Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Chaos and Graphics

Efficient volume rendering on the body centered cubic lattice using box splines

Bernhard Finkbeiner[a,*], Alireza Entezari[d], Dimitri Van De Ville[b,c], Torsten Möller[a]

[a] School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, BC, Canada V5A 1S6
[b] Institute of Bioengineering, Ecole Polytechnique Fédérale de Lausanne, Station 17, CH-1015 Lausanne, Switzerland
[c] Department of Radiology, University of Geneva, Rue Micheli-du-Crest 24, CH-1211 Geneva 14, Switzerland
[d] CISE Department, E301 CSE Building, University of Florida, P.O. Box 116120, Gainesville, FL 32611-6120, USA

ARTICLE INFO

Article history:
Received 21 August 2009
Received in revised form
31 December 2009
Accepted 4 February 2010

Keywords:
Reconstruction
Volume rendering
Box splines
Body centered cubic lattice

ABSTRACT

We demonstrate that non-separable box splines deployed on body centered cubic lattices (BCC) are suitable for fast evaluation on present graphics hardware. Therefore, we develop the linear and quintic box splines using a piecewise polynomial (pp)-form as opposed to their currently known basis (B)-form. The pp-form lends itself to efficient evaluation methods such as de Boor's algorithm for splines in box splines basis. Further on, we offer a comparison of quintic box splines with the only other interactive rendering available on BCC lattices that is based on separable kernels for interleaved Cartesian cubic (CC) lattices. While quintic box splines result in superior quality, interleaved CC lattices are still faster, since they can take advantage of the highly optimized circuitry for CC lattices, as it is the case in graphics hardware nowadays. This result is valid with and without prefiltering. Experimental results are shown for both a synthetic phantom and data from optical projection tomography. We provide shader code to ease the adaptation of box splines for the practitioner.

1. Introduction

Volumetric data are typically associated to Cartesian cubic (CC) lattices; i.e., samples are taken on an orthogonal grid, usually with equal spacing in all dimensions. These lattices are easy to use since indexing, interpolation, and representation can be done conveniently in a separable way, that is, dimension by dimension. Despite their common use for volumetric data, it is known that CC lattices are not the optimal [32]. For example, assuming a radially symmetric power spectrum, the best periodic lattice corresponds to the one with the best sphere-packing property in the frequency domain; i.e., the face-centered cubic (FCC) lattice in 3D that corresponds to the body-centered cubic (BCC) lattice in the spatial domain [38].

The BCC lattice consists of a CC lattice with an additional sample added to each cube. The FCC lattice consists of a CC lattice with an additional sample added to each face of a cube. According to the Fourier scaling property, a sparse grid spacing in one domain yields a dense spacing in the dual domain. Thus, if one can pack the replicated frequency spectra in the Fourier domain as close as possible, the sparsest grid spacing in spatial domain will be obtained without any (pre-)aliasing. Therefore, using a BCC lattice instead of a CC lattice can reduce the number of samples by 29% without any loss of information. This directly translates into a reduction of storage and computational cost.

Although the optimality of the BCC lattice has been known for a long time, appropriate kernels for 3D volume rendering that (1) guarantee approximation order; (2) are easy to use; (3) are numerically stable; and (4) allow an efficient implementation, have only been proposed recently. In earlier work [38,14], a practical method was proposed using CC-kernels for the two interleaved CC lattices that constitute the BCC lattice. This allowed for a simple and efficient hardware implementation, but dealing with the BCC structure by separate CC-kernels does not exploit the specificity of the neighborhood information on BCC. Using box splines, Entezari et al. [19,21] have introduced a class of basis functions specially tailored to the geometry of the BCC lattice. These box splines possess attractive theoretical properties for reconstruction of data on the BCC lattice.

The contributions of this paper are:

- We present an efficient algorithm for convolution of BCC-sampled data with the linear (C^0) and quintic (C^2) box splines. Specifically, Entezari et al. [21] characterize the box spline basis functions (i.e., the B-form); here we demonstrate how to efficiently implement the convolution of BCC data with these box splines. Our method evaluates the reconstructed spline

* Corresponding author.
E-mail addresses: bfa4@cs.sfu.ca (B. Finkbeiner), entezari@cise.ufl.edu (A. Entezari), dimitri.vandeville@epfl.ch (D. Van De Ville), torsten@cs.sfu.ca (T. Möller).

0097-8493/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2010.02.002

function in the so-called piecewise polynomial (i.e., pp-) form. The quintic polynomials are represented in power form which benefit from a (partial) factorization into quadratic and cubic polynomials. The pp-form allows for the fastest evaluation scheme at arbitrary points, a prerequisite for ray casting as used in volume rendering. The optimized pp-form evaluation presented in Section 3 allows us to achieve competitive interactive frame rates utilizing the GPU.

- We compare the box splines with traditional CC B-splines and with prefiltered BCC B-splines [14] in order to determine which one provides the better quality considering the implementation penalty. To ensure fairness in quality comparisons, we employ the generalized interpolation of Blu et al. [2] which is commonly employed (see [7,8,14,13]) as a prefiltering for reconstruction. We assert the better image quality of prefiltering for all methods, but we also see that quintic box splines provide us with the best reconstruction quality with and without prefiltering. Under present hardware implementations using dedicated circuitry for CC lattices, the method based on the separation of the BCC lattices into two CC lattices is still faster.

In Section 5, we verify our results using synthetic and real data which illustrate the theoretical and practical advantages of the BCC lattice in combination with box splines. We also demonstrate that the prefiltering solution for the quintic box spline allows interactive high-quality volume rendering of the BCC lattice.

2. Previous work

CC-sampled data have traditionally been the main focus of research in visualization. The common practice is to extend the univariate reconstruction algorithms to the trivariate setting by a tensor-product approach [31,5,15,30]. Marschner and Lobb [27] developed a framework for evaluation of these commonly used reconstruction techniques for volumetric reconstruction. In contrast to tensor-product reconstruction, Rössl et al. [33] proposed so-called super splines as a local interpolation model for CC-sampled data; they subdivided each cube into 24 tetrahedra and fit a quadratic polynomial for each tetrahedron. This domain partition and the polynomial degree allows for a C^1 reconstruction; however, the approximation order is limited to two. Their construction allows a fast GPU-based implementation [25] that is suitable for iso-surface rendering. A non-separable box spline approach for Cartesian data reconstruction was also presented in [18].

Over the past years, there has been an increased focus on optimal sampling lattices. Theußl et al. [38] assumed samples on the BCC lattice together with a spherical extension of reconstruction kernels; they achieve the same quality as a CC representation with fewer samples. However, the quality of the images rendered with this approach was rather unsatisfying since the images were blurry. Since then, several more appropriate kernels have been suggested: box splines [19,21], a prefiltering operator followed by a Gaussian filter [10] and BCC-splines [11,20].

The implementation of the box splines [19] turned out to be inefficient and numerically instable due to the implicit recursive representation of the box spline. The Gaussian kernel approach does not guarantee approximation order [10] and disregards the geometry of the neighborhood information of BCC lattice points. Recently, BCC-splines [11,20] have been proposed as a generalization of 2D hex-splines [39]. However, their constituting polynomial patches have not (yet) been characterized analytically and therefore their application remains limited. These methods are not able to render non-trivial data in real-time. In recent work, Entezari et al. [21] derived an explicit polynomial representation of their proposed box splines which can be used to efficiently evaluate C^0 and C^2 filter kernels for the BCC lattice. Due to fewer samples needed compared to the CC lattice (4 instead of 8 and 32 instead of 64 samples for C^0 and C^2 filtering, respectively) a speed up of a factor of two was achieved. While their approach allowed efficient evaluation of the box spline kernels, it did not offer insights on how to convolve the BCC-sampled data with these box splines, which is the main contribution of our paper. Their work did not include a GPU implementation and therefore the frame rates were non-interactive. Further on, proper prefiltering for box splines and its (efficient) implementation has not been considered.

Mattausch [28] was the first to use BCC lattices in real-time volume rendering employing commodity graphics hardware. However, this approach led to ambiguous results; e.g., the suggested sheared trilinear interpolation resulted in view-dependent artifacts. Csébfalvi et al. [14] introduced prefiltered B-spline reconstruction that is theoretically adapted for each of the CC lattices, but not for the BCC one. Specifically, separable B-splines do not form a Riesz basis on the BCC lattice which leads to an ambiguous representation; i.e., several sets of coefficients can represent the same signal. Moreover, separation of BCC lattices into two CC lattices bears the risk of ignoring high-frequency components on the BCC lattice thus leading to aliasing artifacts if the dataset is not sampled at high enough resolutions.

Consider the following example of a BCC dataset, where the primary CC lattice points are all set to 1, and the secondary CC lattice points are all set to 0. In 1D, it is equivalent to every second sample being 1. A proper Reisz basis such as the box spline solution ought to recover the oscillatory nature of the signal. In fact, the first order solution (nearest neighbor interpolation) as well as box splines will recover the oscillatory nature of this signal. However, when this dataset is separated into two CC lattices, a B-spline on each sub-lattice will recover a constant function (due to the partition of unity). The average of the two separately reconstructed signals will again result in a constant function reconstructed on the full BCC lattice.

Nevertheless, they presented appealing visual results for some datasets and this method is, to our knowledge, the best existing algorithm that combines real-time frame rates and high image quality on the BCC lattice. Therefore, we will compare our new method to Csébfalvi et al.'s [14] method in Section 5.

Both, box splines [22,17] and prefiltered B-splines [9] have also been employed for quasi-interpolation on the BCC lattice.

The prefilters proposed in [9] do not remedy the aforementioned problem. The AC prefilter modifies the values on each lattice separately, so it does not resolve the above issue. Similarly, the AB prefilter will result on the same values on all primary lattice points, and have a different value (negative) on all secondary lattice points. The reconstruction of each lattice will lead again to a constant signal, so will their average.

The superior visual quality of BCC lattices in volume rendering has also been shown by Meng et al. [29]. They confirmed that the theoretical bound of needing 30% fewer samples compared to a CC lattice also results in no loss of visual quality.

Using graphics hardware is the standard approach to achieve real-time rendering of volumetric data on CC lattices. Methods can be split into two classes: slice-based approaches [4] that sample a 3D texture using polygons that intersect the volume; and ray casting [34,26] where rays are cast through a volume that gets sampled at several points along the rays. Whereas the first approach is well adjusted to the graphics pipeline, ray casting offers a more flexible framework. An excellent and comprehensive overview of GPU-accelerated volume rendering can be found in the book of Engel et al. [16].

3. Fast box spline evaluation

3.1. BCC lattice

We define the BCC lattice as a sub-lattice of the CC lattice where only those points $(x,y,z) \in \mathbb{Z}^3$ belong to the lattice where x, y, and z are all even or all odd. These points can then be generated by integer linear combinations of the columns of the *sampling matrix*:

$$M_{BCC} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{bmatrix}. \tag{1}$$

Note that this definition of the BCC lattice has a density of 1/4 compared to the CC lattice. However, in order to have the same number of samples in the unit cube (i.e., to normalize the lattice) we scale the three axes of the BCC lattice by a factor of $1/\sqrt[3]{4}$ (i.e., we shrink it). A scaling of $1/\sqrt[3]{4}$ translates into a multiplication of the filter with 4. Thus, the BCC lattice can be seen as a CC lattice with an additional sample placed in each cube of the CC lattice (see Fig. 1), or as two interleaved CC lattices where the second CC lattice is shifted by one half of the grid spacing.

