, which provides the detail coefficients and the approximation coefficients respectively. After this step, we can simply store the detail coefficients

and run the approximation coefficients through this whole mechanism again. Assuming the original data has been of size $N = 2^k$ k, $\in \mathbb{N}$, this procedure can be repeated until we are left with only one approximation coefficient (see figure 3). The data could be perfectly reconstructed using reconstruction high and low pass filters ($\hat{h}$ and $\hat{g}$) and up sampling.

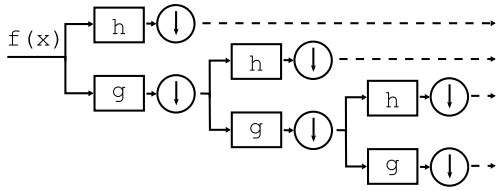


Figure 3: Block diagram of a 1D DWT

The three dimensional DWT (3D DWT) is constrcuted using the same structure we used for the one dimensional case. The three dimensional signal needs to be filtered along all three coordinate axis. After filtering along the first coordinate axis (e.g. x-axis), two outputs are obtained which are then filtered along the second coordinate axis (e.g. y-axis). This gives us a total of four outputs, which are now filtered along the last coordinate axis (e.g. z-axis), finally computing eight different signals. Now all seven signals which were high pass filtered along at least one coordinate axis are stored, while the one remaining signal which has been filtered using only low pass filters is being analyzed again. (See figure 4).

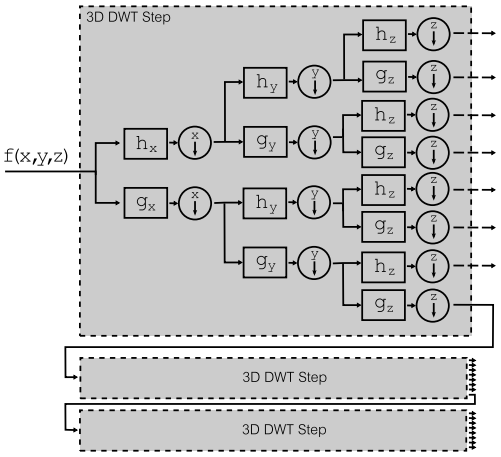


Figure 4: Block diagram of a 3D DWT

An efficient implementation of the 3D DWT takes into account that only a few filter coefficients are non zero for every filter and only every second value of the convolution has to be computed due to the down sampling. It should be considered that the structure of the 3D DWT leaves room for a quite high degree of parallelism.

2.2 Bi-Orthogonal Spline Wavelets

As already mentioned before, we used bi-orthogonal spline wavelets as described in [1]. If our sole aim had been to reconstruct the data perfectly after the DWT, there would have been better choices for wavelets with better time frequency characteristics. We have chosen this kind of wavelets, as we are reconstructing the data using splines. Thus, we can simply go to continuous space directly, instead of performing an inverse discrete wavelet transformation.

First of all, we want to take a closer look at this kind of wavelets. In our approach, we used the same filters for all three dimension of the 3D DWT. One example of the used filters and the corresponding wavelet and scaling function $\psi(x)$ and $\varphi(x)$ are visualized in figure 5.

g	-0.25	0.75	0.75	-0.25
h	0.125	-0.375	0.375	-0.125
$\hat{g}$	0.25	0.75	0.75	0.25
$\hat{h}$	0.5	1.5	-1.5	-0.5

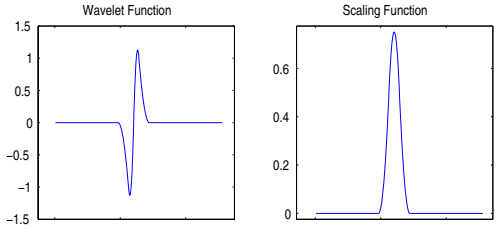


Figure 5: Analysis and Reconstruction Filters, Wavelet and Scaling Function for Spline Wavelet with $N_1 = 3$ and $N_2 = 1$ Vanishing Moments.

