IEEE/EG International Symposium on Volume Graphics (2010)
R. Westermann and G. Kindlmann (Editors)

Wavelet-based Multiresolution Isosurface Rendering

Markus Steinberger and Markus Grabner

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

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.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

1. Introduction

Volumetric datasets are a widely used representation of three-dimensional data in a variety of applications, such as medical visualization, engineering, and computer games. Direct volume rendering [EHK*06] is a suitable approach to display volumetric data in many cases. 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.

A convenient form of volumetric data is a *voxel grid*, which is a set of samples on a regular grid in 3D-space. For a well-defined isosurface to exist, we 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

$$f(x,y,z) = \sum_{0 \le i,j,k \le d} f_{ijk} x^i y^j z^k = 0, \qquad (1)$$

with f_{ijk} given and d being the surface's degree along each dimension, resulting in an overall degree of $3d$.

Due to the advance of data acquisition technologies such as computed tomography (CT), the datasets investigated by researchers become increasingly larger. 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 simplifying the data, isosurface rendering remains a demanding task. 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 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. 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 (in particular loop unrolling), which we accomplish by an advanced preprocessing mechanism.

2. Related work

Ray casting algebraic surfaces [Han83] is traditionally related to rendering a single polynomial, for which the task can be greatly accelerated [RS08]. Organizing piecewise polynomial functions (splines) in tetrahedrons offers the abil-

DOI: 10.2312/VG/VG10/013-020

ity to render more complex surfaces of low degree [LB06]. This approach can even be extended to isosurfaces of volume datasets when an appropriate transformation step is applied [KOR08,KZ08]. Unfortunately this limits the approach to volumes of smaller extent, as the memory footprint of the data is strongly increased.

These approaches are somewhat related to the well-known Marching Cubes algorithm [LC87], which also tries to extract a surface description – in this case a triangle mesh – for a given isovalue. Follow-up methods [TPG99, ABJ05] achieve better quality and are still used frequently.

Direct isosurface renderers allow the user to interactively modify the isovalue. Mostly based on shaders, hardware features are exploited such as fast texture filtering or rasterization of bounding geometry. 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]. This direct isosurface renderer directly works on raw voxel data, which makes it a good candidate for integration into a mixed direct volume and isosurface rendering system.

A common problem in volume rendering is the huge amount of data required for high-quality rendering. One possible solution is compression, e.g., by using wavelets [Chu92, SDS96], which is well covered in the literature [Wes94, GLDK95, KS99]. If the wavelet transform is used to reconstruct a mixed resolution representation, areas of interest can be displayed in more detail. Special care needs to be taken at the boundaries between different resolutions to avoid cracks in the surface. Thus wavelets are often used on previously extracted surfaces and not directly on the volume dataset [GSG96, WKE99, BDHJ00, LHJ07].

Resolution boundaries in volume datasets are easily handled with linear wavelets [UHP00]. The reduced quality of linear filtering is less apparent when constructing surface normals 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]. Given a multiresolution description of the data, a good out-of-core memory management strategy is desired for rendering volumes of arbitrary size. Recently Crassin et al. showed a shader based approach [CNLE09] for direct volume rendering, on how to use GPU memory as efficiently as possible whilst reporting information back from the rendering process itself.

3. Our method

The core of our method is an algebraic surface renderer, which is embedded in a framework that can handle piecewise algebraic surfaces and therefore is able to render isosurfaces of voxel datasets.

3.1. Rendering algebraic surfaces

We begin the discussion with a general overview of algebraic surface rendering. The two core steps of common algebraic surface raycasters, which typically also form the bottleneck for higher order surfaces, are *composition* and *rootfinding*. Given a viewing ray

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{v}, \tag{2}$$

one is interested in the description of f along the ray: $(f \circ \mathbf{r})(t)$ (composition). After subtracting the isovalue c, a rootfinder is used to identify the first root position t_0 of the polynomial $(f \circ \mathbf{r})(t) - c = 0$, which represents the intersection of the viewing ray and the isosurface. We offer an efficient approach for the composition step, which shows a tremendous speedup in comparison to naive implementations.