We define $p \in \mathbb{R}^3$ to be in *CC coordinates* which means that the axes of the frame are the canonical vectors $(1,0,0)^T$, $(0,1,0)^T$ and $(0,0,1)^T$. Furthermore, we define $p' = M_{BCC}^{-1} p$ as the corresponding point in *BCC coordinates*, where

$$M_{BCC}^{-1} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}; \tag{2}$$

i.e., p is expressed with the basis vectors of Eq. (1).

3.2. Box splines

In general, box splines are defined by a set of direction vectors. A box spline is then constructed by successive convolution of line segments along its direction vectors. For notation, the direction vectors are gathered in a matrix Ξ, specifically a box spline in $\mathbb{R}^d$ is characterized by $n \geq d$ vectors ξ_k $(1 \leq k \leq n)$ in $\mathbb{R}^d$, and $\Xi = [\xi_1 \dots \xi_n]$. The support of a box spline is defined as all points $x \in \mathbb{R}^d$ such that $x = \Xi t$ where $t \in \mathbb{R}^d$ and $0 \leq t_k \leq 1$ for $1 \leq k \leq n$. Thus, the support of the box spline is obtained by all convex combinations of the direction vectors.

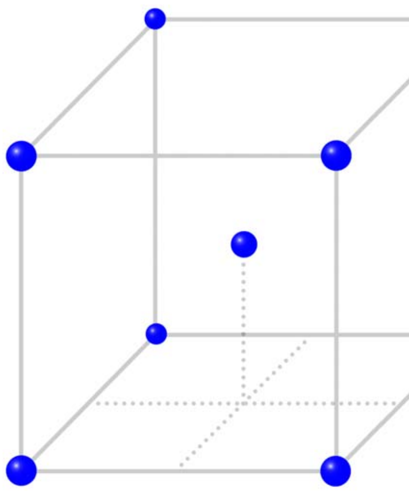

Fig. 1. The BCC lattice.

A first simple box spline can be obtained if $n=d$. In this case, it is the characteristic function of its support:

$$M_{\Xi}(x) = \begin{cases} \dfrac{1}{|\det \Xi|} & \text{where } x = \Xi t \text{ and } t \in [0,1)^n, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

In the case where $n > d$, the box splines are defined recursively:

$$M_{[\Xi,\xi]}(x) = \int_0^1 M_{\Xi}(x - t\xi)\,dt. \tag{4}$$

Thus, to construct higher order box splines, beginning with the base case in Eq. (3), the characteristic function is smeared along the additional direction vectors. We refer the interested reader to [3] for a more thorough introduction to box splines.

For the BCC lattice, we have d=3 and n=4. Therefore, we denote the four directions of the box spline [3] as

$$\Xi = [\xi_1, \xi_2, \xi_3, \xi_4] = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 \end{bmatrix}. \tag{5}$$

Note that $M_{BCC} = [\xi_1, \xi_2, \xi_3]$.

Entezari et al. [21] derived the polynomial pieces in *B-form* (Basis form) for the linear and quintic box splines leading to C^0 and C^2 reconstructions, respectively. However, they did not present an efficient evaluation of the resulting spline in piecewise polynomial form (i.e., *pp-form*) which is crucial for practical applications. The B-form describes a function as a weighted sum of splines; i.e., for every neighbor the spline gets evaluated independently and its contributions are summed up. The pp-form describes a function in terms of its local polynomial coefficients which is easier and faster to evaluate than a spline in B-form. Also, Entezari et al. [21] did not focus on an optimized implementation using the GPU, and therefore their method was not able to achieve interactive frame rates.

In this section, we show how to calculate the semi-discrete convolution in an efficient way using the GPU, aiming at interactive frame rates; i.e., we illustrate how to find and weight the neighbors that are needed for the convolution sum. A semi-discrete convolution is the convolution of a discrete lattice with a continuous filter.

The time-consuming part of the convolution is finding the contributions of the four (linear box spline) and 32 (quintic box spline) neighbors that are needed for evaluating the box spline at these positions. First, we describe how to find these neighbors; second, how to evaluate the box spline symbolically, sum the resulting polynomials into a single polynomial and evaluate that polynomial (the pp-form) at runtime *without* needing to do a full kernel evaluation for every point. We elaborate on this for the linear and the quintic box spline.

First, we show the explicit representation of the box spline [21] in pseudocode in Algorithm 1.

Algorithm 1. bsp(x,y,z)

```
1:   { Transform the point (x,y,z) to tetrahedron of focus }
2:   $x \leftarrow |x|$, $y \leftarrow |y|$, $z \leftarrow |z|$
3:   sort $x$, $y$, $z$ in decreasing order by swapping
4:   if linear box spline then
5:      if $x+y > 2$ then
6:         return 0
7:      end if
8:      return $(2-(x+y))/8$
9:   else if quintic box spline then
10:     if $(x+y) > 4$ then
11:        return 0
12:     end if
```

```
13:       if $(x+y)<2$ then
14:         return $R_1(x,y,z)$
15:       else if $(x+z)<2$ then
16:         return $R_2(x,y,z)$
17:       else if $(y+z)<2$ then
18:         if $(x-z)>2$ then
19:           return $R_{3A}(x,y,z)$
20:         else
21:           return $R_{3B}(x,y,z)$
22:         end if
23:       else
24:         return $R_4(x,y,z)$
25:       end if
26:     end if
```

The input to Algorithm 1 (bsp) are $x,y,z \in \mathbb{R}$. bsp makes use of the following box spline polynomials:

$$\begin{aligned}R_1(x,y,z) = {} & \mu(x+y-4)^3(-3xy-5z^2+2x+2y+20z+x^2+y^2-24)\\ & +\nu(x+z-2)^3(x^2-9x-3xz+10y-5y^2+14+11z+z^2)\\ & +\nu(y+z-2)^3(46-30x-z-y+3zy+5x^2-y^2-z^2)\\ & -\eta(x+y-2)^3(x^2+x-3xy-5z^2+y^2+y-6), \end{aligned} \tag{6}$$

$$\begin{aligned}R_2(x,y,z) = {} & \mu(x+y-4)^3(-3xy-5z^2+2x+2y+20z+x^2+y^2-24)\\ & -\nu(x+z-2)^3(-z^2-11z+3xz-14+5y^2+9x-10y-x^2)\\ & -\nu(y+z-2)^3(-46+z+30x+y-3zy-5x^2+y^2+z^2), \end{aligned} \tag{7}$$

$$R_{3A}(x,y,z) = \mu(x+y-4)^3(-x^2+8x+3xy-y^2+5z^2-16-12y), \tag{8}$$

$$\begin{aligned}R_{3B}(x,y,z) = {} & \mu(x+y-4)^3(-3xy-5z^2+2x+2y+20z+x^2+y^2-24)\\ & -\nu(y+z-2)^3(30x+z-46-3yz+y-5x^2+y^2+z^2), \end{aligned} \tag{9}$$

and

$$R_4(x,y,z) = \mu(x+y-4)^3(-3xy-5z^2+2x+2y+20z+x^2+y^2-24) \tag{10}$$

with constants $\mu=1/960$, $\nu=1/480$ and $\eta=1/240$. We point the interested reader to Entezari et al. [21] for the complete derivation of this box spline.

In order to reconstruct a spline f at an arbitrary point $p \in \mathbb{R}^3$ using its box spline representation, n neighbors $P_{1\ldots n} \in \mathbb{R}^3$ have to be found and each neighbor has to be weighted with bsp. The linear box spline's support is a rhombic dodecahedron that contains $n=4$ points. The support of the quintic box spline becomes a larger rhombic dodecahedron since its direction vectors have been multiplied by 2; i.e., $n=32$ neighbors have to be found and weighted [21]. Therefore, bsp is called n times and the results are summed up:

$$f(p) = \sum_{i=1}^{n} \mathrm{bsp}(P_i - p)D_i, \tag{11}$$

where we define D_i as the sample value on the BCC lattice at position P_i.

Eq. (11) shows the box spline in B-form. Obviously, using this representation results in n independent calls to bsp which is a very expensive operation on the GPU due to excessive branching.

We show in Sections 3.2.1 and 3.2.2 how Eq. (11) can be transformed to pp-form which will be faster to evaluate. In particular, the n sort operations in Line 3 in Algorithm 1 will be reduced to one.

3.2.1. Linear box spline

Let $p \in \mathbb{R}^3$ be the location where the underlying function f is interpolated. The support of the linear box spline is a rhombic dodecahedron. Four points $P_{1\ldots4} \in \mathbb{Z}^3$ fall into the support and form a tetrahedron [21]. Fig. 2(a) shows one possible configuration; i.e., the red point p is surrounded by the red rhombic dodecahedron but also lies inside the green tetrahedron. We indicate the vertices of the tetrahedron by $P_{1\ldots4} \in \mathbb{Z}^3$. There are six possible tetrahedra in the BCC lattice which are shown in Fig. 2(b). Since there are exactly six cases when sorting three numbers, the orientation of the tetrahedron is determined by the sort operation in Line 3 in Algorithm 1.

To determine the orientation of the tetrahedron we use the inverse of the matrix in Eq. (1) and transform p into BCC coordinates: $p' = M_{BCC}^{-1}p$. We are interested in the fractional part

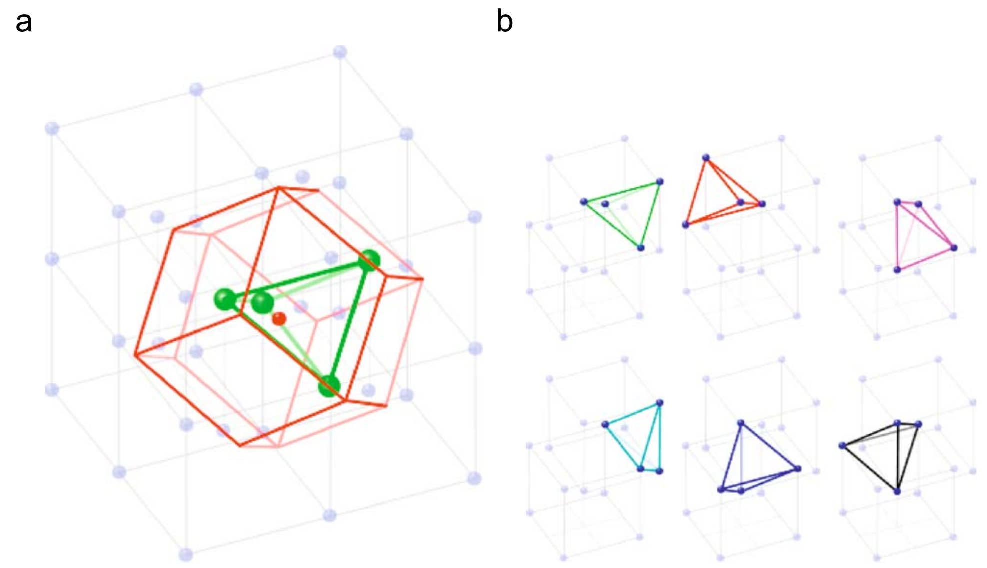

Fig. 2. (a) The rhombic dodecahedron (red) is the support of the linear box spline. Four points (green) fall into the support. These four points always form a tetrahedron. (b) There are six possible orientations for a tetrahedron in the BCC lattice. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of $p' : (\alpha,\beta,\gamma)^T = p' - \lfloor p' \rfloor \in [0,1]^3$; i.e., we center our coordinate system around $\lfloor p' \rfloor$ which is the first of the four neighbors: $P_1 = M_{BCC}\lfloor p' \rfloor$. The second neighbor is found immediately: $P_2 = P_1 - \xi_4 = P_1 + (1,1,1)^T$. The remaining two points $P_{3,4}$ are found by adding/subtracting vectors ξ_i to/from P_1/P_2, respectively.