Examining the wavelet function $\psi(x)$ and the way it is constructed during an inverse wavelet transformation, one can easily conclude that this function can be constructed using instances of the scaling function $\varphi(x)$ which are half the size of the wavelet function:

$$\psi(x) = \sum_i \hat{h}_k \cdot \varphi(2 \cdot x - d_k)$$

where $\hat{g}_i$ corresponds to the filter coefficients of the reconstruction filter $\hat{h}$. d_k depends on the time delay of the chosen filters as well as the vanishing moments of the filter, but we can state that $d_{k+1} - d_k = \text{const} \; \forall \; k$ (see figure 6).

This relationship is very important for the later reconstruction and rendering process, as we just have to deal with scaled and dilated versions of the scaling function. In fact, the knots of these half sized scaling functions coincide with the knots of the scaling functions used to construct the next higher resolution's low pass data. From now on we will only consider coefficients that correspond to scaling functions and name them wavelet coefficients for simplicity.

2.3 Inverse Continuous Wavelet Transform of DWT Data

As mentioned before, we are going to mix continuous and discrete wavelet transformation. For the reconstruction of the DWT data we are using the inverse continuous wavelet

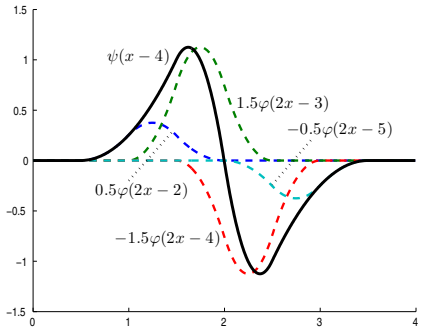


Figure 6: Construction of the wavelet function $\psi(x)$ using dilated versions of half sized scaling function $\varphi(x)$

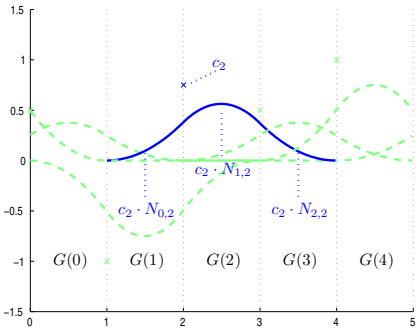


Figure 7: Scaling Function $\varphi(x)$ for $N_1 = 3$ and $N_2 = 1$ Vanishing Moments placed in 1D Space given the coefficients c_i. The Scaling Function is given by the B-spline curve of degree two, which is constructed using the polynomials $N_{0,2}, N_{1,2}, N_{2,2}$.

transform. By doing so, it can not be guaranteed that the reconstructed data will interpolate the original sampled data anymore. Although this will probably work out. A different point of view will provide more insight into this idea:

If we are not taking the sampled input data as our starting point, but use the scaling function as our spline function to pre-compute the spline coefficients of a three dimensional interpolating spline, it can be guaranteed that the fully reconstructed signal will interpolate the data perfectly[2].

2.3.1 Localization and Support of Scaling Functions

As stated before, we are going to reconstruct the data only using information of the scaling function. We already mentioned how the wavelet function can be created using the scaling function, which is in fact just the B-spline curve of degree $k = N_1 - 1$. Ignoring the boundary conditions, we can immediately derive its support. For simplicity, we derive it by means of overlap between adjacent wavelet coefficients: Given a scaled and dilated scaling function which equals a B-spline curve of degree k, this scaling function will interfere with k adjacent wavelet coefficients in all dimensions.
If we think of three dimensional data and a three dimensional grid $G(x_1,x_2,x_3)$ of size $n \times n \times n$ for $0 \leq x_i < n$ and n^3 wavelet coefficients, we can assign one wavelet coefficient to each voxel. Now placing in three dimensional space one scaling function for one wavelet coefficient c_{r_1,r_2,r_3}, it will influence the voxel $G(r_1,r_2,r_3)$ as well as all voxels $G(u_1,u_2,u_3)$ with

$$\left\lfloor r_i - \frac{k}{2} \right\rfloor \leq u_i \leq \left\lfloor r_i + \frac{k}{2} \right\rfloor$$

