GPU Isosurface Raycasting of FCC Datasets

Minho Kim[a]

[a]*School of Computer Science, University of Seoul, Republic of Korea*

Abstract

This paper presents an efficient and accurate isosurface rendering algorithm for the natural C^1 splines on the face-centered cubic (FCC) lattice. Leveraging fast and accurate evaluation of a spline field and its gradient, accompanied by efficient empty-space skipping, the approach generates high-quality isosurfaces of FCC datasets at interactive speed (20–70 fps). The pre-processing computation (quasi-interpolation and min/max cell construction) is improved 20 to 30-fold by OpenCL kernels. In addition, a novel indexing scheme is proposed that allows an FCC dataset to be stored as a four-channel 3D texture. When compared with other reconstruction schemes on the Cartesian and BCC (body-centered cubic) lattices, this method can be considered a practical reconstruction scheme that offers both quality and performance. The OpenCL and GLSL (OpenGL Shading Language) source codes are provided as a reference.

Keywords: Volume rendering, Raycasting, FCC lattice, GPU, Box-spline

1. Introduction

While it has been known for a while that the BCC and FCC lattices are more efficient sampling lattices than the classical Cartesian lattice, they have not been widely used in practice due to the computational overhead and lack of proper reconstruction filters tailored for them. Some researchers recently investigated spline-based reconstruction schemes on those lattices and demonstrated their superior power [6, 2, 5, 14, 13]. To further utilize the FCC lattice, this paper introduces a real-time GPU (graphics processing unit) isosurface raycaster. The contributions are

- fast pre-processing using OpenCL computing kernels,
- fast, accurate and stable evaluation of a spline field and its gradient,
- efficient empty space skipping, and
- a novel indexing scheme that allows an FCC dataset to be stored compactly as a four-channel (RGBA) 3D texture.

2. Previous Work

Since programmable GPUs were first introduced, GPU raycasting has been actively investigated, but mostly limited to datasets on the Cartesian lattice. See Engel et al. [4] for a complete list of volume rendering techniques. For non-Cartesian lattices, no GPU raycaster has been investigated for FCC datasets, but a couple have been investigated for BCC datasets. Csébfalvi and Hadwiger [2] implemented a GPU raycaster for the tri-cubic B-spline filter. Thanks to the hardware-accelerated tri-linear interpolation, their method is very fast, but the resulting volume is over-smoothed. They stored a BCC dataset in two separate 3D textures on the GPU. While this strategy is suitable for their specific filter, it introduces overhead for other reconstruction filters. Based on the efficient evaluation method developed by Entezari et al. [7], Finkbeiner et al. [8] implemented a GPU raycaster for the eight-direction box-spline filter [6]. They packed the BCC dataset as a one-channel 3D texture by shifting every second layer along the z direction by $(1, 1, 0)$. Such a conversion works well in most cases, but it is not symmetric and therefore less cache-friendly along the z-axis.

Künsch et al. [15] showed that the FCC lattice is optimal for a dataset sampled at a low rate. Ibáñez et al. [11] investigated a ray-tracer for FCC datasets. Petkov et al. [17] showed the superiority of the lattice-Bolzmann method on the FCC lattice compared to the Cartesian lattice. Leveraging the isotropic structure of the FCC lattice, Qiu et al. [18] proposed an efficient global illumination method on the FCC lattice by discretizing photon tracing. The six-direction box-spline filter on the FCC lattice was first proposed by Entezari [5] and later investigated in detail by Kim et al. [14].

Hadwiger et al. [9] proposed an efficient empty space skipping method for a GPU raycaster, where tight entry and exit ray positions are computed by selectively rendering quad faces of blocks enclosing voxels based on their min/max values.

Email address: minhokim@uos.ac.kr (Minho Kim)

3. Background

In this section we first describe the FCC lattice and box-splines. For the complete theory of box-splines, refer to the book by de Boor et al. [3]. Then we review the six-direction box-spline M_{fcc} and its properties.

Hereafter vectors are typeset as bold lower case letters, *e.g.*, $\boldsymbol{x}$, matrices are typeset as bold upper case letters, *e.g.*, $\boldsymbol{M}$, and the j-th component of a vector $\boldsymbol{x}$ is denoted as $\boldsymbol{x}(j)$,

3.1. The FCC Lattice

A three-dimensional lattice $\mathcal{L}$ is defined by all of the integer linear combinations of a non-singular *generator matrix* $\boldsymbol{G}$:

$$\mathcal{L} := \{\boldsymbol{G}\boldsymbol{j} : \boldsymbol{j} \in \mathbb{Z}^3, \boldsymbol{G} \in \mathbb{R}^{3\times 3}, \det \boldsymbol{G} \neq 0\}.$$