Now $(\alpha,\beta,\gamma)^T$ are the coordinates of p in the BCC coordinate system centered around $\lfloor p' \rfloor$. The order of these three values is used to determine the tetrahedron that encloses p, together with its orientation.

Starting at point P_1 we can determine P_3 by adding a box spline vector ξ_i to P_1 that is determined by $\max\{\alpha,\beta,\gamma\}$:

$$P_3 = P_1 + \begin{cases} (1,1,-1)^T = \xi_1 & \text{if } \max\{\alpha,\ \beta,\ \gamma\} = \alpha, \\ (1,-1,1)^T = \xi_2 & \text{if } \max\{\alpha,\ \beta,\ \gamma\} = \beta, \\ (-1,1,1)^T = \xi_3 & \text{if } \max\{\alpha,\ \beta,\ \gamma\} = \gamma. \end{cases}$$

Finally, we determine P_4 by subtracting another box spline vector ξ_j from P_2 that is determined by $\min\{\alpha,\beta,\gamma\}$:

$$P_4 = P_2 - \begin{cases} (1,1,-1)^T = \xi_1 & \text{if } \min\{\alpha,\ \beta,\ \gamma\} = \alpha, \\ (1,-1,1)^T = \xi_2 & \text{if } \min\{\alpha,\ \beta,\ \gamma\} = \beta, \\ (-1,1,1)^T = \xi_3 & \text{if } \min\{\alpha,\ \beta,\ \gamma\} = \gamma. \end{cases}$$

An example is presented in Fig. 3: p is the green point and the tetrahedron $P_{1\ldots4}$ is shown in red. The black axes X, Y, and Z denote the canonical axes and the blue axes $\xi_{1\ldots3}$ are the axes of the BCC lattice as defined in Eq. (1). Let $P_1=(0{,}0{,}0)^T$, $P_2=(1{,}1{,}1)^T$ and $p=(1,\ 1/2,\ 1/5)^T$ (grey lines) and therefore $(\alpha,\beta,\gamma)^T = (3/4, 3/5, 7/20)^T$ (dotted lines). We have $\alpha > \beta > \gamma$ and therefore P_3 is determined by adding ξ_1 to P_1: $P_3 = P_1 + \xi_1 = (1,1,-1)^T$. P_4 is calculated by subtracting ξ_3 from P_2: $P_4 = P_2 - \xi_3 = (2,0,0)^T$. In general, according to the order of α, β, and γ different vectors ξ_i are picked to be added and subtracted.

By determining the tetrahedron in this way the sort operation in Line 3 in Algorithm 1 has only to be performed once in advance and we can transform Eq. (11) into

$$f(p) = D_1 + a(D_3 - D_1) + b(D_4 - D_3) + c(D_2 - D_4) \tag{12}$$

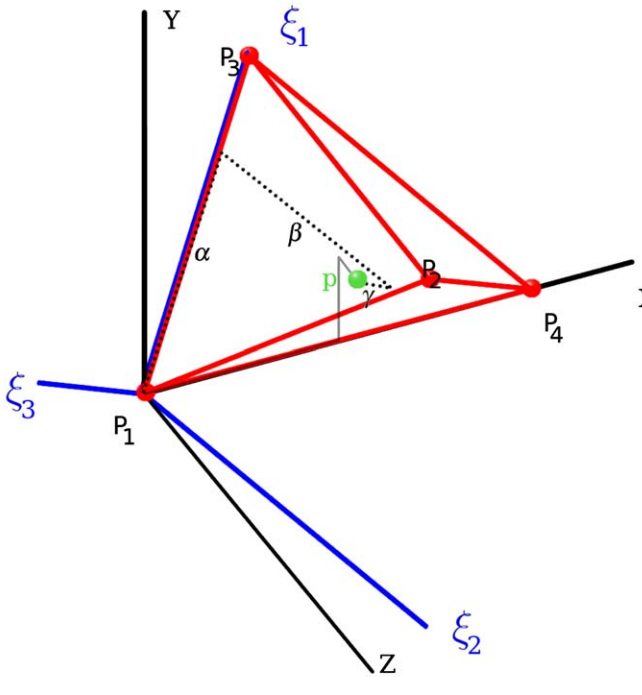

Fig. 3. The point p (green) is inside the tetrahedron $P_{1\ldots4}$. P_3 and P_4 are determined according to the order of (α,β,γ) (dotted lines). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

(where $a = \max\{\alpha,\beta,\gamma\}$, $b = \text{mid}\{\alpha,\beta,\gamma\}$ and $c = \min\{\alpha,\beta,\gamma\}$) which is faster than evaluating the box spline four times. Eq. (12) is in pp-form where we only need to perform the sort operation once. Eq. (12) is also equivalent to barycentric interpolation of the tetrahedron's points.

To see this, we assume without loss of generality that $p = (x,y,z)^T \in [0,1)^3$ and $x > y > z$ (the other five cases work analogously). Therefore, $p' = (\alpha,\beta,\gamma)^T = M_{BCC}^{-1}(x,y,z)^T = \frac{1}{2}(x+y,\ x+z, y+z)^T$, and thus $\alpha > \beta > \gamma$. Consequently, it follows that $P_1=(0{,}0{,}0)^T$, $P_2=(1{,}1{,}1)^T$, $P_3=(1{,}1{,}-1)^T$, and $P_4=(2{,}0{,}0)^T$. Plugging this into Eq. (11), we obtain

$$\begin{aligned} f(p) = f((x,y,z)^T) &= \sum_{i=1}^{4} \text{bsp}(P_i - p)D_i \\ &= \text{bsp}(P_1-(x,y,z)^T)D_1 + \text{bsp}(P_2-(x,y,z)^T)D_2 + \text{bsp}(P_3-(x,y,z)^T)D_3 \\ &\quad + \text{bsp}(P_4-(x,y,z)^T)D_4 = \left(1 - \frac{1}{2}(x+y)\right)D_1 + \frac{1}{2}(y+z)D_2 \\ &\quad + \frac{1}{2}(y-z)D_3 + \frac{1}{2}(x-y)D_4 = D_1 + \frac{1}{2}(x+y)(D_3-D_1) \\ &\quad + \frac{1}{2}(x+z)(D_4-D_3) + \frac{1}{2}(y+z)(D_2-D_4) = D_1 + \alpha(D_3-D_1) \\ &\quad + \beta(D_4-D_3) + \gamma(D_2-D_4), \end{aligned} \tag{13}$$

which is equivalent to Eq. (12).

3.2.2. *Quintic box spline*

The linear box spline is an interpolating reconstruction kernel that guarantees C^0 continuity. For C^2 continuity, the quintic box spline is employed. As in the linear case, we want to avoid calling `bsp` for every point because it would result in expensive operations. We show how the neighborhood for the quintic box spline can be found and weighted in an efficient way by transforming Eq. (11) into pp-form which will reduce the number of sort operations (Line 3 in Algorithm 1). As we will see in Section 5, this enables us to use the high-quality quintic box spline for interactive volume rendering.

The support of the quintic box spline is a rhombic dodecahedron where the box spline's direction vectors ξ_i are multiplied by 2. In total, 32 points of the BCC lattice fall into this rhombic dodecahedron. Furthermore, the rhombic dodecahedron can be split into four parallelepipeds [21], each containing eight points (Fig. 4).

The first task is to find these 32 neighbors. This can be done by examining the four parallelepipeds $P_{i_j}, i = 1\ldots4, j = 1\ldots8$, separately. First, we determine the first four neighbors P_{i_1} which are exactly the same points as in Section 3.2.1 (linear box spline); i.e., P_{i_1} form a tetrahedron. These four points build the *anchors* of each of the four parallelepipeds. Starting at these four points, we can determine the diagonal q_i of every parallelepiped, which is one of the four directions of the box spline ξ_i. Fig. 5 shows the symbolic rhombic dodecahedron (red) indicated in Fig. 4 and one of the four parallelepipeds (blue). The anchor point of the blue parallelepiped is P_{1_1} and the diagonal is $P_{1_8} - P_{1_1} = (-1,-1,-1)^T = \xi_4$.

To determine the four diagonals q_i we use again the order of $(\alpha,\beta,\gamma)^T$, which is computed as in Section 3.2.1. Starting at the four anchor points we obtain q_i $(i = 1\ldots4)$ as follows: $q_1 = (-1,-1,-1)^T = \xi_4$, $q_3 = P_{3_1} - P_{1_1}$. The remaining diagonals are

$$q_2 = \begin{cases} (1,1,-1)^T = \xi_1 & \text{if } \min\{\alpha,\ \beta,\ \gamma\} = \alpha, \\ (1,-1,1)^T = \xi_2 & \text{if } \min\{\alpha,\ \beta,\ \gamma\} = \beta, \\ (-1,1,1)^T = \xi_3 & \text{if } \min\{\alpha,\ \beta,\ \gamma\} = \gamma, \end{cases}$$

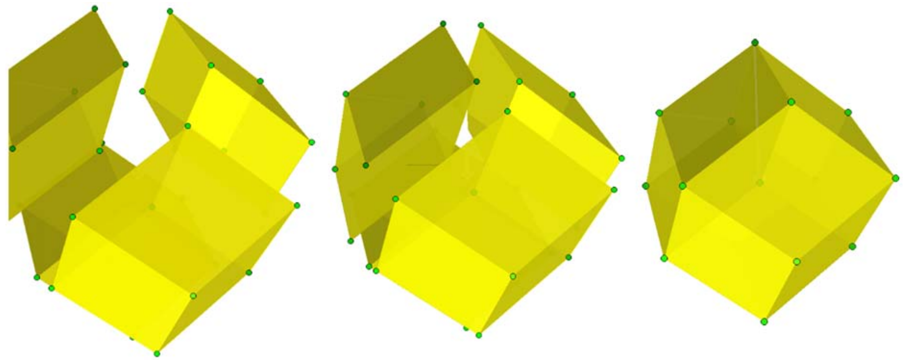

Fig. 4. The samples falling into the support of the quintic box spline form four parallelepipeds. By symbolically moving them together we get a rhombic dodecahedron and we can use the direction vectors of the box spline to determine all 32 points.

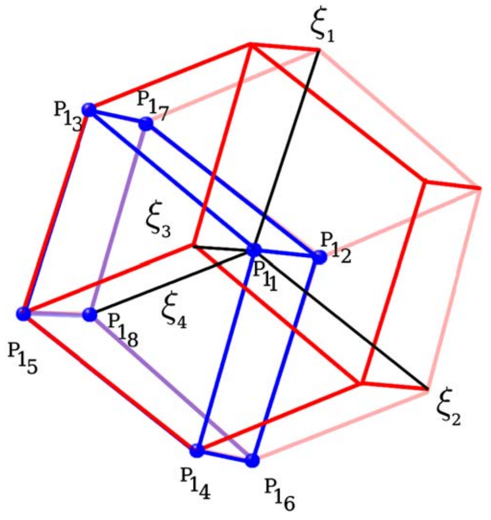

Fig. 5. The rhombic dodecahedron can be split into four parallelepipeds where for simplicity of illustration only one (blue) is displayed. Using the vectors of Ξ we can determine all eight points in each parallelepiped. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

$$q_4 = \begin{cases} (1,1,-1)^T = \xi_1 & \text{if } \operatorname{mid}\{\alpha, \ \beta, \ \gamma\} = \alpha, \\ (1,-1,1)^T = \xi_2 & \text{if } \operatorname{mid}\{\alpha, \ \beta, \ \gamma\} = \beta, \\ (-1,1,1)^T = \xi_3 & \text{if } \operatorname{mid}\{\alpha, \ \beta, \ \gamma\} = \gamma. \end{cases}$$

Once the diagonals are determined we have $P_{i_8} = P_{i_1} + q_i$ and the remaining six points of each parallelepiped are fixed. For every parallelepiped i the points P_{i_j}, $i = 1\ldots4$ are given by

$$P_{i_2} = P_{i_8} - (2q_{i_x},0,0)^T, \quad P_{i_5} = P_{i_1} + (2q_{i_x},0,0)^T,$$

$$P_{i_3} = P_{i_8} - (0,2q_{i_y},0)^T, \quad P_{i_6} = P_{i_1} + (0,2q_{i_y},0)^T,$$

$$P_{i_4} = P_{i_8} - (0,0,2q_{i_z})^T, \quad P_{i_7} = P_{i_1} + (0,0,2q_{i_z})^T.$$

P_{i_2} through P_{i_7} are the other six points that (together with P_{i_1} and P_{i_8}) span the parallelepiped.