3.2. Composition of a univariate polynomial

The composition $(f \circ \mathbf{r})(t)$ transforms a trivariate polynomial of maximum degree $3d$ (e.g., tri-linear, tri-square...), into a univariate polynomial. A straightforward implementation of the composition in power form is the substitution of the ray equation (2) into the surface definition (1). The expansion of this equation yields an expression for all polynomial coefficients. To reduce the number of computations, one can extract terms appearing more often and only compute them once. Still this approach needs a lot of operations and too many registers to fit into a GPU based framework (see Table 1). To obtain a more efficient structure, we write the composition with convolution operators [SR03]:

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

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 (2), $\mathbf{r}_y$, $\mathbf{r}_z$ defined likewise, and $\mathbf{f}$ being the coefficient sequence of the resulting polynomial $f(t)$ of degree $3d$. The power expressions are n-fold convolutions, which are computed incrementally, i.e., $\mathbf{r}_x^i = \mathbf{r}_x^{i-1} * \mathbf{r}_x$ (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

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

An important property of this structure is the fact that no more than one factor $\mathbf{f}_{ij}$ and $\mathbf{f}_i$ needs to be available at the same time (i.e., the memory used to store $\mathbf{f}_{00}$ can be reused to store $\mathbf{f}_{01}$, and so on), leading to less consumption of register space and therefore better parallelism.

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 through the rendering pipeline, better stability for the composition, and also avoids basis changes from power form to Bernstein form for root finding:

$$\begin{aligned}\widetilde{\mathbf{f}} &= \sum_{i,j,k=0}^{d} \widetilde{b}_{ijk} \cdot \widetilde{\mathbf{q}}_x^{d-i} * \widetilde{\mathbf{u}}_x^{i} * \widetilde{\mathbf{q}}_y^{d-j} * \widetilde{\mathbf{u}}_y^{j} * \widetilde{\mathbf{q}}_z^{d-k} * \widetilde{\mathbf{u}}_z^{k} \\ &= \sum_{i=0}^{d} \widetilde{\mathbf{q}}_x^{d-i} * \widetilde{\mathbf{u}}_x^{i} * \sum_{j=0}^{d} \widetilde{\mathbf{q}}_y^{d-j} * \widetilde{\mathbf{u}}_y^{j} * \sum_{k=0}^{d} \widetilde{b}_{ijk} \widetilde{\mathbf{q}}_z^{d-k} * \widetilde{\mathbf{u}}_z^{k}, \quad (3)\end{aligned}$$

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 $\widetilde{\mathbf{c}}$ can be obtained in a recursive manner (similar to the Horner scheme for the power form):

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

Applying this substitution to the three sum terms in eq. (3) saves a large amount of computational work. However, the expression $\widetilde{\mathbf{q}}^{d-k} * \widetilde{\mathbf{u}}^{k}$ in the sum over k is computed repeatedly, so precomputing and reusing it (for $k = 0 \ldots d$) further reduces the number of operations. The same argument applies to the sum over j. See Table 1 for an overview of the different methods. The difference of 2 to 3 orders of magnitude in performance between the naive approach and our proposed structure is due to the fact that our structure easily fits into the register space of the GPU. See Section 4 for more results, including a comparison with other methods [SH05, RS08].

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 1: *Comparison between different algorithms for trivariate to univariate composition $(f \circ \boldsymbol{r})(t)$ of random polynomial data. 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.*

3.3. Root finding

The first intersection of a viewing ray with the isosurface corresponds to the first root of the polynomial $f(t) - c$. We focus on 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.