The FCC lattice $\mathbb{Z}_{\text{fcc}}$ is defined as a subset of the Cartesian lattice $\mathbb{Z}^3$ where the sum of its components is even, *i.e.*,

$$\mathbb{Z}_{\text{fcc}} := \{\boldsymbol{j} \in \mathbb{Z}^3 : \boldsymbol{j}(1) + \boldsymbol{j}(2) + \boldsymbol{j}(3) \text{ is even.}\},$$

or by the generator matrix

$$\boldsymbol{G}_{\text{fcc}} := \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

Due to the frequent use of the three column vectors of $\boldsymbol{G}_{\text{fcc}}$, we denote them as $\boldsymbol{\xi}_{011}$, $\boldsymbol{\xi}_{101}$, and $\boldsymbol{\xi}_{110}$, respectively. Note that the FCC lattice can be decomposed into four Cartesian lattices scaled by 2:

$$\mathbb{Z}_{\text{fcc}} = (2\mathbb{Z}^3) \cup (2\mathbb{Z}^3 + \boldsymbol{\xi}_{110}) \cup (2\mathbb{Z}^3 + \boldsymbol{\xi}_{101}) \cup (2\mathbb{Z}^3 + \boldsymbol{\xi}_{011}).$$

Also note that $\mathbb{Z}^3 \backslash \mathbb{Z}_{\text{fcc}}$ is another FCC lattice. The FCC lattice shows better sampling efficiency than the Cartesian lattice [1]. While the BCC lattice is the optimal 3D sampling lattice for band-limited and isotropic signals, the FCC lattice is optimal when the signal is sampled at a low rate, as indicated by Künsch et al. [15].

3.2. Box-Splines

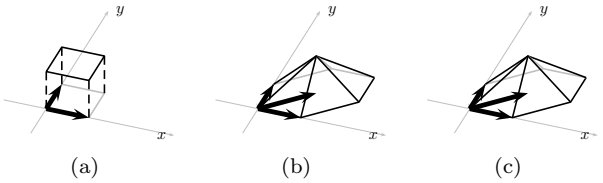
In the subsequent text, a matrix also denotes a set of column vectors allowing multiplicity, depending on the context.

A box-spline is a piecewise polynomial with a finite support and certain continuity and is uniquely defined by a *direction matrix*. Given an $n \times m$ (usually $n \leq m$) direction matrix $\boldsymbol{\Xi}$, a box-spline $M_{\boldsymbol{\Xi}}$ can be constructed by taking consecutive directional convolutions along each column direction (Figure 1). In other words, starting from the base case ($n = m$)

$$M_{\boldsymbol{\Xi}}(\boldsymbol{x}) := \frac{1}{|\det \boldsymbol{\Xi}|}\chi_{\boldsymbol{\Xi}}(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{R}^n$$

where $\boldsymbol{\Xi}$ is invertible and $\chi_{\boldsymbol{\Xi}}(\boldsymbol{x})$ is the characteristic function on the half-open parallelepiped $\boldsymbol{\Xi}[0,1)^m$. A box-spline defined by the direction matrix $\boldsymbol{\Xi} \cup \{\boldsymbol{\xi}\}$ can be recursively defined as

$$M_{\boldsymbol{\Xi}\cup\{\boldsymbol{\xi}\}}(\boldsymbol{x}) := \int_0^1 M_{\boldsymbol{\Xi}}(\boldsymbol{x} - t\boldsymbol{\xi})dt, \quad \boldsymbol{\xi} \in \mathbb{R}^n.$$



(a) (b) (c)

Figure 1: Construction of box-splines with direction matrices (a) $\left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$, (b) $\left[\begin{smallmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{smallmatrix}\right]$ and (c) $\left[\begin{smallmatrix} 1 & 0 & 1 & -1 \\ 0 & 1 & 1 & 1 \end{smallmatrix}\right]$ via consecutive directional convolutions.

Given a discrete dataset on the Cartesian lattice, we can reconstruct a continuous spline $s(\boldsymbol{x})$ by a convolution of the dataset $V : \mathbb{Z}^n \mapsto \mathbb{R}$ and a box-spline filter $M_{\boldsymbol{\Xi}}$:

$$s(\boldsymbol{x}) := V * M_{\boldsymbol{\Xi}} := \sum_{\boldsymbol{j}\in\mathbb{Z}^n} V(\boldsymbol{j}) M_{\boldsymbol{\Xi}}(\boldsymbol{x} - \boldsymbol{j}).$$

But in most cases we can obtain better reconstruction by applying a discrete quasi-interpolation prefilter q on the dataset beforehand:

$$(V \star q) * M_{\boldsymbol{\Xi}},$$

where $\star$ denotes the *discrete convolution*

$$(V \star q)(\boldsymbol{k}) := \sum_{\boldsymbol{j}\in\mathbb{Z}^n} V(\boldsymbol{j}) q(\boldsymbol{k} - \boldsymbol{j}).$$

Then the reconstructed spline annihilates all the lower terms of the Taylor expansion of the input signal reducing approximation error. Refer to de Boor et al. [3] for the procedure to compute discrete quasi-interpolation prefilters.