The last step is to weigh each of the 32 neighbors. Therefore, we exploit the symmetries between the four parallelepipeds of the rhombic dodecahedron. Lines 2 and 3 in Algorithm 1 can be interpreted as a transformation of p to the first parallelepiped [21]. The convolution can be computed in one parallelepiped and the remaining three can be obtained by a symmetry transform into the first one [21].

To that aim, we need to keep the consistency (i.e., the order) of the points in a parallelepiped when transforming between parallelepipeds. This is necessary in order to map the points of the other three parallelepipeds to points of the first parallelepiped that is being evaluated by the exact same polynomial of `bsp`.

Preserving the correct order of the points translates into two permutations π_1 and π_2 of the six points $P_{i_{j_1}}$ $(j_1 = 2\ldots4)$ and $P_{i_{j_2}}$ $(j_2 = 5\ldots7)$: π_1 and π_2 depend on the order of α, β, and γ as these values determine the order of the axes that have to be picked to get the points of the parallelepipeds:

$$\pi_1(k) = \begin{cases} (2,3,4), \\ (3,2,4), \\ (4,2,3), \\ (2,4,3), \\ (3,4,2), \\ (4,3,2), \end{cases} \quad \pi_2(k) = \begin{cases} (7,6,5) & \text{if } \alpha > \beta > \gamma, \\ (6,7,5) & \text{if } \alpha > \gamma > \beta, \\ (5,7,6) & \text{if } \gamma > \alpha > \beta, \\ (7,5,6) & \text{if } \beta > \alpha > \gamma, \\ (6,5,7) & \text{if } \beta > \gamma > \alpha, \\ (5,6,7) & \text{if } \gamma > \beta > \alpha, \end{cases}$$

where $k \in \{1,2,3\}$ denotes which value of π is chosen. The if-statements apply to π_1 and π_2. Having obtained all 32 neighbors in this way, we build the convolution sum in every parallelepiped i (where D_{i_j} are again the values at points P_{i_j}):

$$f_i(p) = D_{i_1}\operatorname{bsp}(P_{i_1} - p) + D_{i_8}\operatorname{bsp}(P_{i_8} - p) + \sum_{k=1}^{3} (D_{i_{\pi_1(k)}}\operatorname{bsp}(P_{i_{\pi_1(k)}} - p) + D_{i_{\pi_2(k)}}\operatorname{bsp}(P_{i_{\pi_2(k)}} - p)). \tag{14}$$

We can now compute the convolution sum $f_i(p)$ for one parallelepiped. By employing the symmetries of the rhombic dodecahedron, we just have to transform all other three parallelepipeds into the first one and we can compute the convolution sum for all 32 points. The total result is

$$f(p) = \sum_{i=1}^{4} f_i(p). \tag{15}$$

Examining the 32 calls to `bsp` in Eq. (15), one can see that per parallelepiped i and for every permutation π_1 and π_2 the same eight branches are taken; i.e., by sorting α, β, and γ once, we know in advance which polynomials of `bsp` have to be evaluated. For each parallelepiped the data points D_{i_j} get weighted with the following polynomials (in this order):

$D_{i_1} \to R_1,$

$D_{i_8} \to R_4,$

$D_{i_{\pi_1(0)}} \to R_2,$

$D_{i_{\pi_1(1)}} \to R_{3A},$

$D_{i_{\pi_1(2)}} \to R_{3B},$

$D_{i_{\pi_2(0)}} \rightarrow R_{3A}$,

$D_{i_{\pi_2(1)}} \rightarrow R_4$,

$D_{i_{\pi_2(2)}} \rightarrow R_{3B}$.

Thus, no branching is needed except for the sorting of α, β, and γ and Eq. (15) can be expanded and is then in pp-form. This yields an efficient evaluation of the quintic box spline.

In the Appendix, we make the GLSL functions for the linear and quintic box spline available to the community.

4. Implementation

4.1. Volume ray casting

Having set up the linear and quintic box spline as filter kernels for the BCC lattice, we employ them in a high-quality ray caster; i.e., we use ray marching to sample a volume along rays cast from the camera into the volume. At each sample position, a filter kernel is used to reconstruct the volume from discrete samples. Then, all samples are composed resulting in an intensity value of a pixel.

We compare the linear and quintic box spline on the BCC lattice with the traditional trilinear and tricubic B-spline on the CC lattice as well as with the prefiltered trilinear and tricubic B-spline for BCC lattices [14]. We shall call the first comparison method *CC B-splines* and the second one *BCC B-splines*.

The skeleton of the ray caster implementation is the same for every method, only the fragment shader with the filter kernels and the storage schemes of the volumes have to be adjusted for all three methods.

A BCC lattice can be represented as two interleaved CC lattices. Whereas Csébfalvi et al. [14] store the BCC lattice as two separate CC textures, our method stores the BCC lattice in an interleaved manner: The samples of the second lattice are shifted by half a grid spacing in every dimension. Thus, in every cube spanned by eight samples of the first CC lattice an additional sample is placed in the center of this cube. By storing the BCC lattice in a 3D array and using the following mapping, a fast conversion of a BCC point (x,y,z) to its index in the 3D array (i,j,k) is achieved by $i = x \div 2$, $j = y \div 2$, and $k = z$ where $\div$ is an integer division. Note that x, y, and z are either all even or all odd in a BCC lattice. Fig. 6 illustrates this principle.

Being able to store the two interleaved CC lattices as one 3D array, it is loaded as a 3D texture into GPU memory.

4.2. Prefiltering

Generalized interpolation [2] can be used to improve the reconstruction quality of a non-interpolating filter kernel. In a discrete prefiltering step, a non-interpolating filter is used to preprocess the samples of the volume yielding a new volume of coefficients. Now using the same filter for reconstruction in combination with the coefficients, the filter becomes interpolating improving its reconstruction quality.

The fundamental concepts can be best explained by the 1D case. Let us consider the samples f_k, $k \in \mathbb{Z}$, taken on a grid with spacing T, and the reconstruction kernel $h(x)$, $x \in \mathbb{R}$. We can express the reconstruction $\tilde{f}$ as a discrete/continuous convolution

$$f(x) \approx \tilde{f}(x) = \sum_k c_k h(x/T - k), \tag{16}$$

where c_k are the coefficients to be determined by the prefiltering step [2]. Imposing the interpolation condition leads to the constraint

$$\tilde{f}(iT) = \sum_k c_k h(i-k) = c \otimes g = f_i, \tag{17}$$

where $g = h(l)$ is the sampled kernel. Consequently, we have $c = f \otimes g^{-1}$ and the coefficients can be obtained by discrete convolution of the inverse filter g^{-1}. Notice that the prefiltering step becomes trivial when the reconstruction kernel h is interpolating; i.e., from $h(k) = \delta_k$ follows readily $c_k = f_k$.

Implementing the inverse-filtering step can be done by a recursive algorithm in the spatial domain for 1D B-splines [2], or in general in the Fourier domain assuming periodic boundary conditions. The discrete Fourier transform (DFT) of the samples f_k, $k = 0, \ldots, N-1$, is defined as

$$F(e^{j\omega}) = \sum_{l=0}^{N-1} f_l e^{j\omega l}, \tag{18}$$

where the discrete Fourier coefficients are obtained for $\omega = 2\pi k/N$, $k = 0, \ldots, N-1$. By the convolution theorem, one can perform inverse filtering in the Fourier domain as

$$c_k \longleftrightarrow C(e^{j\omega}) = \frac{F(e^{j\omega})}{G(e^{j\omega})}, \tag{19}$$

where G is the DFT of g. The Fourier domain algorithm is easy to apply to the 3D case and for non-CC lattices if the associated DFT is at hand.

Discrete prefiltering can be employed if the filter kernel is admissible in the sense that the discrete Fourier transform (DFT) of the sampled kernel is non-zero [2]. In 1D, all popular kernels

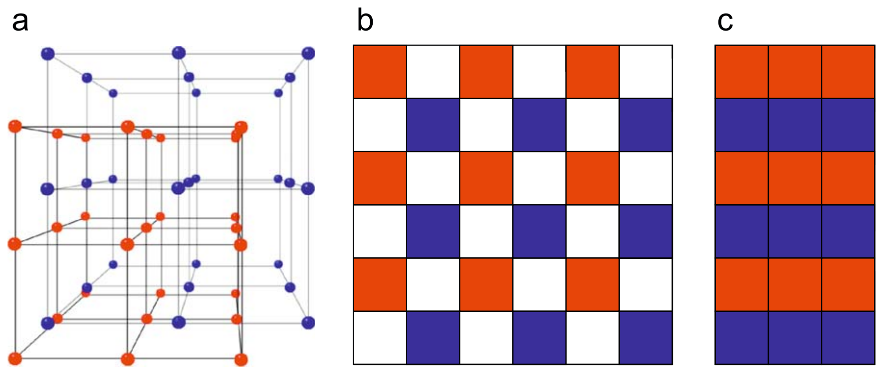

Fig. 6. (a) A BCC lattice is built from two CC lattices (red and blue). (b) 2D scheme of the BCC lattice. Every odd (blue) row (i.e., slice in 3D) is shifted to the center of the "even" (red) CC lattice. (c) shows the corresponding memory layout. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

including causal B-splines and odd-degree symmetric B-splines are admissible and guarantee a stable prefilter. Consequently, their separable extension can be used without any problem for 3D [12]. On the BCC lattice, the quintic box spline is also shown to be an appropriate choice [21]. It should be noted that higher-order B-splines and box splines are smoothing and non-interpolating, which makes prefiltering an essential step to exploit their improved approximation quality.

First, we implemented the discrete Fourier transform for BCC lattices using Alim et al.'s method [1] which makes use of the multi-dimensional FFT algorithm. Therefore, using the FFTW package [24], a DFT on the BCC lattice can be implemented efficiently.

Prefiltered BCC B-splines are theoretically not well adapted since the B-splines are not designed for BCC lattices. As a consequence, during prefiltering division by zero can occur, which can be avoided by "lifting" the denominator of the Fourier expression of the inverse filter by a small positive number ε. However, the obtained filter kernels are not interpolating any more. Therefore, Csébfalvi et al. call their method "practically interpolating" [14]. The method offers reconstruction kernels for the BCC lattice that can be used on the GPU and, to our knowledge, delivers the best compromise between rendering speed and image quality for BCC lattices so far. Csébfalvi et al. [14] suggest to set ε as a percentage of the signal's DC component. In accordance with their work, we therefore set ε to 1% of the signal's DC component to render the images in Section 5. Increasing ε would increase the smoothing property of the filter and decrease the "practically interpolating" property.