E.g. the Haar wavelet which corresponds to $k = 0$, will not influence its adjacent voxels; a B-spline wavelet of degree 2 will influence a box of $3 \times 3 \times 3$ voxels centered at its location (see figure 7).
If the wavelet coefficient is placed at the border of the grid, the reconstruction of the data depends on the chosen border conditions. There are several options:

- *use a cyclic repetition:* every part of the scaling function which exceeds the border shows up at the opposite side of bounded space. This might lead to artefacts, if the density close to the border is not close to zero.
- *zero extension:* filling up the surrounding space with zeros and extending the border of the volume, removes not only the artefacts of the cyclic repetition, but also falsifies the representation if only low frequencies are considered, as the data is 'smeared out'. Another problem of this approach is that as many zeros as there are data samples have to be inserted for each dimension, if we want to get rid of the border problem for all resolutions. This results in eight times the original data.
- *mirror at the border:* every part of the scaling function which exceeds the border is reflected back in. This is a very accurate way of constructing the data, however it is also very complicated.
- *end point interpolating splines:* this approach uses different basis functions to interpolate the coefficients placed at the border of the grid [12].

For efficient rendering, we have to consider how different resolution – especially how single scaling functions of different size coming from different resolutions – of the same data, are placed in space, and how the corresponding voxels within which they can be represented as a simple polynomial form up.
The answer to this question can be obtained by considering the refinement equation:

$$\varphi(x) = \sum_{i \in \mathbb{Z}} g_i \varphi(2x - i) \qquad (1)$$

[2] One might choose to model the inverse system of the scanner used for creating the dataset instead of interpolating the original data. E.g. to to choose spline coefficients such that the integral of the spline over each voxel equals the corresponding sample value.

Each scaling function of a lower resolution is refined by half sized scaling functions in the higher resolution. Therefore a n-dimensional grid element of a low resolution is refined by 2^n grid elements. If we consider the three dimensional case, each voxel is refined by eight half sized voxels. If we do not only consider two resolutions, but o, we end up with an octree structure of depth o.

Using all the aforementioned information, it is possible to reconstruct any given three dimensional discrete sampled data at any resolution, using three dimensional B-spline functions, which each correspond to piecewise tensor products of polynomials of degree k.

3 Multiresolution Rendering

In chapter 2 we showed a way of how to represent any discrete sampled data, using piecewise tensor products of polynomial functions. Our next step is showing a way of rendering an arbitrary isosurface of such kind of data.

3.1 Rendertree Setup

It is possible to arrange different resolution levels of the data using an octree structure. For rendering it would be sufficient to store the spline information in the leaves of the octree only. By additionaly adding the information about every resolution level to each node, it is possible to stop the traversal earlier to render a low resolution view of the data. Using this structure, one can place just as many levels of the octree on the graphics card, which hold enough information to render the current view on this object. So if the object is far away from the camera, it should be sufficient to use less information about the volume data for rendering.
The results for this method might not always be optimal, as high amplitude and high frequency information is not considered for rendering. Another approach which will lead to better results, might just truncate the data by leaving out small amplitude coefficients. If parts of the data are truncated which sum up to a bigger space, a speedup can be achieved. In this case an adaptive pruning of the octree will also result in less memory consumption, less render effort and better frame rates.