While the theory put forth by de Boor et al. [3] is based on shifts on the Cartesian lattice, box-splines on non-Cartesian lattices can be easily obtained by change-of-variables [13].

3.3. Six-Direction Box-Spline on the FCC Lattice

The six-direction box-spline on the FCC lattice M_{fcc} is the 'centered' and 'scaled' version of the box-spline defined by the direction matrix $\boldsymbol{\Xi}_{\text{fcc}}$ (Figure 2(a)) [14]:

$$M_{\text{fcc}}(\boldsymbol{x}) := |\det \boldsymbol{G}_{\text{fcc}}| M_{\boldsymbol{\Xi}_{\text{fcc}}}(\boldsymbol{x} + \boldsymbol{\xi}_c),$$

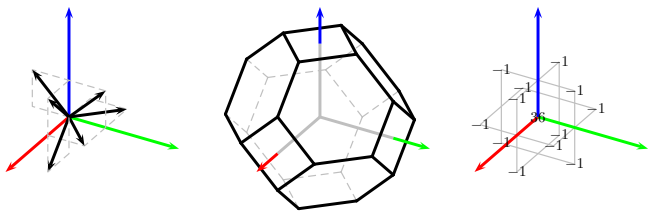
where $|\det \boldsymbol{G}_{\text{fcc}}| = 2$,

$$\boldsymbol{\Xi}_{\text{fcc}} := \begin{bmatrix} 1 & -1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{bmatrix},$$

and

$$\boldsymbol{\xi}_c := \frac{1}{2} \sum_{\boldsymbol{\xi} \in \boldsymbol{\Xi}_{\mathrm{fcc}}} \boldsymbol{\xi} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Its (total) polynomial degree is $6 - 3 = 3$ and the approximation order is 2. The tensor-product counterpart with the same approximation order is the tri-quadratic B-spline with polynomial degree $9 - 3 = 6$. Its support has the shape of a truncated octahedron (Figure 2(b)).



(a) (b) (c)

Figure 2: The (a) directions, (b) support, and (c) quasi-interpolation prefilter (scaled by 24) of M_{fcc}.

Given a discrete volume dataset V sampled on the FCC lattice, a continuous spline $s(\boldsymbol{x})$ can be constructed by a convolution with M_{fcc} on the FCC lattice. One of the discrete quasi-interpolation prefilters of M_{fcc} that provide the optimal approximation order 2 is given by Kim et al. [14] (Figure 2(c)).

$$q_{\mathrm{fcc}}(\boldsymbol{j}) := \begin{cases} 36/24 & \boldsymbol{j} = \boldsymbol{0} \\ -1/24 & \boldsymbol{j} \in \{\pm\boldsymbol{\xi} : \boldsymbol{\xi} \in \boldsymbol{\Xi}_{\mathrm{fcc}}\} \\ 0 & \text{otherwise.} \end{cases}$$

4. GPU Raycasting of FCC Datasets

In this section, an efficient evaluation algorithm for the spline and its gradient on the FCC lattice is outlined, followed by an efficient empty space skipping method and a novel indexing scheme for FCC datasets.

4.1. Evaluation of Splines on the GPU

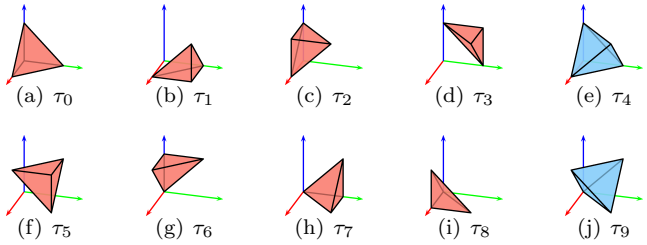
For evaluation of a spline, it is important to know the polynomial structure induced by the knot planes of M_{fcc}, since all the points in the same polynomial piece share the *stencil*, which is the relative location of finite data on $\mathbb{Z}_{\mathrm{fcc}}$ required for evaluation. Kim et al. [14] already analyzed the spline structure, but here we do it in a slightly different way since we need to evaluate its gradient too.