Attention has to be paid to the prefiltering step; i.e., after prefiltering the coefficients can be negative. This has to be taken into account when storing the volume as textures; e.g., by using 32-bit textures.

5. Results

We tested our method on a PC with a Nvidia GeForce 8800 GTX graphics card using OpenGL and GLSL. We compare the linear and quintic box spline (with and without prefiltering) to the prefiltered trilinear and tricubic BCC B-spline [14] and the trilinear and tricubic CC B-spline (with and without prefiltering).

5.1. Rendering speed

We measured the frames per second for all three methods. Note that discrete prefiltering is a preprocessing step performed once and has no influence on these results.

Table 1 indicates the number of texture lookups every reconstruction kernel needs. GPUs usually support two types of 3D texture lookups in CC lattices: A lookup that fetches the nearest neighbor and a lookup that automatically performs a trilinear interpolation of 8 texels. For the remainder of the article we shall call the first type *NN lookups* and the second type *LIN lookups*. Three-dimensional textures for GPUs are stored as CC lattices. Further, GPUs are especially designed to perform LIN lookups as fast as a NN lookup (although accessing eight instead of one texel). LIN lookups cannot directly be used for box splines because BCC lattices use a different storage scheme (see Fig. 6) and since box splines are not tailored to the CC lattice. Therefore, our method has to use NN lookups making a hardware-accelerated box spline implementation slower. In contrast, a B-spline on a CC lattice (and therefore also BCC B-splines) can be computed using LIN lookups reducing the number of texture lookups by a factor of 8.

We show the numbers of NN and LIN lookups in Table 1 since the number of lookups has considerable impact on the performance of each kernel. As the reader may verify, we also show the number of NN lookups for linear interpolation on the CC lattice which is not always necessary since special purpose units for linear interpolation are available (i.e., such a special purpose unit performs a LIN lookup).

In general, one should use LIN lookups if possible to perform trilinear interpolation. However, under some circumstances, one has to switch back to NN lookups when using CC lattices to compute trilinear interpolation manually. For high-quality renderings 32-bit textures have to be employed and only graphics cards of the latest generations support LIN lookups on the 128-bit pipeline. If working with older graphics cards (e.g., Nvidia GeForce 6800) hardware-accelerated LIN lookups are not available and therefore NN lookups are mandatory.

As mentioned before, LIN lookups on the GPU are only available for the CC but not for the BCC lattice. This disadvantage could be mitigated by the evolution of future graphics hardware such as the Larrabee architecture [35]. Although the Larrabee architecture still has a hardware CC texture unit, it is likely that the relative performance of the box splines will be improved due to Larrabee's more flexible instruction set.

Therefore, we also compare the box splines (that so far can only work with NN lookups) to a manual implementation of linear interpolation using NN lookups.

Note that the CC B-splines and also the BCC B-splines can make use of LIN lookups and therefore are easy to use and allow for a fast implementation.

We used three different datasets of different sizes which we rendered at a resolution of 512×512 pixels: The Marschner–Lobb [27] dataset sampled on a CC lattice with a size of $40 \times 40 \times 40 = 64\text{k}$ and on a BCC lattice with $28 \times 28 \times 56 \approx 44\text{k}$ samples, the carp dataset (CC: $180 \times 180 \times 180 = 5832\text{k}$, BCC: $129 \times 129 \times 258 \approx 4293\text{k}$), and the mouse embryo dataset (CC: $449 \times 663 \times 449 \approx 133,661\text{k}$, BCC: $321 \times 474 \times 642 \approx 97,683\text{k}$). The BCC datasets have approximately 70% of the samples of the CC datasets which is justified due to the optimality of the BCC lattice.

The Marschner–Lobb test function is analytically defined and can therefore be sampled very easily on an arbitrary lattice. The carp dataset is a CT scan originally stored on a fairly densely sampled CC lattice. We constructed comparable BCC and CC datasets by merely subsampling from the densely sampled CC dataset. The mouse embryo dataset was acquired from 400 projections via OPT (optical projection tomography) [36]. We extended the standard implementation of the expectation maximization algorithm so it reconstructs the volume on CC and BCC lattices [23].

We created volume renderings using on-the-fly gradient computation as this delivers most accurate results (i.e., we used central differencing with a relatively small step size). Images of the Marschner–Lobb dataset are found in Fig. 8, the carp dataset is found in Fig. 10, and the mouse embryo dataset is found in Fig. 9.

Table 1
Samples needed for every reconstruction kernel.

Method	Box spline	BCC B-spline	CC B-spline
C^0 NN	4 NN	2×8 NN	8 NN
C^0 LIN	n.a.	2×1 LIN	1 LIN
C^2 NN	32 NN	2×64 NN	64 NN
C^2 LIN	n.a.	2×8 LIN	8 LIN

C^0 denotes the linear kernels, C^2 denotes the quintic and tricubic kernels. NN and LIN denote the implementation of each kernel with nearest neighbor texture lookups and trilinear texture lookups, respectively. The CC B-spline version of "C^2 LIN" is the tricubic B-spline according to Sigg and Hadwiger's implementation [37].

5.1.1. *Box splines: B-form and pp-form*

Table 2 shows the difference between the box splines in B-form and pp-form. The pp-form of the linear box spline (lines 1 and 2) is four times faster than the B-form. For the quintic box spline, the pp-form is crucial, since the B-form achieves at most 0.02 fps, whereas the pp-form is able to achieve almost interactive frame rates. This shows the necessity of transforming the box spline from B-form to pp-form. Our fast evaluation scheme guarantees the desired speedup to make box splines attractive for interactive volume rendering.

5.1.2. *Performance comparisons for box splines, CC B-splines, and BCC B-splines*

Table 3 compares the timings of the box spline pp-form with the BCC B-splines and the CC B-splines.

First, we compare the box splines to the BCC B-splines and CC B-splines when LIN lookups are used for both B-spline methods: the box splines, of course, do not perform as well as the two other methods. However, frame rates are more similar as soon as the dataset size increases (compare the figures in lines 1, 4, and 8 in the last column of Table 3, and lines 2, 6, and 10) since memory access becomes more expensive: The box splines achieve the following frame rates: 5.62 fps (linear box spline) compared to 7.32 (trilinear BCC B-spline) and 11.06 fps (trilinear CC B-spline), and 0.47 fps (quintic box spline) compared to 0.80 (tricubic BCC B-spline) and 2.64 fps (tricubic CC B-spline).

When only NN lookups are employed, the linear box spline is approximately three times faster than the prefiltered trilinear BCC B-spline and slightly faster than the trilinear CC B-spline. Furthermore, the quintic box spline is faster than the prefiltered tricubic BCC B-spline for all three datasets but still slower than the tricubic CC B-spline. This indicates that box splines could compete with the comparison methods in terms of rendering speed with the emergence of new graphics hardware in the future. We will see in the next section that quintic box splines deliver superior reconstruction accuracy and better visual quality.

Note that the advantage of using LIN lookups is due to the fact that trilinear interpolation on the CC lattice is available through special purpose units on the GPU. These lookups are almost as fast as NN lookups (although accessing 8 times as much data). This advantage is likely to be mitigated as soon as more evolved architectures are available, and our comparisons show that the box splines will then be able to catch up with the comparison methods. Furthermore, note that one has to switch back to NN lookups if one wants to use 32-bit textures on older graphics hardware and in this case box splines become more attractive as well.

Table 2
Frames per second (fps) for the linear and quintic box spline using the B-form and our fast evaluation scheme (pp-form).

	Method	ML	Carp	Mouse
1	*Linear box spline (B-form) (NN)*	3.57	3.10	1.58
2	*Linear box spline (pp-form) (NN)*	14.58	12.47	5.62
3	*Quintic box spline (B-form) (NN)*	0.02	0.02	0.01
4	*Quintic box spline (pp-form) (NN)*	1.03	1.01	0.47

Three datasets were used: The Marschner–Lobb (ML), the carp, and the mouse embryo dataset.

Table 3
Frames per second (fps) for the linear and quintic box spline using our fast evaluation scheme (pp-form) and prefiltered B-spline reconstruction for BCC and CC lattices.

	Method	ML	Carp	Mouse
1	*Linear box spline (pp-form) (NN)*	14.58	12.47	5.62
2	*Quintic box spline (pp-form) (NN)*	1.03	1.01	0.47
3	*Trilinear BCC B-spline (NN)*	4.94	4.65	2.11
4	*Trilinear BCC B-spline (LIN)*	26.02	20.54	7.32
5	*Tricubic BCC B-spline (NN)*	0.74	0.69	0.32
6	*Tricubic BCC B-spline (LIN)*	1.81	1.67	0.80
7	*Trilinear CC B-spline (NN)*	13.49	12.04	5.10
8	*Trilinear CC B-spline (LIN)*	39.68	32.63	11.06
9	*Tricubic CC B-spline (NN)*	1.49	1.37	0.63
10	*Tricubic CC B-spline (LIN)*	6.80	6.04	2.64

Three datasets were used: The Marschner–Lobb (ML), the carp, and the mouse embryo dataset.

5.2. *Image quality*

To compare the reconstruction quality of each filter kernel, we employed the Marschner–Lobb test function sampled on lattices as in Section 5.1 and produced volume renderings.

Since the linear B-spline and the linear box spline are already interpolating on the CC and BCC lattice, discrete prefiltering results in the identity; i.e., the coefficients obtained via prefiltering will be the same as the original samples. Fig. 7 shows results for the linear filters (C^0 reconstruction). The first row shows volume rendered images of the test function and the second row shows the corresponding error images: They visualize the angular error when estimating the surface normals using central differences with a relatively small step size from the reconstructed test function. An angular error of more than 30° is mapped to white. Black denotes an angular error of zero. All three filters have difficulties to reconstruct the signal correctly and the linear box spline shows more visually noticeable artifacts due to the tetrahedral structure of the kernel. However, the error image of the linear box spline in the second row is slightly darker than the other two images and has therefore a smaller error. Furthermore, the error is distributed more regularly.

Fig. 8 shows analog results for C^2 reconstruction. Both, the tricubic CC B-spline and the quintic box spline (a–d) show improvements when prefiltering is used: The valleys are deeper and the filters are not as smoothing as when used without prefiltering. The error images in the second row confirm that prefiltering substantially improves reconstruction accuracy and image quality. However, the quintic box spline delivers better image quality and error behaviour in both cases. When comparing all three error images where prefiltering is used (second row, (b), (d) and (e)) it is apparent that the prefiltered quintic box spline is able to reconstruct the test function more accurately than the prefiltered tricubic B-spline for CC (b) and BCC (e) lattices, which makes it the best choice.

Although the tricubic BCC B-spline (top e) reconstructs the circular shapes of the outer rings better, the angular error is worse and the experiments with real data sets (see below) show that the quintic box spline is superior.