The Bernstein form offers a control polygon for a given polynomial, which has variation diminishing property. [LR81] is based on the idea of subdividing the control polygon using the de Casteljau algorithm at the intersection of the control polygon with the zero axis. This algorithm can be implemented with constant memory usage and without index operations. [LR81] shows quadratic convergence speed with quadratic complexity for each step due to the de Casteljau algorithm. For volume rendering, a fixed number of 3 to 7 iterations, depending on the chosen interpolation type, seems to be sufficient and does not produce any visible artifacts.

On the other hand, [MR07] is based on knot insertion, which requires index operations, and the used memory grows with every iteration step, limiting the maximum number of iterations. [MR07] shows linear convergence speed with linear complexity. Our optimized version of this algorithm is more than 10 times slower than our [LR81] implementation.

A root isolated from all others can be found using simpler methods, like regula falsi or binary search. Our implementation of this root finder combines both to guarantee exponential convergence for the second phase of the rootfinder. Although the second phase of this algorithm outperforms our [LR81] implementation when a root has been isolated, the isolation step itself is (due to diverging branches) not as well suited for the CUDA architecture as our [LR81] implementation.

Root finding is not limited to polynomials in Bernstein form only, and as we are able to formulate all other parts of the pipeline using power form, we also implemented a root-finder in power form. We use the Sturm Series, which can be used to count multiplicities of roots in a given interval. When subdividing the search interval in a binary manner, one can also identify an exponential convergence with constant memory consumption. The Sturm Series, which is computed only at the beginning of the algorithm, is 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, are getting more severe for more iterations. The second down side of the algorithm is its high memory consumption, as the whole Sturm Series needs to be kept in memory during the entire execution.

For a comparison between the root-finders see Table 2. We suggest to use the [LR81] implementation, as it shows the best overall characteristics of the tested rootfinders.

3.4. Rendering many trivariate polynomials

As the size of common voxel datasets 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 multiresolution approach naturally arises. Still

method	[LR81]	[MR07]	RF	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 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.*

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.

Our approach shows good frame rates for this scenario, as every ray can work through the whole rendering pipeline independently from all others. One problem when tackling huge datasets with algebraic methods is storing the explicit description of every single polynomial in memory, which would increase the memory consumption by the factor $(d+1)^3$. Normally, one just desires to interpolate the given data, for which it is possible to work directly with the original representation. When a different description is needed, the data describing the current voxel can be transformed on-the-fly.

In our case, however, this transformation corresponds to a basis transform only. 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 basis vector with a matrix of size $(d+1)\times(d+1)$. Aiming at matrices containing several zeros and ones, the matrices can be scaled with a constant factor, thus reducing the number of required operations. For a one dimensional basis transform only 6 and 8 multiplications are needed for transition to scaled Bernstein basis and power basis, respectively. The N-dimensional basis transform 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 transform can be efficiently integrated in our Horner Scheme for composition, which also shows this separation of single dimensions.

For handling big datasets, knowledge of the underlying data and the location in memory is required. We store all meta information in an octree. Similarly to [WFM*05], every node stores a conservative estimate of the minimum and maximum value appearing in its subtree to quickly identify regions which do not contain the isosurface (empty space skipping). Each leaf corresponds to a group of b^3 adjacent polynomials, where b is a trade-off factor between more efficient empty space skipping and memory requirement of the tree structure. The min/max values at leaf nodes can be estimated using the convex combination property of the B-spline basis 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].

3.5. Wavelet transform

For large datasets, it is not useful to display the entire data at full resolution. Depending on the current view, the resolution of distant parts can be decreased to improve rendering performance (see Figure 1 and 6 and Table 3) with little impact on visual quality.