3.1.1 Structure and Memory Consumption

To evaluate their performances, different implementations of the render tree structure have been tested. To achieve best frame rates, one should not underestimate the importance of pre-calculation of all the coefficients of the polynomial for each voxel. The isosurface of this polynomial shall be given by:

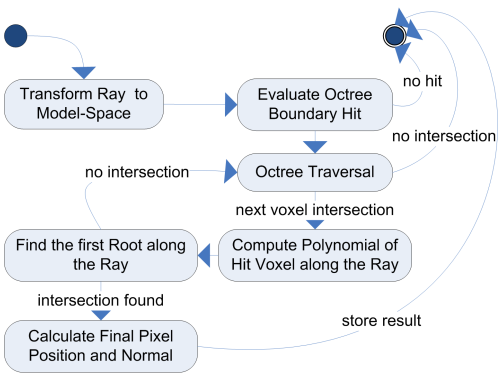
$$f(x,y,z) = \sum_{i=0}^{n_{max}} \sum_{j=0}^{n_{max}} \sum_{k=0}^{n_{max}} c_{ijk} \cdot x^i y^j z^k = l \qquad (2)$$

where n_{max} corresponds to the function's maximal exponent along one axis. Multivariate polynomials of this type are called tri-linear, tri-square and tri-cubic functions for $n_{max} = 1$, $n_{max} = 2$ and $n_{max} = 3$ respectively. Every voxel's polynomial equation is given by $(n_{max}+1)^3$ constants c_{ijk}. Due to the structure of the 3D DWT, the wavelets as well as the scaling function correspond to tensor products of the one dimensional wavelet and scaling function. So the polynomials representing single wavelet or scaling functions are separable. It seems to be sufficient to build a system adapted to rendering separable polynomials. In doing so, one tends to ignore the fact, that the sum of separable functions – that occurs after combining low resolution and high resolution data – is in general not separable anymore.
To get an idea of the memory consumption when placing the whole structure on the graphics card, a simple example for $N_1 = 2$ vanishing moments shall be given: This kind of wavelets are constructed using tri-square functions, so each voxel's polynomial is described using $(2+1)^3 = 27$ coefficients. A dataset of $128 \times 128 \times 128$ results in an octree of depth seven containing $\sum_{i=1}^{7}(2^i)^3 = 2396744$ voxels. Using floating point data types (32 bit), the overall memory consumption sums up to $2396744 * 27 * 4B \approx 247MB$. A dataset of 256^3 samples already needs $\approx 2.2GB$, which exceeds the capablities of a consumer graphics card. Solutions for this problems might be found by

- storing the coefficients which correspond to the scaling functions only and calculate the polynomials online, when needed
- keeping parts of the data on the graphics card at a time
- using an adaptive render tree structure for pruned trees, which does not allocate memory for inactivated voxels

3.2 Isosurface Rendering Pipeline

For rendering an isosurface setup using the given structure, an algorithm has to locate the first intersection of a viewing ray originating at a given pixel position. An implementation could consist of the following steps:



For a good visual impression of the isosurface, a perspective projection should be used. As the main complexity is

contained within the render tree data, it is obvious to transform the ray to model space and not vice versa. Thus, an octree traversal of any kind can be carried out. We have chosen to use a parametric algorithm [11] for the implementation, as this suits the application well.
Using this information, the first three stages of the pipeline are covered.

3.2.1 Computing the Polynomial of the Hit Voxel Along the Ray

As soon as a voxel that contains a possible hit of the isosurface and happens to be at the chosen depth of the render tree has been identified, information about the polynomial along the ray needs to be gathered.
Let the ray be given by

$$R : \vec{r}(t) = \vec{s} + t \cdot \vec{d} \tag{3}$$

with $\vec{s}$ being the ray's origin, $\vec{d}$ its direction and $[t_{min}, t_{max}]$ the ray's parameter interval which corresponds to the segment lying inside the voxel.
Substituting x, y and z in the voxel's polynomial function $f(x,y,z) = l$ by the parameterised form 3, the multivariate function is converted into a univarite polynomial in parameter t: $f(t) = l$. By doing so, the polynomial function stays polynomial while the highest exponent is tripled. E.g. a tri-squared polynomial leads to a polynomial of sixth order. [3]