There are seven knot planes generated by M_{fcc}, three of which are axis-aligned and decompose the whole space into cubes; $\{\boldsymbol{j} + [0,1)^3 : \boldsymbol{j} \in \mathbb{Z}^3\}$. Depending on the lower corner $\boldsymbol{j}$ of each cube, we can split the cubes into two groups:

$$\{\boldsymbol{j} + [0,1)^3 : \boldsymbol{j} \in \mathbb{Z}_{\mathrm{fcc}}\} \text{ and } \{\boldsymbol{j} + [0,1)^3 : \boldsymbol{j} \in (\mathbb{Z}^3 \backslash \mathbb{Z}_{\mathrm{fcc}})\}.$$

Note that each group can be identified by computing the *parity* of $\boldsymbol{j}$; $\boldsymbol{j}(1)+\boldsymbol{j}(2)+\boldsymbol{j}(3)$. Then by the remaining four knot planes, cubes in each group are decomposed into five tetrahedra $\{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ and $\{\tau_5, \tau_6, \tau_7, \tau_8, \tau_9\}$ in Figure 3. Note that each tetrahedron τ_j in the second group (Figure 3(f)−−(j)) can be transformed to τ_{j-5} with a reflection with respect to the origin followed by a translation by $(1,1,1)$:

$$\tau_{j-5} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tau_j + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{for } j \in \{5,6,7,8,9\}.$$



(a) τ_0 (b) τ_1 (c) τ_2 (d) τ_3 (e) τ_4

(f) τ_5 (g) τ_6 (h) τ_7 (i) τ_8 (j) τ_9

Figure 3: Shift-invariant tetrahedral polynomial pieces induced by M_{fcc} for (top) even and (bottom) odd parity cubes.

While a spline is defined as a convolution of an infinite number of data on $\mathbb{Z}_{\mathrm{fcc}}$ and the filter M_{fcc}, due to the finite support of M_{fcc}, we only need (at most) 16 data values on $\mathbb{Z}_{\mathrm{fcc}}$. Let $\mathcal{J}_j$ be the stencil of τ_j. Then, if $\boldsymbol{x} \in \tau_j$,

$$s(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathcal{J}_j} V(\lfloor \boldsymbol{x} \rfloor + \boldsymbol{k}) M_{\mathrm{fcc}}(\boldsymbol{x} - \lfloor \boldsymbol{x} \rfloor - \boldsymbol{k}).$$

In general, we need to consider each shift-invariant polynomial piece separately in order to evaluate a spline as described above. But thanks to the symmetry of M_{fcc}, we need to consider only two types of tetrahedra, τ_0 and τ_4, for evaluation [14]. All of the other types can be transformed to one of the two by an orthogonal transformation composed of reflections and a translation (Figure 4). Let

$$\iota(j) := \begin{cases} 0 & j \in \{0,1,2,3,5,6,7,8\} \\ 4 & j \in \{4,9\} \end{cases}$$

and let $\boldsymbol{T}_j$ be the orthogonal transformation such that

$$\boldsymbol{T}_j(\tau_j) = \tau_{\iota(j)},$$

i.e., the whole tetrahedron τ_j is mapped to $\tau_{\iota(j)}$ by $\boldsymbol{T}_j$. For example,

$$\boldsymbol{T}_3(\boldsymbol{x}) := \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \left(\boldsymbol{x} - \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \right)$$

and

$$\boldsymbol{T}_9(\boldsymbol{x}) := \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Then it is straightforward that

$$\{\boldsymbol{T}_j(\boldsymbol{k}) : \boldsymbol{k} \in \mathcal{J}_j\} = \{\boldsymbol{i} : \boldsymbol{i} \in \mathcal{J}_{\iota(j)}\}.$$

Therefore we get

$$s(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathcal{J}_{\iota(j)}} V(\lfloor \boldsymbol{x} \rfloor + \boldsymbol{k}) M_{\mathrm{fcc}}(\boldsymbol{x} - \lfloor \boldsymbol{x} \rfloor - \boldsymbol{T}_j^{-1}(\boldsymbol{k})),$$