To show the visual difference of prefiltering for a real dataset, we rendered close-ups of a smaller version of the mouse embryo dataset with box splines. The dataset has $87 \times 192 \times 174 \approx 1953$k BCC samples and was obtained the same way as described in Section 5.1. Fig. 9 shows the mouse embryo (a) and three close-ups (b–d) where (b) was rendered using the linear box spline, (c) was rendered using the quintic box spline without prefiltering, and (d) shows the mouse embryo head using the quintic box spline with prefiltering. The linear box spline is

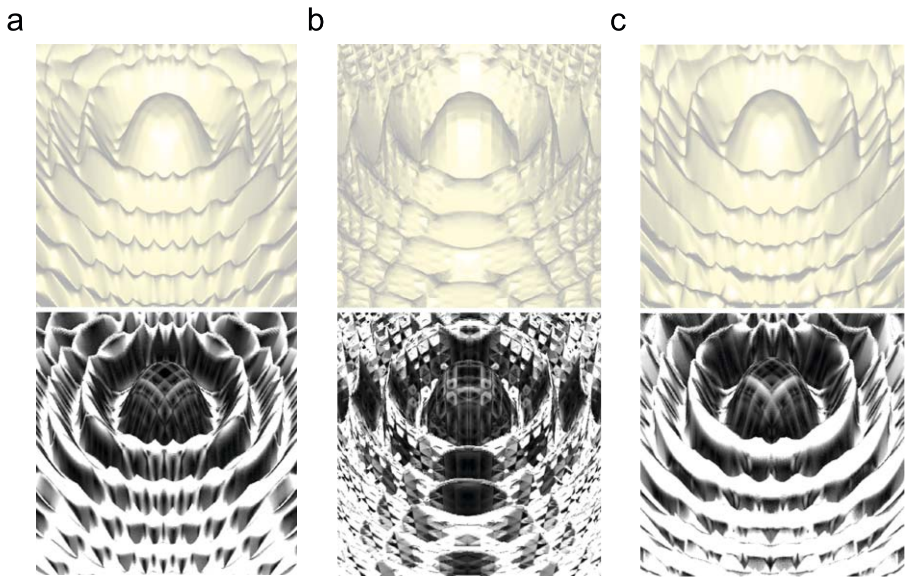

Fig. 7. First row: Marschner–Lobb test function. Second row: Corresponding error images. An angular error of the reconstructed surface normal of more than 30° is mapped to white. Black denotes an error of zero. (a) Trilinear B-spline reconstruction from $40\times 40\times 40$ CC samples. (b) and (c) Linear box spline and refiltered trilinear B-spline reconstruction from $28\times 28\times 56$ BCC samples.

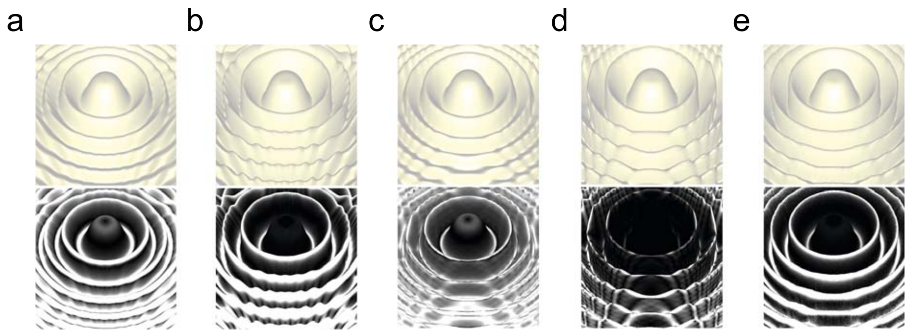

Fig. 8. First row: Marschner–Lobb test function. Second row: Corresponding error images analog to Fig. 7. (a) and (b) Tricubic B-spline reconstruction from $40\times 40\times 40$ CC samples without and with prefiltering. (c) and (d) Quintic box spline reconstruction from $28\times 28\times 56$ BCC samples without and with prefiltering. (e) Prefiltered tricubic B-spline reconstruction from $28\times 28\times 56$ BCC samples.

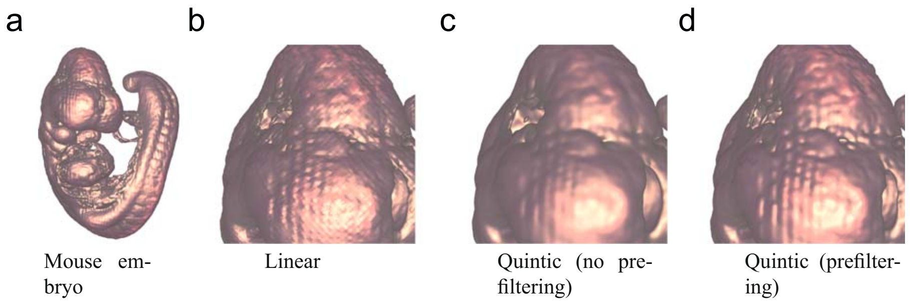

Fig. 9. Close-up of the mouse embryo with $87\times 192\times 174$ BCC samples rendered with the linear and quintic box spline.

able to reconstruct details but girdering artifacts [6] are visible. The quintic box spline shows its smoothing property when no prefiltering is employed and details are lost. The best visual result is obtained with the quintic box spline and prefiltering: The girdering artifacts vanish and small details are preserved.

Last, we compare the prefiltered filter kernels for C^2 reconstruction on a real dataset. Fig. 10 shows the carp dataset with the prefiltered quintic box spline (b), the prefiltered tricubic BCC B-spline (c), and the prefiltered tricubic CC B-spline (d). The datasets have approximately the same number of samples: The BCC dataset has $129\times 129\times 258\approx 4293$k samples and the CC dataset has $164\times 164\times 164\approx 4411$k samples. The prefiltered tricubic BCC B-spline (c) does not preserve as many details as the other two methods. The prefiltered quintic box spline (b) is able to reconstruct the carp's ribs more accurately and shows better image quality than the prefiltered tricubic CC B-spline (d).

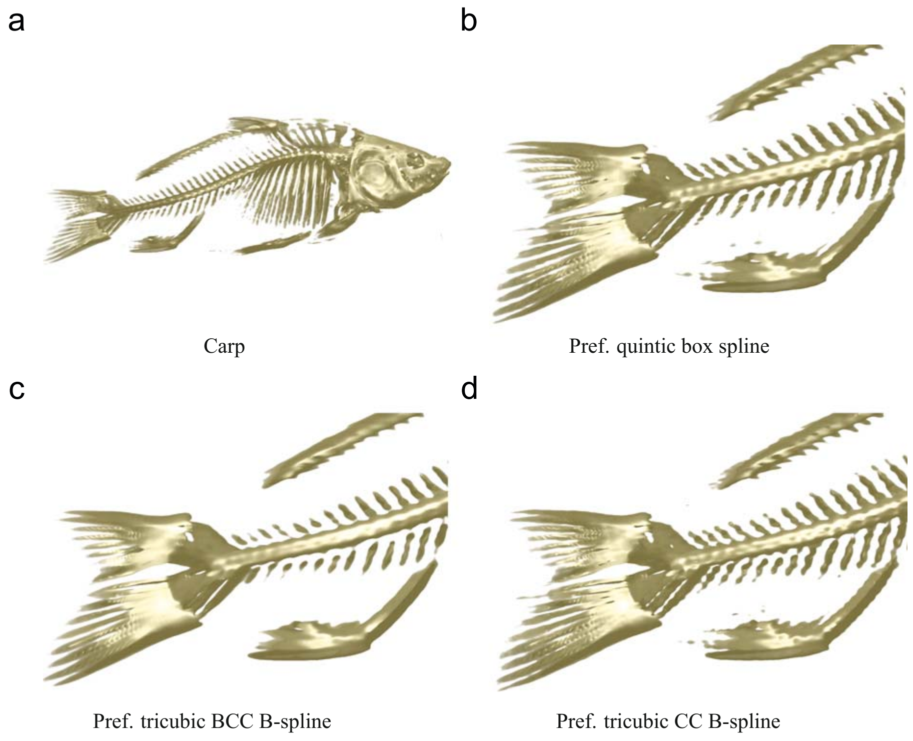

Fig. 10. The carp dataset sampled on $\approx$4293k BCC samples (b, c), and on $\approx$4411k CC samples (d). (b–d) compare the prefiltered quintic box spline (b), the prefiltered tricubic BCC B-spline (c) and the prefiltered tricubic CC B-spline (d).

5.3. Numerical comparisons

In the context of volume rendering, a visual comparison between different reconstruction filters as seen in Section 5.2 is the most important quality measure. Besides that, a numerical comparison of the filters is of interest as well. To that aim, we compute two error measures, the L_1 and L_2 error, for the analytically defined ML dataset.

Let $D=[-1,1]^3 \subset \mathbb{R}^3, x \in D, f: D \rightarrow \mathbb{R}$ the scalar ML test function (ground truth) as defined by Marschner and Lobb [27], and $f_r: D \rightarrow \mathbb{R}$ the reconstructed function (i.e., f is sampled on a lattice and then reconstructed from these samples). For $i \in \{1,2\}$, we define

$$e_i(x)=|f(x)-f_r(x)|^i \qquad (20)$$

as the pointwise L_i error. Therefore, the average expected L_i error is

$$\mu_i=\int_D e_i(x)\,dx, \qquad (21)$$

and the standard deviation is obtained via

$$\sqrt{\sigma_i}=\sqrt{\int_D (e_i(x)^2-\mu_i^2)\,dx}=\sqrt{\int_D (e_i(x)-\mu_i)^2\,dx}. \qquad (22)$$

By taking N random samples $\xi_j \in D$, $j=1\ldots N$, Eq. (21) can be approximated by the sum

$$\mu_i=\int_D e_i(x)\,dx \approx \frac{1}{N}\sum_{j=1}^{N} e_i(\xi_j). \qquad (23)$$

We conducted the experiment by sampling the ML test function on a CC lattice of size $40\times40\times40$ and on a BCC lattice of size $31\times31\times62$. The CC lattice has a slightly higher sampling density than the BCC lattice. We compute f_r at position ξ_j by convolving these lattices with the different filters (box splines, BCC B-splines, and CC B-splines). Thus, we can compute the pointwise error $e_i(\xi_j)$. Having computed $e_i(\xi_j)$ for all $j=1\ldots N$, we estimate μ_i and $\sqrt{\sigma_i}$ for each lattice and reconstruction kernel by taking N=100,000 random samples that are identical in each computation; i.e., we precompute N=100,000 random samples and use them in each estimation of μ_i and $\sqrt{\sigma_i}$ for each lattice and filter. This allows for a numerical comparison of the different reconstruction qualities of the filters. The numerical results for the L_1 and L_2 error are found in Table 4.

If examining the L_1 error (columns two and three), it is noticeable that linear interpolation (linear box spline

Table 4
Numerical comparison between the three different filter classes (box splines, BCC B-splines, and CC B-splines).

Filter	L_1 Error		L_2 Error	
	μ_1	$\sqrt{\sigma_1}$	μ_2	$\sqrt{\sigma_2}$
Box splines				
Nearest neighbor	0.035139	0.031392	0.002220	0.003581
Linear	0.022240	0.019245	0.000865	0.001315
Quintic	0.034278	0.021428	0.001634	0.001594
Quintic (prefiltering)	0.008349	0.010025	0.000170	0.000367
BCC B-splines				
Trilinear	0.022640	0.016787	0.000794	0.000976
Tricubic	0.010683	0.009625	0.000207	0.000305
CC B-splines				
Nearest neighbor	0.035489	0.031308	0.002240	0.003588
Trilinear	0.025813	0.020246	0.001076	0.001386
Tricubic	0.038570	0.023649	0.002047	0.001940
Tricubic (prefiltering)	0.009364	0.011484	0.000220	0.000480

Columns two to five show the average expected L_1 and L_2 error μ_1 and μ_2 as well as the corresponding standard deviations $\sqrt{\sigma_1}$ and $\sqrt{\sigma_2}$.