3.2.2 Finding the Roots of the Computed Polynomial

There are a lot of ways for calculating the roots of a polynomial. We have chosen an algorithm based on the Sturm Series [6]. The Sturm Series is a sequence of polynomials from which it is possible to compute the number of roots within an interval by counting sign changes in the series. For a closer look of this adapted binary search for root finding using Sturm Series, see algorithm 1.
We have chosen this approach, as it can identify the case when there is no root within the given interval quickly. If there is at least one root, it always needs the same number of steps to find a small interval which covers the front most root independently of the number of roots.

```
Calculate the Sturm Series for the polynomial
Evaluate number of sign changes (SC) of Sturm Series
if SC(t_min) = SC(t_max) then
  | return (no intersection)
end
while true do
  | compute the midpoint t_mid = (t_min + t_max)/2
  | if interval t_max − t_min < ε then
  |   | return t_mid
  | else if SC(t_mid) = SC(t_min) then
  |   | there is no intersection in the front half
  |   | make t_min = t_mid
  | else
  |   | there is an intersection in the front half
  |   | make t_max = t_mid
  | end
end
```

Algorithm 1: Adapted binary search for polynomial root finding using Sturm Series.

3.2.3 Final Output

If an intersection with the isosurface has been identified, the final pixel position and the normal needs to be calculated from the obtained parameter $t_{intersect}$. Evaluating the ray equation 3 using the final parameter $t_{intersect}$ leads to the final pixel position: $\vec{p} = \vec{r}(t_{intersect})$. For lighting, the normal at this point of the isosurface is also of interest. It can be obtained by calculating the gradient of the polynomial function: $\vec{n} = \nabla f(x,y,z)|_{\vec{p}}$.

4 Results

We implemented our system on top of the OpenGL graphics API running on Linux and Windows. The test framework was implemented using C++ for the CPU code and CUDA for the GPU code. The test system's Graphics Card was an NVIDIA 8800 GT 512.
During the implementation we mainly focused on feasibility and just partially optimized for performance, leaving still a lot of performance potential to be obtained using this approach.

From our first experiments we conclude the following: The basic implementation using only floating point 32 arithmetic suffers from a lot of artefacts, especially for big data sets (see figure 8). Although, current graphics cards (NVIDIA GT200) support floating point 64 arithmetic a coordinate transformation seems to be the better solution. If the polynomials are not scaled with the voxels but are transformed to a coordinate system reaching from $[-1,+1]$ for each coordinate axis, most of the image artefacts are suppressed. This idea not only influences the image quality positively, but also saves computations, as the scaling function does not have to be scaled and dilated for every single voxel. All different parts of the scaling function which contribute to one voxel can simply be precomputed and just need to be multiplied by the influence coefficient while building the render tree. The enhanced image can also be found in figure 8. For best image quality, especially when using tri-cubic and higher order basis functions, floating point 64 arithmetic is recommendable.
To get an even better image quality, especially for rays, which just pass through the corners of a voxel, additional refinements should be made. If the parameter interval of the ray adapts to the segment length lying inside the voxel, the same parameter interval corresponds to a different segment length. Especially for a very small segment length, this again introduces artefacts. One possible solution is setting the full parameter interval ($[-1,+1]$) to coincide with two planes limited by a circum-sphere

[3] This procedure suffers from high computational complexity.

	tri-linear	tri-square	tri-cubic
no optimization 64^3	1.9fps	0.9fps	0.3fps
min-max-check 64^3	10.1fps	1.9fps	0.7fps
no optimization 128^3	0.9fps	0.4fps	0.1fps
min-max-check 128^3	6.5fps	1.6fps	0.3fps

Table 1: Frame rate analysis for the Foot dataset for different resolution levels, wavelets and optimization criteria: screen resolution 800×600

around the voxel. Then the search interval has to be confined to meet the octree borders.
Even after all of these measures, some pixel holes appear at the voxel borders. This again is a problem due to the limited floating point precision. We can also take countermeasures for this problem: By increasing the search interval by a small epsilon these gaps are filled (see 8).
For best performances of this approach it is important, not to carry out highly computationally complex calculations if we can discard a voxel or a whole subtree beforehand. This is especially possible, if the dataset contains bigger spaces that do not contain samples matching the current isosurface level. One way of implementing this behaviour can be achieved by storing the minimum and maximum value contained within the subtree/voxel for each rendertree node. This min-max-check can then be carried out during the octree traversal, especially before calculating the polynomial along the ray. The performance increase of this technique is stated in table 1.