and we only need to consider evaluating polynomial pieces τ_0 and τ_4. Figure 4 shows the GLSL code piece that fetches FCC data samples for evaluation. For an input point, we first transform the cube such that we only need to consider the even parity case, and we test the local point against four knot planes to determine the tetrahedral type τ_j. Then we fetch FCC samples using transformed stencil vectors. Note that we fetch $\#(\mathcal{J}_0 \cup \mathcal{J}_4) = 19$ data samples to avoid conditional branching for evaluation.

```
ivec3 T[10] = ivec3[10](
  ivec3(0,0,0),ivec3(1,1,0),ivec3(1,0,1),
  ivec3(0,1,1),ivec3(0,0,0),ivec3(1,1,1),
  ivec3(0,0,1),ivec3(0,1,0),ivec3(1,0,0),
  ivec3(1,1,1));
ivec3 R[10] = ivec3[10](
  ivec3( 1, 1, 1),ivec3(-1,-1, 1),ivec3(-1, 1,-1),
  ivec3( 1,-1,-1),ivec3( 1, 1, 1),ivec3(-1,-1,-1),
  ivec3( 1, 1,-1),ivec3( 1,-1, 1),ivec3(-1, 1, 1),
  ivec3(-1,-1,-1));
ivec3 stencil[19] = ivec3[19](
    ivec3( 0, 0, 0),ivec3( 2, 0, 0),ivec3( 0, 2, 0),
    ivec3( 0, 0, 2),ivec3( 2, 1, 1),ivec3( 0, 1, 1),
    ivec3( 0, 1,-1),ivec3( 0,-1,-1),ivec3( 0,-1, 1),
    ivec3( 1, 2, 1),ivec3( 1, 0, 1),ivec3(-1, 0, 1),
    ivec3(-1, 0,-1),ivec3( 1, 0,-1),ivec3( 1, 1, 2),
    ivec3( 1, 1, 0),ivec3(-1, 1, 0),ivec3(-1,-1, 0),
    ivec3( 1,-1, 0));
vec3  p_local;
ivec3 origin;
origin = ivec3(floor(p_in));
p_cube = p_in-vec3(origin);
parity = (origin.x+origin.y+origin.z)&1;
p_local = p_cube;
if(parity==1) p_cube = ivec3(1,1,1) - p_cube;
if(dot(p_cube,vec3(-1,-1,-1))>-1)     itet = 0;
else if(dot(p_cube,vec3(1,1,-1))>1)   itet = 1;
else if(dot(p_cube,vec3(1,-1,1))>1)   itet = 2;
else if(dot(p_cube,vec3(-1,1,1))>1)   itet = 3;
else                                  itet = 4;
if(itet==4) type=1;
else        type=0;
vitet = vec4(itet==0,itet==1,itet==2,itet==3);
if(parity==1) itet+=5;
p_local = p_local*R[itet] + T[itet];
origin += T[itet];
ivec3 fcc;
for(int i=0 ; i<19 ; i++)
{
  fcc = origin + R[itet]*stencil[i];
  FETCH(i)
}
```

Figure 4: GLSL code for fetching 19 FCC data samples for evaluation. Note that the variable `vitet` is used in other parts of the code.

While the code in Figure 4 is easy to understand and correctly fetches 19 samples, its performance is extremely slow for several reasons. First, it contains too many conditional branches, which severely degrades the performance on the GPU. Secondly, using look-up tables is also very inefficient. Thus, we have to remove this overhead as much as possible to obtain real-time performance. Figure 5 shows the improved version where all of the conditional branches and look-up tables are removed. Moreover, the stencils are ordered carefully such that the coordinate change between adjacent stencils is minimized. Our experiment shows that replacing the code in Figure 4 with that in Figure 5 improves the performance by 10 to 16-fold.

```
vec3  p_local;
ivec3 R;
ivec3 f;
f = ivec3(floor(p_in));
p_cube = p_in-vec3(f);
parity = (f.x+f.y+f.z)&1;
p_local = p_cube;
p_cube += float(parity)*(1-2*p_cube);
vitet = vec4( dot(p_cube,vec3(-1,-1,-1))>-1,
              dot(p_cube,vec3( 1, 1,-1))> 1,
              dot(p_cube,vec3( 1,-1, 1))> 1,
              dot(p_cube,vec3(-1, 1, 1))> 1);
type = 1-(vitet.x+vitet.y+vitet.z+vitet.w);
itet = int(dot(vitet.yzw,vec3(1,2,3)))
     + 4*int(type) + parity*5;
ivec3 offset = int(vitet.y+vitet.z+vitet.w)
             *(1-ivec3(vitet.wzy));
offset += parity*(1 - 2*offset);
R = 1-2*offset;
p_local = vec3(offset) + vec3(R)*p_local;
f += offset;
R *= 2;
                                          FETCH(0)
f.x+=R.x;                                 FETCH(1)
f.x-=R.x; f.y+=R.y;                       FETCH(2)
f.y-=R.y; f.z+=R.z;                       FETCH(3)
f.x+=R.x; f.y+=R.y>>1;  f.z-=R.z>>1;      FETCH(4)
f.x-=R.x;                                 FETCH(5)
f.z-=R.z;                                 FETCH(6)
f.y-=R.y;                                 FETCH(7)
f.z+=R.z;                                 FETCH(8)
f.x+=R.x>>1;  f.y+=(3*R.y)>>1;            FETCH(9)
f.y-=R.y;                                 FETCH(10)
f.x-=R.x;                                 FETCH(11)
f.z-=R.z;                                 FETCH(12)
f.x+=R.x;                                 FETCH(13)
f.y+=R.y>>1;  f.z+=(3*R.z)>>1;            FETCH(14)
f.z-=R.z;                                 FETCH(15)
f.x-=R.x;                                 FETCH(16)
f.y-=R.y;                                 FETCH(17)
f.x+=R.x;                                 FETCH(18)
```

Figure 5: Improved GLSL code for determining the tetrahedral polynomial structure and fetching FCC data.