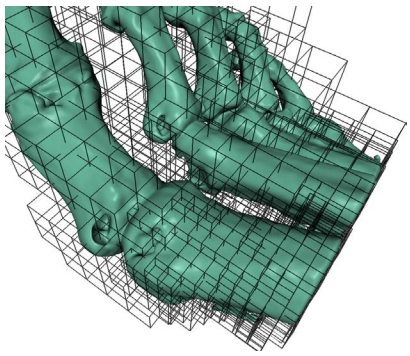


Figure 1: *View-dependent multiresolution scene spanning several levels of the wavelet hierarchy with octree nodes superimposed (indicated by the wireframe cubes).*

To enable a multiresolution data description, we perform a wavelet transform on the volume dataset using polynomial spline wavelets [CDF92] with different degrees and the least possible number of vanishing moments to keep the support of the filters as small as possible. Since the basis functions (i.e., wavelets) are B-splines, they naturally fit into the above discussion regarding rendering of piecewise algebraic surfaces. Moreover, due to the continuity properties of B-spline wavelets, the reconstruction (and hence the isosurface) is guaranteed to be continuous as well, thus avoiding cracks in the surface.

An approximation of the original data is a linear combination of a subset of the available basis functions, weighted by the corresponding wavelet coefficients. The levels of the octree form a set of nested vector spaces (much like the wavelets themselves [SDS96]). Hence octree nodes (starting from the root) are expanded as needed until the vector space spanned by the currently expanded nodes contains the desired approximation.

However, a straightforward implementation would be inefficient for two reasons. First, distributing spline coefficients within a wavelet's support into all intersecting octree nodes 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 octree hierarchy and *gathering* all contributions of wavelets which intersect the octree node under consideration. As the number of coefficients influencing a single node grows cubicly with the support of the one dimensional wavelet (e.g., 175 influence coefficients for the tri-cubic case), we make use of the separability of the B-spline wavelet basis and sequentially perform the transformation in x-, y-, and z-direction. This reduces the computation 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).

A second issue is the granularity at which the wavelet reconstruction is carried out. Although possible, it is not feasible to keep track of activation and influence of every single wavelet. Only tasks which can be subdivided into a sufficient number of similar subtasks can be implemented with maximum performance on SIMD hardware. We therefore operate on chunks of $k \times k \times k$ nodes (we choose $k = 8$) and evaluate the view-dependent error metric for every chunk (such that all nodes within the chunk can be processed efficiently in a similar way).

One CUDA kernel is used for the convolution along each dimension, the first kernel is additionally responsible for evaluating the error metric and checking the neighborhood along the other two dimensions to analyze if this chunk needs to be reconstructed. We use a simple error metric which activates a chunk if it can possibly contain the current isovalue (min/max comparison with rendertree entry) and 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.

Nodes which are not activated do not add any data to the reconstruction, and their detail coefficients can be set to zero. If all detail coefficients contributing to a chunk are 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.

The CUDA kernels for all three dimensions each consist of blocks of $k \times k$ threads. Each thread computes a one-dimensional convolution with the wavelet filter kernels (low pass and high pass). One block is responsible for a beam of $k \times k$ voxels along the current convolution direction. Reconstruction is applied to one chunk after the other, deactivated chunks are simply skipped (see Figure 2). 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.

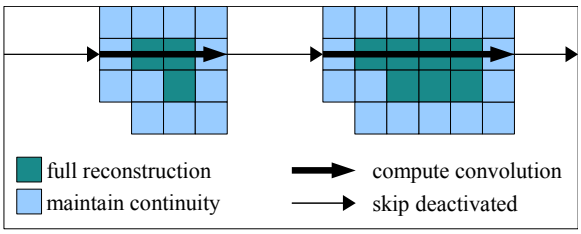


Figure 2: *Processing scheme for selective wavelet reconstruction, the arrows indicate the execution of a single thread block in the CUDA kernel. Boxes do not refer to octree nodes, but to chunks of $k \times k \times k$ nodes as explained in Section 3.5.*

3.6. Code optimization

A key factor for good performance is full utilization of computational resources, which in the CUDA context includes the necessity of high arithmetic intensity [LNOM08]. A straightforward implementation of the equations in Section 3.2 yields code containing several nested loops with the trip count known in advance. However, while the CUDA compiler is designed to detect such static program flow constructs and optimize the code accordingly (e.g., unroll loops), it fails to do so in this particular case due to the dependency of inner loop limits on outer loop indices.