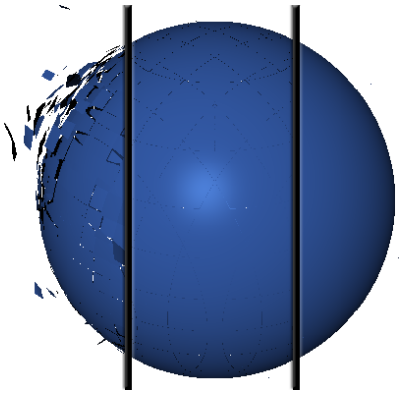
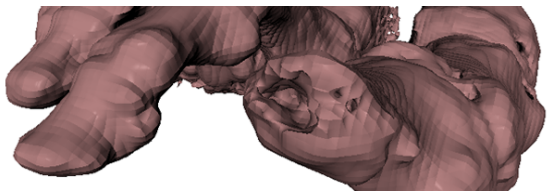


Figure 8: Artefacts for single precision floating point rendering. Left: no countermeasures; Center: uniform voxel size and 'circum-sphere' approach; Right: additional increase of intersection search interval

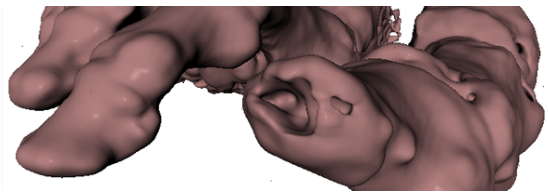
4.1 Quality of different Basis Function Orders

There is of course a difference between the use of simple wavelets and more complex ones. Figure 9 shows the results of tri-linear, tri-square and tri-cubic basis functions.

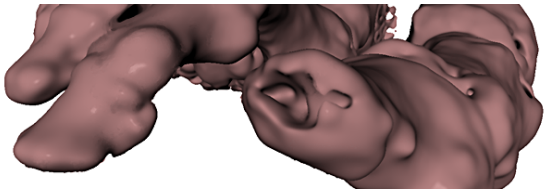
The benefit for increasing the order of the basis function is a continuity increase which is especially noticeable when changing form tri-linear to tri-squared



(a) tri-linear @ 128^3



(b) tri-square @ 128^3



(c) tri-cubic @ 128^3

Figure 9: Comparison between tri-linear, tri-square and tri-cubic basis functions. Note the continuity increase from lower to higher order basis functions. Remark: 9(c) has been median filtered to remove pixel artefacts occurring for limited floating point accuracy.

basis functions. Considering the tri-cubic case the downsides of higher order basis functions arise quickly: Due to the higher complexity introduced by larger basis functions the rendering gets significantly slower and high frequency details seem to get smoothed out a little bit.

Figure 10 shows the independence of the zoom level for tri-square basis functions, which allows close up studies of the data on the fly. A comparison of different resolution levels for tri-linear and tri-square basis functions can be found in the color plate section.

Even with only $1/8$ of the data samples the boundary of the isosurface does not change notably. Only when the resolution is decreased further, the difference might also be visible for distant views.

4.2 Performance

Table 1 gives an overview of the performance of different resolution levels as well as different basis functions. Although framerates which allow interactivity to a full extend are not yet realisable, we are certain that this approach has enough potential to meet the required frame rates for real time user interactivity[4].
While the computation time of the preprocessing step is not as important as the rendering time, it should be still

[4]We think that an integration of the 'Frustum Form'[10] into our approach will help to achieve better framerates.