After collecting all the 16 data samples, we can construct a BB-form of the polynomial piece based on Table 1 in Kim et al. [14] and evaluate it using the de Casteljau algorithm.

4.2. Analytic Gradient Computation

For isosurface rendering, accurate normal vectors need to be calculated on the surface for correct shading. In most raycasting kernels, a finite difference method based on six neighbor values is used to approximate the gradient.

While this method is effective in most cases, it induces a large overhead if the evaluation is expensive, as in this case.

For a box-spline defined by Ξ, a directional derivative along $\boldsymbol{\xi} \in \Xi$, $D_{\boldsymbol{\xi}} M_{\Xi}$, can be represented as a backward difference of the box-spline defined by $\Xi \backslash \{\boldsymbol{\xi}\}$ [3]:

$$D_{\boldsymbol{\xi}} M_{\Xi} = \nabla_{\boldsymbol{\xi}} M_{\Xi \backslash \{\boldsymbol{\xi}\}},$$

where $\nabla_{\boldsymbol{\xi}}$ is the backward difference operator along $\boldsymbol{\xi}$. Therefore, a directional derivative along $\boldsymbol{\xi} \in \Xi_{\text{fcc}}$ of a spline s can be expressed as another spline as follows:

$$\begin{aligned} D_{\boldsymbol{\xi}} \left(V * M_{\text{fcc}} \right) &= V * \left(D_{\boldsymbol{\xi}} M_{\text{fcc}} \right) = V * \left(\nabla_{\boldsymbol{\xi}} 2 M_{\Xi_{\text{fcc}} \backslash \{\boldsymbol{\xi}\}} (\cdot + \boldsymbol{\xi}_c) \right) \\ &= V * M_{\boldsymbol{\xi}}, \end{aligned}$$

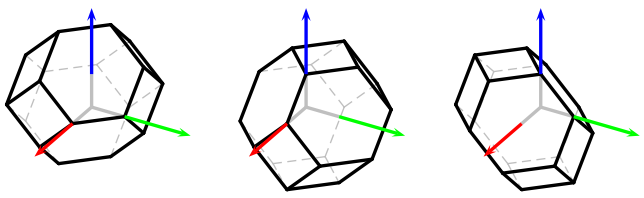
where

$$\begin{aligned} M_{\boldsymbol{\xi}}(\boldsymbol{x}) &:= \nabla_{\boldsymbol{\xi}} 2 M_{\Xi_{\text{fcc}} \backslash \{\boldsymbol{\xi}\}} (\boldsymbol{x} + \boldsymbol{\xi}_c) \\ &= 2 M_{\Xi_{\text{fcc}} \backslash \{\boldsymbol{\xi}\}} (\boldsymbol{x} + \boldsymbol{\xi}_c) - 2 M_{\Xi_{\text{fcc}} \backslash \{\boldsymbol{\xi}\}} (\boldsymbol{x} + \boldsymbol{\xi}_c - \boldsymbol{\xi}). \end{aligned}$$

Since Ξ_{fcc} includes no axis-aligned directions, we compute the three directional derivatives along $\boldsymbol{\xi}_{110}$, $\boldsymbol{\xi}_{101}$, and $\boldsymbol{\xi}_{011}$ then obtain the gradient as follows:

$$\nabla \left(V * M_{\text{fcc}} \right) = \frac{1}{2} \begin{bmatrix} d_{110} + d_{101} - d_{011} \\ d_{110} + d_{011} - d_{101} \\ d_{101} + d_{011} - d_{110} \end{bmatrix},$$

where $d_{\boldsymbol{\xi}} := V * M_{\boldsymbol{\xi}}$. Since each derivative kernel $M_{\boldsymbol{\xi}}$ is the sum of two box-splines, each can be computed in a similar way as M_{fcc} using the de Casteljau algorithm. Those derivative kernels are quadratic and can therefore be evaluated quickly. Moreover, thanks to permutational symmetries among them (Figure 6), we need the BB-coefficients for M_{110} only. But to evaluate M_{110}, due to its lower symmetry compared to that of M_{fcc}, we need to consider three types of polynomial pieces: τ_0, τ_2 and τ_4. Table 1 shows the stencils and BB-coefficients of M_{110} for each type, computed using the method developed by Kim and Peters [12].



(a) (b) (c)

Figure 6: Supports of (a) $M_{\boldsymbol{\xi}_{110}}$, (b) $M_{\boldsymbol{\xi}_{101}}$, and (c) $M_{\boldsymbol{\xi}_{011}}$.