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 (e.g., polynomial coefficients) are stored in off-chip 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. It manipulates the code similar to the standard C preprocessor, but also supports loop control structures. 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 for tri-cubic rendering. After optimization, it only contributes $30 \pm 5\%$ to the frame time, overall performance has been improved by a factor of approximately 17. The remaining $70 \pm 5\%$ are mainly made up of traversal and data transfer time.

4. Results

The test framework for our implementation consisted of an Intel Core i7 and an NVIDIA GeForce GTX 285 with 1GB of graphics memory running on both Microsoft Windows Vista and Linux.

4.1. Single algebraic surface rendering

Although our technique for rendering piecewise algebraic surfaces is not optimized for rendering a surface described by a single equation, we still want to draw a comparison to the frustum form approach. [RS08] showed that several operations can be saved when pre-computing common terms for all rays hitting the same polynomial, which can be efficiently implemented by means of matrix multiplications. Figure 4 shows a comparison to our approach for different view port sizes as well as different polynomial degrees. We believe the main reason for our approach outperforming the frustum form is found in better resource utilization.

4.2. Isosurface rendering for volume datasets

When it comes to interactive isosurface rendering of volume datasets, [HSS*05] is – as far as we know – still the reference. Their approach relies on fast tri-cubic texture sampling [SH05], which is constructed using fixed-function hardware supported tri-linear texture sampling. Root-finding is accomplished by drawing samples along the ray and refining the (thus isolated) root using binary search. The step length for the isolation part of the algorithm is chosen as high as possible, depending on the maximum frequency occurring in the data. Our approach, on the other hand, needs to check every voxel that can possibly contain a hit. We also benefit from low frequency data using the wavelet transform to reduce the number of voxels if possible. The rendering performance scales linearly with the number of voxels that are checked.

We tested our approach with various examples, showing interactive frame rates for large datasets (see Table 3 and Figure 5). Tricubic texture sampling outperforms our method, but 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, where a few voxels make up for a big part of the screen area. In these situations, 9 bits are often not sufficient for high quality renderings, introducing artifacts which may be distracting or misleading (see Figure 3). The quality of our approach, on the other hand, is independent of the voxel sizes, it is even faster for bigger voxels.

5. Conclusions and future work

We demonstrated the suitability of spline wavelets for the representation and direct rendering of multiresolution piecewise algebraic surfaces in the CUDA framework. 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 storing the wavelet coefficients proved to be an efficient way to select relevant basis functions.

	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 3: *Performance measurement for datasets from Figure 5 and 6: Viewport size* 1024×768*; first three methods correspond to our approach using different degree for interpolation (no multiresolution applied); sampling is our implementation of [SH05] using our octree structure for empty space skipping. The relatively low frame rates for sampling approach occur as it has been configured for high visual quality – still some difference to the true solution is visible.*

	full	6(b)	6(c)	6(d)	6(e)
reconst.	7.23	8.72	9.25	6.11	6.92
rendering	225	219	161	115	89.3
MSE	0	24	45	57	176
chunks	100%	73%	40%	12%	6%

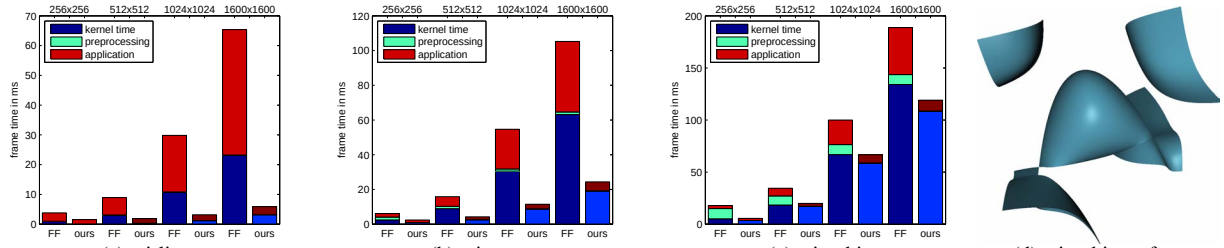
Table 4: *Performance measurement for view dependent wavelet selection of datasets from Figure 6. Reconstruction and rendering times are given in ms, MSE is scaled by* 10^3 *and calculated over all colored pixels. Fully reconstructed Dragon dataset* 128^3 *consists of a total of* 4681 *chunks.*