($\mu_1 = 0.022240$) and trilinear CC B-spline ($\mu_1 = 0.025813$)) has a smaller numerical error than the quintic box spline ($\mu_1 = 0.034278$) and tricubic CC B-spline ($\mu_1 = 0.038570$) (no prefiltering). Results improve drastically when prefiltering is used. Both, the quintic box spline ($\mu_1 = 0.008349$) and the tricubic CC B-spline ($\mu_1 = 0.009364$) show the smallest error for their lattice. Furthermore, the box splines perform better than their CC counterparts.

Note that for this experiment, the trilinear CC B-spline is more accurate than the tricubic CC B-spline, and the linear box spline is more accurate than the quintic box spline. A possible explanation could be that for random samples lying close to a lattice point, the interpolating filters give more accurate results compared to the smoothing C^2 filters without prefiltering.

As for the BCC B-splines, the trilinear BCC B-spline ($\mu_1 = 0.022640$) shows a lower error than the trilinear CC B-spline ($\mu_1 = 0.025813$), but performs slightly worse than the linear box spline ($\mu_1 = 0.022240$). The tricubic BCC B-spline ($\mu_1 = 0.010683$) does not perform as well as the prefiltered quintic box spline ($\mu_1 = 0.008349$) and prefiltered tricubic CC B-spline ($\mu_1 = 0.009364$). The same observations hold for the L_2 error.

6. Conclusion and future work

In this paper, for the first time, we have demonstrated the feasibility of box splines on the BCC lattice for interactive volume rendering. By converting the box spline from its B-form into its pp-form, we reduced the branching needed to a minimum, which is necessary for an interactive GPU implementation. Using discrete prefiltering, volumetric data stored on BCC lattices can be rendered using box splines showing notable improvements in terms of image quality compared to Csébfalvi et al.'s method [14] and traditional B-spline filtering for the CC lattice, however, at a performance penalty.

While our method of evaluating box splines on current graphics hardware is not the fastest one, it is important for two reasons: (a) the quintic box spline is the most accurate way of reconstructing BCC data using a compact kernel which has been independently confirmed by the work of Csébfalvi [9]; and (b) the current class of graphics hardware that favors CC lattices might evolve further where BCC lattices might improve their relative performance [35]. We did show that box splines are comparable when removing the advantage of special circuitry that favors CC lattices (i.e., when using NN lookups).

Furthermore, we have confirmed that discrete prefiltering in the context of volume rendering enhances reconstruction accuracy which improves image quality. Since discrete prefiltering is a preprocessing step which only needs to be performed once, this technique should always be used.

For the future, we plan to investigate the possibilities of using trilinear interpolation texture lookups for box splines to increase rendering speed. Other possibilities are the design and simulation of a graphics processing unit supporting BCC lattices with special purpose units for box splines or using new more flexible rendering pipelines such as the Larrabee architecture [35]. Another issue is image quality using discrete prefiltering; i.e., we plan to integrate a regularizer in the prefiltering step in order to introduce a controllable smoothing factor which we believe will improve reconstruction quality in the presence of noise.

7. Acknowledgements

This work was supported in part (DVDV) by the Swiss National Science Foundation (Grant PP00P2-123438) and the Natural Science and Engineering Research Council of Canada. The authors would like to thank Dr. Ralf Spörle and Prof. Dr. Bernhard G. Herrmann, Max-Planck-Insititut für molekulare Genetik, Berlin-Dahlem, Germany for the mouse embryo OPT scan.

Appendix A. GLSL shader code

```
/*
Variables needed
*/

/*
Let nx, ny, nz be the size of the BCC texture:
*/

uniform float nx, ny, nz;

vec3 size_volume = vec3 (2*nx-1, 2*ny-1, nz-1);
vec3 oneOverVoxels = vec3 (1.0/(nx-1.0)),
1.0/(ny-1.0),
1.0/(nz-1.0));
// conversion between 3D coordinates and texture
  index
vec3 convert = vec3 (0.5, 0.5, 1.0);


/*Linear Box Spline
Input: vec3 pos scaled to [0,1]^3 (texture coordinates)
*/

float BCC_Linear (vec3 pos)
{
vec3 P1, P2, P3, P4;
float D1, D2, D3, D4, mymin, mymid, mymax;

vec3 posOS = pos * size_volume;
vec3 abc = vec3 (posOS.x+posOS.y,
posOS.x+posOS.z,
posOS.y+posOS.z) * .5;

// truncate this point which results
// in the first of the four neighbors
vec3 floors = floor (abc);

// truncation: we shift p into the origin
// of the BCC coordinate system
abc = abc - floors;

// transform the first neighbor back to
// the Cartesian coordinate system
P1 = vec3 (floors.x+floors.y-floors.z,
floors.x-floors.y+floors.z,
-floors.x+floors.y+floors.z);
// the second neighbor is found immediately
P2 = P1 + vec3(1.0, 1.0, 1.0);
// assume case: mymax = alpha
P3 = P1 + vec3(1.0, 1.0, -1.0);
// assume case: mymin = gamma
P4 = P1 + vec3(2.0, 0.0, 0.0);

// sorting
vec4 sorting = vec4 (1.0, 0.0, 0.0, 0.0);

sorting.y = max (abc.x, max (abc.y, abc.z));
sorting.z = min (abc.x, min (abc.y, abc.z));
sorting.w = (abc.x + abc.y + abc.z) - sorting.y -
 sorting.z;
// get the missing two neighbors
P3 += ((sorting.y==abc.y)*vec3(0.0,-2.0,2.0)
+ (sorting.y==abc.z)*vec3(-2.0,0.0,2.0));
P4 += ((sorting.z==abc.x)*vec3(-2.0,0.0,2.0)
+ (sorting.z==abc.y)*vec3(-2.0,2.0,0.0));

// four texture lookups
D1 = texture3D (tex, (floor (P1*convert)) *
 oneOverVoxels);
D2 = texture3D (tex, (floor (P2*convert)) *
 oneOverVoxels);
D3 = texture3D (tex, (floor (P3*convert)) *
 oneOverVoxels);
D4 = texture3D (tex, (floor (P4*convert)) *
 oneOverVoxels);

// interpolate using barycentric coordinates
vec4 values = vec4 (D1, D3-D1, D2-D4, D4-D3);
return dot (sorting, values);
}


#define one_over_6 0.1666666666666666
#define two_over_24 0.083333333333333329
#define six_over_120 0.05

#define BCC2CC(index3D,x0,y0,z0) \
index3D = floor(vec3(x0*0.5, y0*0.5, z0));

#define FILL_PPIPED(p0,p1,p2,p3,p4,p5,p6,p7,Q) {\
BCC2CC(index3D,x0,y0,z0) \
p0 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,(x0-(Q.x)),(y0+(Q.y)),(z0+(Q.z)))
 \
p1 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,(x0+(Q.x)),(y0-(Q.y)),(z0+(Q.z)))
 \
p2 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,(x0+(Q.x)),(y0+(Q.y)),(z0-(Q.z)))
 \
p3 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,(x0+2*(Q.x)),y0,z0) \
p4 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,x0,(y0+2*(Q.y)),z0) \
p5 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,x0,y0,(z0+2*(Q.z))) \
p6 = texture3D (tex, (index3D)*oneOverVoxels); \
BCC2CC(index3D,(x0+(Q.x)),(y0+(Q.y)),(z0+(Q.z)))
 \
p7 = texture3D (tex, (index3D)*oneOverVoxels); \
}


#define RHO_FAST(alpha,beta,gamma) \
-alpha3*(gamma*beta-0.5*alpha*(gamma+beta)+0.3
 *alpha2);

#define RHO22(A, B, G) (-((A)*(A))*(A) \
*(one_over_6*(G)*(B)-two_over_24*(A) \
*((G)+(B))+six_over_120 *((A)*(A))))

#define CONV_PPIPED(value,pa,pb,alpha,beta,gamma){\
float p123 = pa[1] + pa[2] + pa[3]; \
float p0 = 4.0 * pa[0]; \
float alpha2 = alpha * alpha; \
float alpha3 = one_over_6 * alpha2 * alpha; \
float base_rho = RHO_FAST(alpha,beta,gamma); \
float base_rho2 = base_rho - 0.5 * alpha3 * alpha; \
value += (-2.5*p0+4.0*(p123)-
 2.0*(pb[0]+pb[1]+pb[2])+pb[3]) \
* base_rho; alpha2 = base_rho2 + alpha3 * beta; \
base_rho2 += alpha3 * gamma; \
value += (p0-2.0*(p123-pa[1])+pb[0]) * (alpha2) \
+ (p0-2.0*(p123-pa[2])+pb[1]) * (base_rho2) \
+ (-.5*p0+pa[3]) \
* (base_rho2 + alpha2 - alpha3 - base_rho); \
alpha -= 1.0; alpha2 = beta * beta; \
alpha3 = one_over_6 * alpha2 * beta; \
base_rho = RHO_FAST(beta, gamma, alpha); \
value += (p0-2.0*(p123-pa[3])+pb[2]) \
* (base_rho); p0 = -.5 * p0; \
value += (p0+pa[2]) * (base_rho + alpha3 * (alpha - .5
 * beta)) \
+ (p0+pa[1]) * RHO22(gamma, alpha, beta-1.0) + (-.5 *
 p0) \
* RHO22(alpha, beta-1.0, gamma-1.0); \
}


/*
Quintic Box Spline
Input: vec3 pos scaled to [0,1]^3 (texture coordinates)
*/

float BCC_Quintic (vec3 pos)
{
// convert point into tetrahedron of focus
vec3 posOS = pos * size_volume;
vec3 bcc_coords = vec3 ((posOS.x+posOS.y),
(posOS.x+posOS.z),
(posOS.y+posOS.z)) * 0.5;

float alpha = bcc_coords.x;
float beta = bcc_coords.y;
float gamma = bcc_coords.z;

vec3 floors = floor (bcc_coords);

alpha = alpha - floors.x;
beta = beta - floors.y;
gamma = gamma - floors.z;

float x1, y1, z1;
// P1 = (x0, y0, z0)
float x0 = x1 = floors.x+floors.y-floors.z;
float y0 = y1 = floors.x-floors.y+floors.z;
float z0 = z1 = floors.y+floors.z-floors.x;
vec3 index3D;

vec4 p1a, p1b, p2a, p2b, p3a, p3b, p4a, p4b;

vec3 Q1, Q2, Q3, Q4;
vec3 P2, P3, P4;

// variables needed for the sorting
float alpha_GE_beta = alpha >= beta;
float beta_GE_gamma = beta >= gamma;
float alpha_GE_gamma = alpha >= gamma;
float mymax, mymid, mymin;

// sorting
mymax = max (alpha, max (beta, gamma));
mymin = min (alpha, min (beta, gamma));
mymid = (alpha + beta + gamma) - mymax - mymin;

float i = (alpha_GE_beta*4.0
+ beta_GE_gamma*2.0
+ alpha_GE_gamma);

vec3 I1 = equal (vec3(i,i,i), vec3(7,5,4));
vec3 I2 = equal (vec3(i,i,i), vec3(3,2,0));

// determine neighbors and offsets
P2 = vec3 (1.0, 1.0, 1.0);
Q1 = vec3(-1.0, -1.0, -1.0);

P3 = vec3(1,1,-1);
P3 += (I1.z+I2.z)*vec3(-2,0,2) +
  (I2.x+I2.y)*vec3(0,-2,2);

P4 = vec3(2,0,0);
P4 += (I1.y+I1.z)*vec3(-2,2,0) +
  (I2.y+I2.z)*vec3(-2,0,2);

Q2 = vec3(-1,1,1);
Q2 += (I1.y+I1.z)*vec3(2,-2,0) +
  (I2.y+I2.z)*vec3(2,0,-2);

Q4 = vec3(1,-1,1);
Q4 += (I1.y+I2.y)*vec3(-2,2,0) +
  (I1.z+I2.x)*vec3(0,2,-2);

Q3 = P3;

// 32 texture lookups
FILL_PPIPED(p1a[0], p1a[1], p1a[2], p1a[3],
p1b[0], p1b[1], p1b[2], p1b[3], Q1);
x0 = x1+P2.x; y0 = y1+P2.y; z0 = z1+P2.z;
FILL_PPIPED(p2a[0], p2a[1], p2a[2], p2a[3],
p2b[0], p2b[1], p2b[2], p2b[3], Q2);
x0 = x1+P3.x; y0 = y1+P3.y; z0 = z1+P3.z;
FILL_PPIPED(p3a[0], p3a[1], p3a[2], p3a[3],
p3b[0], p3b[1], p3b[2], p3b[3], Q3);
x0 = x1+P4.x; y0 = y1+P4.y; z0 = z1+P4.z;
FILL_PPIPED(p4a[0], p4a[1], p4a[2], p4a[3],
p4b[0], p4b[1], p4b[2], p4b[3], Q4);

// according to the 6 possible cases
// we have to permute the data points
vec4 p1a_r = I1.x*p1a + I1.y*p1a.xzyw + I1.z*p1a.xzwy
+ I2.x*p1a.xywz + I2.y*p1a.xwyz + I2.z*p1a.xwzy;
vec4 p1b_r = I1.x*p1b + I1.y*p1b.yxzw + I1.z*p1b.yzxw
+ I2.x*p1b.xzyw + I2.y*p1b.zxyw + I2.z*p1b.zyxw;
vec4 p2a_r = I1.x*p2a.xwzy + I1.y*p2a.xwyz +
  I1.z*p2a.xywz
+ I2.x*p2a.xzwy + I2.y*p2a.xzyw + I2.z*p2a;
vec4 p2b_r = I1.x*p2b.zyxw + I1.y*p2b.zxyw +
  I1.z*p2b.xzyw
+ I2.x*p2b.yzxw + I2.y*p2b.yxzw + I2.z*p2b;
vec4 p3a_r = I1.x*p3a.xwzy + I1.y*p3a.xwyz +
  I1.z*p3a.xywz
+ I2.x*p3a.xzwy + I2.y*p3a.xzyw + I2.z*p3a;
vec4 p3b_r = I1.x*p3b.zyxw + I1.y*p3b.zxyw +
  I1.z*p3b.xzyw
 + I2.x*p3b.yzxw + I2.y*p3b.yxzw + I2.z*p3b;
vec4 p4a_r = I1.x*p4a + I1.y*p4a.xzyw + I1.z*p4a.xzwy
 + I2.x*p4a.xywz + I2.y*p4a.xwyz + I2.z*p4a.xwzy;
vec4 p4b_r = I1.x*p4b + I1.y*p4b.yxzw + I1.z*p4b.yzxw
 + I2.x*p4b.xzyw + I2.y*p4b.zxyw + I2.z*p4b.zyxw;

 // do the convolution. Before each CONV_PPIPED call
 // mymin, mymid and mymax are used to transform
 // each parallelepiped and tetrahedron of focus.
float result = 0.0;
alpha=mymax-1.0; beta=mymid-1.0; gamma=mymin-1.0;
CONV_PPIPED(result, p1a_r, p1b_r, alpha, beta,
  gamma);
alpha=-mymin; beta=mymax-mymin-1.0; gamma=mymid-
  mymin-1.0;
CONV_PPIPED(result, p2a_r, p2b_r, alpha, beta,
  gamma);
alpha=(-mymax+mymid); beta=(-mymax+mymin);
  gamma=(-mymax);
CONV_PPIPED(result, p3a_r, p3b_r, alpha, beta,
  gamma);
alpha=(-mymid+mymin); beta=(-mymid);
  gamma=(mymax-mymid-1.0);
CONV_PPIPED(result, p4a_r, p4b_r, alpha, beta,
  gamma);

return result;
}
```