Notice that, from Figure 6(a), we can see that the support of M_{110}, and hence its stencil, is a subset of that of M_{fcc}. Therefore, we do not need additional texture fetches to evaluate the gradient, but can re-use those samples that were already fetched for spline evaluation, resulting in a better performance than the finite difference method, which requires six additional spline evaluations (Table 3).

Table 1: The stencil (leftmost column) and BB-coefficients of M_{110} for (top) τ_0, (middle) τ_2 and (bottom) τ_4.

	2	1	0	1	0	0	1	0	0	0
	0	1	2	0	1	0	0	1	0	0
	0	0	0	1	1	2	0	0	1	0
	0	0	0	0	0	0	1	1	1	2
(0, 0, 0)		−4	−4	−4	−8	−4		−4	−4	
(0, 1, 1)	1			2			2		4	4
(1, 0, 1)	1	2					2	4		4
(1, 1, 0)	2	4	4	4	8	4	2	4	4	
(0, 1,-1)	1			2						
(1, 0,-1)	1	2								
(2, 0, 0)			4							
(0, 2, 0)						4				
(-1,-1, 0)	−2						−2			
(-1, 0, 1)	−1			−2			−2		−4	−4
(0,-1, 1)	−1	−2					−2	−4		−4
(-1, 0,-1)	−1			−2						
(0,-1,-1)	−1	−2								
(1,-1, 0)			−4							
(-1, 1, 0)						−4				
(0, 1, 1)						4				−4
(1, 0, 1)				4	4	4	−4	−4		−4
(1, 1, 0)	1	2	4	2	4		2	4	4	
(2, 0, 0)	1	2	4							
(2, 1, 1)	2	2					4	4		4
(1, 2, 1)										4
(1, 1, 2)	1			2			2		4	
(2, 0, 2)	1									
(-1, 0, 1)						−4				
(0,-1, 1)	−2	−2		−4	−4	−4				
(0, 0, 0)	−1	−2	−4	−2	−4		−2	−4	−4	
(1,-1, 0)	−1	−2	−4							
(0, 0, 2)	−1			−2			−2		−4	
(1,-1, 2)	−1									
(0, 1, 1)	−4	−4						4		4
(1, 0, 1)	−4			−4					4	4
(1, 1, 0)		4	4	4	8	4	4	4	4	
(1, 1, 2)							4			
(1, 2, 1)	4	4								
(2, 1, 1)	4			4						
(0, 2, 0)			4							
(2, 0, 0)						4				
(-1, 0, 1)								−4		−4
(0,-1, 1)									−4	−4
(0, 0, 0)		−4	−4	−4	−8	−4	−4	−4	−4	
(0, 0, 2)							−4			
(-1, 1, 0)			−4							
(1,-1, 0)						−4				

4.3. Empty Space Skipping

While the spline should be evaluated along the whole ray for direct volume rendering, we only need to find the nearest isosurface point along the ray for isosurface rendering. Since spline evaluation is an expensive operation, we can reduce the large overhead by employing an efficient empty space skipping algorithm based on the min/max values of each voxel [9]. This can be achieved efficiently by taking the min/max BB-coefficients in each cube containing five tetrahedral polynomial pieces.

Specifically, during the pre-processing stage for each cubic cell $\{\boldsymbol{j} + [0,1)^3 : \boldsymbol{j} \in \mathbb{Z}^3\}$ we construct the BB-forms of its five tetrahedral polynomial pieces. The min/-

max values of those coefficients are set as the min/max value of the cubic cell. Since this process requires a large overhead but can be easily parallelizable, we implemented a GPU OpenCL kernel that builds the BB-forms of leaf cells. Table 4 shows the performance gain as a result of this approach. The whole min/max octree can then be constructed from the min/max values of the cubic cells. During the rendering stage, in order to compute the entry and exit points for isosurface rendering, instead of rendering one cube enclosing the whole volume, we render the cubes corresponding to one level of the min/max octree. All the vertices of a cube have the same 2D texture coordinates containing the min/max values. To compute the exit positions, we first render the cubes with `glCullFace(GL_FRONT)` and `glDepthFunc(GL_LESS)`. Next we render the cubes again with `glCullFace(GL_BACK)` and `glDepthFunc(GL_GREATER)` to compute the entry positions. In both cases, a fragment shader is loaded, which discards the fragment if the cube is empty or it does not contain the isosurface level (Figure 9).

4.4. Indexing Scheme for FCC Datasets

The volume dataset needs to be stored in the memory of the graphics hardware, usually as a 3D texture. But currently there is no specific hardware that supports non-Cartesian volume datasets. The simplest way to circumvent this is to store an FCC dataset as embedded in a one-channel 3D texture. In this way, indexing induces no overhead, hence the performance is good, but it results in twice the memory footprint. Shifting each layer on the xy-plane in a similar way as Finkbeiner et al. [8] is not a good idea since each layer of $\mathbb{Z}_{\text{fcc}}$ is a 2D Cartesian lattice rotated by $\pi/4$. Loading in four separate textures is not a good idea either, due to the large overhead for data fetching. We propose an efficient way to store an FCC dataset as a four-channel (RGBA) 3D texture.