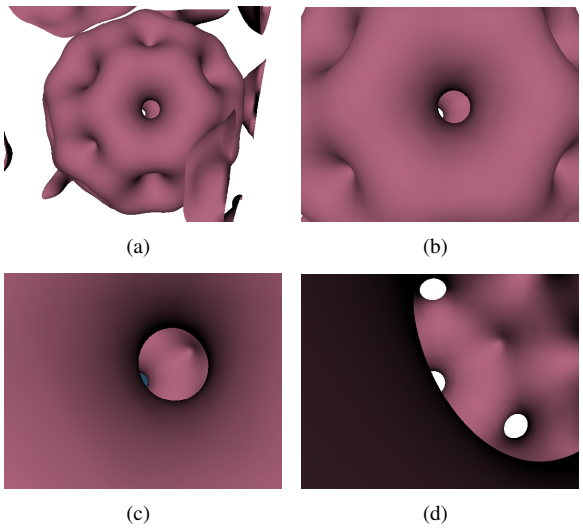


Figure 10: Different zoom levels for an isosurface of the Bucky Ball dataset. Note the smoothness of the surface independent of the zoom level.

mentioned here, that our implementation does not even require a second for this filtering task. Due to high degree of parallelism in the DWT and the render tree setup, these tasks can be completed rather quickly, even for big data sets.

5 Conclusions

So far we have shown a different approach for rendering isosurfaces, which relies on multi resolution analysis. While analysing the three dimensional data using a discrete wavelet transformation, we apply a reconstruction using continuous three dimensional B-splines. The key benefits of this idea are the zoom level independent smoothness of surfaces, data compression and perspective rendering. As the whole data is used for rendering, the isosurface level as well as the current resolution level can be changed without any further computation.

The drawbacks of this approach are of course the occurrence of pixel artefacts for higher order wavelets, which can be suppressed to a big extend by a coordinate transformation and using floating point 64 arithmetic. Although we were able to use less information to render a good view of the given data using lower resolutions, it is normally not allowed to discard this information completely, as medical visualization very strictly demands display of every detail of the data set.

To enable this approach to work to a full extend, optimizations for computationally complex parts of the algorithm should be made. Another important memory saving point is constructing the spline data online from the samples of each resolution level, while traversing the render tree. Combining these ideas with next generation graphics cards, we might be able to combine the visual benefits of this approach and realtime interactivity.

6 Acknowledgement

We would like to thank Markus Grabner for his ideas and support. All volume data has been taken from The Volume Library (http://www9.informatik.uni-erlangen.de/External/vollib/).

References

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

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

[3] Charles K. Chui. *An Introduction to Wavelets*. Academic Press, Inc., 1992.

[4] Tamal K. Dey and Joshua A. Levine. Delaunay Meshing of Isosurfaces. In *SMI '07: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, pages 241–250, Washington, DC, USA, 2007. IEEE Computer Society.

[5] 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.

[6] 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.

[7] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM SIGGRAPH Computer Graphics*, volume 21, 1987.

[8] Kwan-Liu Ma and Steven Parker. Massively Parallel Software Rendering for Visualizing Large-Scale Data Sets. *IEEE Comput. Graph. Appl.*, 21(4):72–83, 2001.

[9] Stephen R. Marschner and Richard J. Lobb. An evaluation of Reconstruction Filters for Volume Rendering. pages 100–107. IEEE Computer Society Press, 1994.

[10] Martin Reimers and Johan Seland. Ray Casting Algebraic Surfaces using the Frustum Form. *Computer Graphics Forum*, 27(2):361–370, April 2008.

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

[12] Eirc J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann Publishers, Inc., 1996.

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

[14] Ingo Wald, Heiko Friedrich, Gerd Marmitt, and Hans-Peter Seidel. Faster Isosurface Ray Tracing Using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005. Member-Philipp Slusallek.

[15] 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.