The actual ray/surface intersection is performed in 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 method is more accurate (though slower) than a texture hardware based intersection algorithm [SH05], and it is more efficient than a dedicated algebraic surface renderer [RS08] in a typical volume rendering configuration.

Data organization in an octree opens a number of opportunities for future improvements. Since irrelevant subtrees are pruned during traversal, it is not necessary to upload the entire tree to GPU memory at once, only those parts required for the current frame have to be available. If transmission cannot keep up with the rate at which data are 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

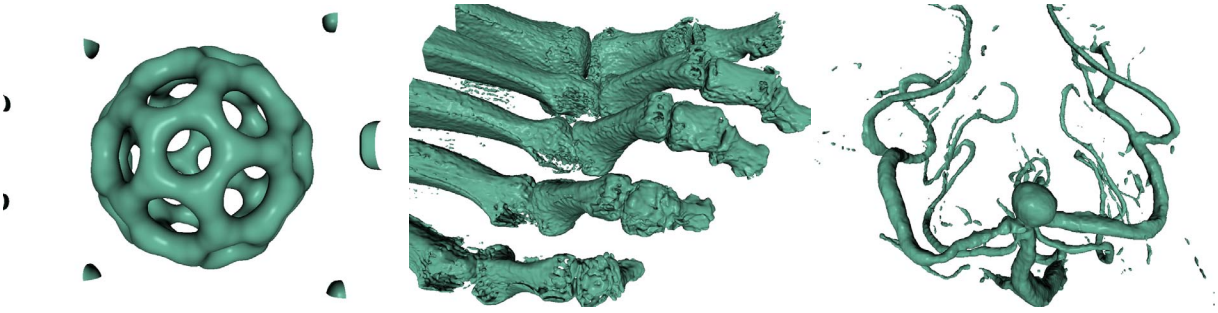
We would like to thank Johan Seland (SINTEF ICT, Norway) for providing the implementation of their method [RS08]. This work was partially sponsored by the Austrian Science Fund FWF under contract W1209.



(a) tri-linear (b) tri-square (c) tri-cubic (d) tri-cubic surface

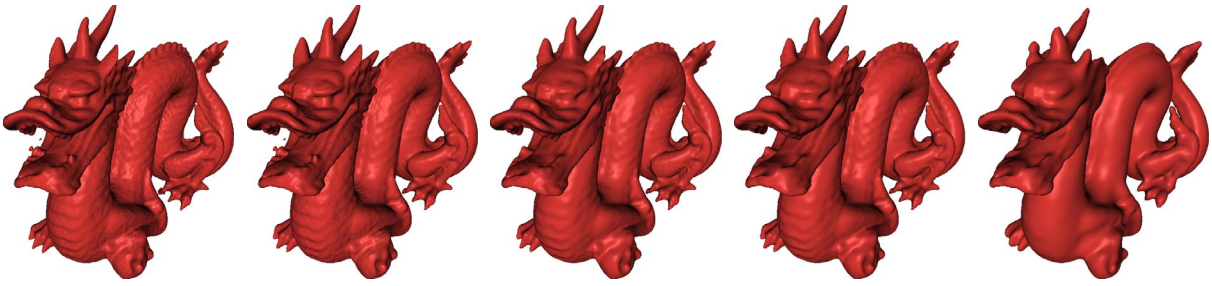
(a): $x+y+z+xyz-1$ (b): $4(x^2+y^2+z^2)+16xyz+(xyz)^2-1$ (c,d): $x^2+y^2+z^2-(xyz)^3-x^2y-1$