References

[1] Alim UR, Möller T. A fast Fourier transform with rectangular output on the BCC and FCC lattices. In:Proceedings of the eighth international conference on sampling theory and applications (SampTA'09), Marseille, France, May 18–22, 2009.
[2] Blu T, Thévenaz P, Unser M. Generalized interpolation: higher quality at no additional cost. In: Proceedings of IEEE international conference on image processing, 1999. p. 667–71.
[3] Boor CD, Höllig K, Riemenschneider S. Box splines, Vol. 98. New York: Springer Verlag; 1993.
[4] Cabral B, Cam N, Foran J. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In: Proceedings of IEEE symposium on volume visualization, 1994. p. 91–8.
[5] Carlbom I. Optimal filter design for volume reconstruction and visualization. In: Proceedings of IEEE visualization, 1993. p. 54–61.
[6] Carr H, Möller T, Snoeyink J. Artifacts caused by simplicial subdivision. IEEE Transactions on Visualization and Computer Graphics 2006;12(2):231–42.
[7] Condat L, Van De Ville D. Quasi-interpolating spline models for hexagonally-sampled data. IEEE Transactions on Image Processing 2007;16(5):1195–206.
[8] Condat L, Blu T, Unser M. Beyond interpolation: optimal reconstruction by quasi-interpolation. In: IEEE International Conference on Image Processing, 2005. ICIP 2005, 1:I-33-6, September 2005. doi: 10.1109/ICIP.2005.1529680.
[9] Csébfalvi B. An evaluation of prefiltered B-spline reconstruction for quasi-interpolation on the body-centered cubic lattice. IEEE Transactions on Visualization and Computer Graphics 2009;99.
[10] Csébfalvi B. Prefiltered Gaussian reconstruction for high-quality rendering of volumetric data sampled on a body-centered cubic grid. In: Proceedings of IEEE visualization, 2005. p. 40–8.
[11] Csébfalvi B. BCC-splines: generalization of B-splines for the body-centered cubic lattice. Journal of WSCG (Winter School of Computer Graphics) 2008;16(1–3):81–8.
[12] Csébfalvi B. An evaluation of prefiltered reconstruction schemes for volume rendering. IEEE Transactions on Visualization and Computer Graphics 2008;14(2):289–301.
[13] Csébfalvi B, Domonkos B. Pass-band optimal reconstruction on the body-centered cubic lattice. In: Deussen O, Keim DA, Saupe D, editors. Proceedings of conference on vision, modeling, and visualization (VMV). Aka GmbH; 2008. p. 71–80. ISBN: 978-3-89838-609-8 <http://dblp.uni-trier.de/db/conf/vmv/vmv2008.html#CsebfalviD08>.
[14] Csébfalvi B, Hadwiger M. Prefiltered B-spline reconstruction for hardware-accelerated rendering of optimally sampled volumetric data. In: Proceedings of workshop on vision, modeling and visualization, 2006. p. 325–32.
[15] Dutta Roy SC, Kumar B. Digital differentiators. In: Handbook of statistics, Vol. 10. Elsevier Science Publishers B.V., North-Holland; 1993. p. 159–205.

[16] Engel K, Hadwiger M, Kniss JM, Rezk-Salama C, Weiskopf D. Real-time volume graphics. A K Peters, Ltd; 2006.
[17] Entezari A. Optimal sampling lattices and trivariate box splines. PhD thesis, Simon Fraser University, Burnaby, Canada; 2007.
[18] Entezari A, Möller T. Extensions of the Zwart–Powell box spline for volumetric data reconstruction on the Cartesian lattice. IEEE Transactions on Visualization and Computer Graphics 2006;12(5):1337–44 , doi:10.1109/TVCG.2006.141. ISSN 1077-2626.
[19] Entezari A, Dyer R, Möller T. Linear and cubic box splines for the body centered cubic lattice. In: Proceedings of IEEE visualization, 2004. p. 11–8.
[20] Entezari A, Morey J, Mueller K, Ostromoukhov V, Van De Ville D, Möller T. Point lattices in computer graphics and visualization. Tutorial presented at IEEE visualization, 2005.
[21] Entezari A, Van De Ville D, Möller T. Practical box splines for reconstruction on the body centered cubic lattice. IEEE Transactions on Visualization and Computer Graphics 2008;14(2):313–28.
[22] Entezari A, Mirzargar M, Kalantari L. Quasi-interpolation on the body centered cubic lattice. Computer Graphics Forum (Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization) 2009;28(3): 1015–22.
[23] Finkbeiner B, Alim UR, Van De Ville D, Möller T. High-quality volumetric reconstruction on optimal lattices for computed tomography. Computer Graphics Forum (Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization) 2009;28(3):1023–30.
[24] Frigo M, Johnson SG. FFTW: an adaptive software architecture for the FFT. Proceedings of ICASSP 2008;3:1381–4.
[25] Kalbe T, Zeilfelder F. Hardware-accelerated, high-quality rendering based on trivariate splines approximating volume data. Computer Graphics Forum 2008;27(2):331–40.
[26] Krüger J, Westermann R. Acceleration techniques for GPU-based volume rendering. In: Proceedings of IEEE visualization, 1994. p. 100–7.
[27] Marschner SR, Lobb RJ. An evaluation of reconstruction filters for volume rendering. In: Proceedings of the IEEE conference on visualization, 1994. p. 100–7.
[28] Mattausch O. Practical reconstruction schemes and hardware-accelerated direct volume rendering on body-centered cubic grids. Master's thesis, Vienna University of Technology; 2003.
[29] Meng T, Smith B, Entezari A, Kirkpatrick AE, Weiskopf D, Kalantari L, et al. On visual quality of optimal 3D sampling and reconstruction. In: Graphics interface, May 2007. p. 265–72.
[30] Möller T, Machiraju R, Mueller K, Yagel R. A comparison of normal estimation schemes. In: Proceedings of the IEEE conference on visualization, October 1997. p. 19–26.
[31] Möller T, Mueller K, Kurzion Y, Machiraju R, Yagel R. Design of accurate and smooth filters for function and derivative reconstruction. In: Proceedings of the symposium on volume visualization, October 1998. p. 143–51.
[32] Petersen DP, Middleton D. Sampling and reconstruction of wave-number-limited functions in *N*-dimensional Euclidean spaces. Information and Control 1962;5(4):279–323.
[33] Rössl C, Zeilfelder F, Nürnberger G, Seidel H-P. Visualization of volume data with quadratic super splines. In: Proceedings of IEEE Visualization, 2003. p. 52–9.
[34] Röttger S, Guthe S, Weiskopf D, Ertl T. Smart hardware-accelerated volume rendering. In: Proceedings of EG/IEEE TCVG symposium on visualization, 2003. p. 231–8.
[35] Seiler L, Carmean D, Sprangle E, Forsyth T, Abrash M, Dubey P, et al. Larrabee: a many-core $\times$86 architecture for visual computing. ACM Transactions on Graphics 2008;27(3):1–15.
[36] Sharpe J. Optical projection tomography. Annual Review of Biomedical Engineering August 2004;6:209–28.
[37] Sigg C, Hadwiger M. Fast third-order texture filtering. In: GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation, 2005. p. 313–29.
[38] Theußl T, Möller T, Gröller E. Optimal regular volume sampling. In: Proceedings of IEEE visualization, 2001. p. 91–8.
[39] Van De Ville D, Blu T, Unser M, Philips W, Lemahieu I, Van de Walle R. Hex-splines: a novel spline family for hexagonal lattices. IEEE Transactions on Image Processing 2004;13(6):758–72.