Figure 4: *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.*



(a) Bucky Ball 32^3, tri-cubic (b) Foot Dataset 256^3, tri-cubic (c) Vertebra 512^3, tri-cubic

Figure 5: *Example of different sized volume datasets used for evaluation. See Table 3 for a performance comparison.*



(a) 100% (b) 73.1% (c) 40.0% (d) 12.5% (e) 6.3%

Figure 6: *Dragon dataset: View dependent Wavelet Reconstruction greatly reduces rendering time (see Table 4), while remaining good visual quality. Note: higher quality reconstruction with the same amount of data could be accomplished, but this would have less influence on rendering speed.*

References

[ABJ05] ANDERSON J. C., BENNETT J., JOY K. I.: Marching Diamonds for Unstructured Meshes. In *IEEE Visualization 2005* (Oct. 2005), pp. 423–429. 2

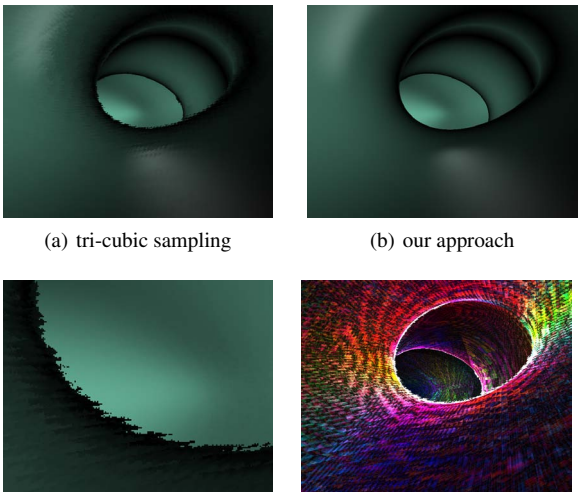
[BDHJ00] BERTRAM M., DUCHAINEAU M. A., HAMANN B., JOY K. I.: Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. In *Proceedings of IEEE Visualization 2000* (Oct. 2000), Ertl T., Hamann B., Varshney A., (Eds.), IEEE, IEEE Computer Society Press, pp. 389–396. 2

[CDF92] COHEN A., DAUBECHIES I., FEAUVEAU J.-C.: Biorthogonal bases of compactly supported wavelets. *Comm. Pure Appl. Math. XLV* (1992), 485–560. 4

[Chu92] CHUI C. K.: *An Introduction to Wavelets*, vol. 1 of *Wavelet Analysis and its Applications*. Academic Press, San Diego, CA, USA, 1992. 1, 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics, SI3D 2009, February 27 - March 1, 2009, Boston, Massachusetts, USA* (2009), Haines E., McGuire M., Aliaga D. G., Oliveira M. M., Spencer S. N., (Eds.), ACM, pp. 15–22. 2

[EHK*06] ENGEL K., HADWIGER M., KNISS J., REZK-



(a) tri-cubic sampling (b) our approach

(c) closeup view (d) difference

Figure 3: *Vertebra dataset (see Figure 5(c)); 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).*

SALAMA C., WEISKOPF D.: *Real-Time Volume Graphics*. A K Peters, Ltd., 2006. ISBN 1-56881-266-3. 1

[FR87] FAROUKI R. T., RAJAN V. T.: On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design 4*, 3 (1987), 191–216. 2

[GLDK95] GROSS M., LIPPERT L., DREGER A., KOCH R.: A new method to approximate the volume rendering equation using wavelet bases and piecewise polynomials. *Computers & Graphics 19* (1995), 47–62. 2

[GSG96] GROSS M. H., STAADT O. G., GATTI R.: Efficient triangular surface approximations using wavelets and quadtree data structures. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996), 130–143. 2

[Han83] HANRAHAN P.: Ray tracing algebraic surfaces. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1983), ACM, pp. 83–90. 1

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M. H.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Comput. Graph. Forum 24*, 3 (2005), 303–312. 2, 6

[KOR08] KLOETZLI J., OLANO M., RHEINGANS P.: Interactive volume isosurface rendering using BT volumes. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 45–52. 2

[KS99] KIM T.-Y., SHIN Y. G.: An efficient wavelet-based compression method for volume rendering. *Computer Graphics and Applications, Pacific Conference on 0* (1999), 147. 2

[KWH09] KNOLL A. M., WALD I., HANSEN C. D.: Coherent multiresolution isosurface ray tracing. *The Visual Computer 25*, 3 (3 2009), 209–225. 2

[KZ08] KALBE T., ZEILFELDER F.: Hardware-accelerated, high-quality rendering based on trivariate splines approximating volume data. *Comput. Graph. Forum 27*, 2 (2008), 331–340. 2

[LB06] LOOP C., BLINN J.: Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph. 25*, 3 (2006), 664–670. 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 163–169. 2

[LHJ07] LINSEN L., HAMANN B., JOY K.: Wavelets for adaptively refined $\sqrt[3]{2}$ subdivision meshes. *International Journal of Computers and Applications 29*, 3 (2007), 223–231. 2

[LNOM08] LINDHOLM E., NICKOLLS J., OBERMAN S., MONTRYM J.: NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (Mar./Apr. 2008), 39–55. 1, 5

[LR81] LANE J. M., RIESENFELD R. F.: Bounds on a polynomial. *BIT Numerical Mathematics 21* (1981), 112–117. 3, 4, 7

[LS75] LAL M., SINGH H. ARD PANWAR R.: Sturm test algorithm for digital computer. *Circuits and Systems, IEEE Transactions on 22*, 1 (Jan 1975), 62–63. 3

[MR07] MØRKEN K., REIMERS M.: An unconditionally convergent method for computing zeros of splines and polynomials. *Math. of Comp. 76* (2007), 845–865. 3, 4

[NVI08] NVIDIA: *NVIDIA CUDA Programming Guide 2.0*. 2008. 6

[RS08] REIMERS M., SELAND J.: Ray casting algebraic surfaces using the frustum form. *Computer Graphics Forum 27*, 2 (2008), 361–370. 1, 3, 6, 7

[RUL00] REVELLES J., UREÑA C., LASTRA M.: An efficient parametric algorithm for octree traversal. In *Journal of WSCG* (2000), pp. 212–219. 4

[SDS96] STOLLNITZ E. J., DEROSE T. D., SALESIN D. H.: *Wavelets for Computer Graphics: Theory and Applications*. Morgann Kaufmann, San Francisco, CA, 1996. 2, 4

[SH05] SIGG C., HADWIGER M.: Fast third-order texture filtering. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, 2005, ch. 20, pp. 313–329. 2, 3, 6, 8

[SR03] SÁNCHEZ-REYES J.: Algebraic manipulation in the Bernstein form made simple via convolutions. *Computer-Aided Design 35*, 10 (2003), 959–967. 1, 2

[TPG99] TREECE G. M., PRAGER R. W., GEE A. H.: Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics 23* (1999), 583–598. 2

[UHP00] UDESHI T., HUDSON R., PAPKA M. E.: *Seamless Multiresolution Isosurfaces Using Wavelets*. Tech. Rep. ANL/MCS-P801-0300, Argonne National Laboratory, 2000. 2

[WB97] WONG P. C., BERGERON R. D.: Performance evaluation of multiresolution isosurface rendering. In *DAGSTUHL '97: Proceedings of the Conference on Scientific Visualization* (Washington, DC, USA, 1997), IEEE Computer Society, p. 322. 2

[Wes94] WESTERMANN R.: A multiresolution framework for volume rendering. In *Symposium On Volume Visualization* (1994), ACM Press, pp. 51–58. 2

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–572. 4

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