By grouping the FCC lattice points as

$$\mathbb{Z}_{\text{fcc}} = \{\boldsymbol{j}, \boldsymbol{j}+\boldsymbol{\xi}_{110}, \boldsymbol{j}+\boldsymbol{\xi}_{101}, \boldsymbol{j}+\boldsymbol{\xi}_{011} : \boldsymbol{j} \in 2\mathbb{Z}\},$$

we can convert an FCC dataset, embedded in the Cartesian lattice of size $N_x \times N_y \times N_z$, into a four-channel Cartesian dataset of size

$$\lfloor (N_x+1)/2 \rfloor \times \lfloor (N_y+1)/2 \rfloor \times \lfloor (N_z+1)/2 \rfloor \times 4.$$

For an FCC index $\boldsymbol{j} \in \mathbb{Z}_{\text{fcc}}$, let

$$\boldsymbol{p_j} := (\boldsymbol{j}(1) \bmod 2, \boldsymbol{j}(2) \bmod 2, \boldsymbol{j}(3) \bmod 2)$$

be the *parity vector* of $\boldsymbol{j}$. Since $\boldsymbol{j} \in \mathbb{Z}_{\text{fcc}}$, $\boldsymbol{j}(1)+\boldsymbol{j}(2)+\boldsymbol{j}(3)$ is always even, therefore

$$\boldsymbol{p_j} \in \{\boldsymbol{0}, \boldsymbol{\xi}_{110}, \boldsymbol{\xi}_{101}, \boldsymbol{\xi}_{011}\}$$

and

$$2\boldsymbol{p_j}(1) + \boldsymbol{p_j}(2) \in \{0,1,2,3\}.$$

Thus, we can convert the FCC index $\boldsymbol{j}$ as follows in order to access the four-channel 3D texture:

$$\boldsymbol{j} \mapsto (\lfloor \boldsymbol{j}(1)/2 \rfloor, \lfloor \boldsymbol{j}(2)/2 \rfloor, \lfloor \boldsymbol{j}(3)/2 \rfloor, 2\boldsymbol{p_j}(1) + \boldsymbol{p_j}(2)).$$

Note that the above mapping can be achieved very efficiently using bitwise operations that are supported for GLSL version 1.30 and later, as shown below.

```
texelFetch(texid,j.x>>1,j.y>>1,j.z>>1)[((j.x&1)<<1)|(j.y&1)]
```

Note that a BCC dataset can also be converted to a two-channel 3D texture in a similar manner and accessed as shown below.

```
texelFetch(texid,j.x>>1,j.y>>1,j.z>>1)[x&1]
```

5. Results and Discussion

Since there is no FCC dataset obtained directly from real subjects, we discard half of the original Cartesian dataset provided by the "vollib library" [19] to obtain FCC datasets for the raycasting module. The dataset in Table 7 are obtained by resampling the original dataset interpolated by tricubic B-spline using the MATLAB [16] `interp3` function. Table 2 shows the four datasets used for the experiments. All of the rendering performance measurements were obtained for the images in Figure 7.

Table 2: Test datasets [19].

dataset	size	cell size (10^{-3})	stepsize
`MRI-Head`	128×128×128×4	$3.91 \times 3.91 \times 3.13$	0.003125
`VisMale`	64×128×128×4	$6.16 \times 3.89 \times 3.94$	0.003890
`CT-Head`	128×128× 57×4	$3.71 \times 3.91 \times 7.37$	0.003711
`Carp`	128×128×256×4	$1.53 \times 0.76 \times 1.95$	0.000763

5.1. Image Quality

For raycasting isosurface rendering, the performance and image quality heavily depend on both the stepsize of the marching ray and the window size. For the test, we adopted a stepsize equal to the width of the cubic cell of the dataset and the window size was set to 500×500. Figure 7 shows the resulting images from four FCC datasets. Thanks to the analytic gradient computation, we can obtain isosurface images of superior quality (§5.2). But, since the stepsize is fixed, the image qualities tend to decrease when zoomed-in. This can be overcome by using an adaptive stepsize depending on the viewing angle and the distance from the camera.

5.2. Analytic Normal Computation

Figure 8 shows the rendering results where normal vectors are computed using the analytic formula and the finite difference method. When computed using the finite difference method, the normals are not accurate if we use offset (δ) values that are too small due to the rounding-off error (Figure 8(b)). Offset values that are too large also result in inaccurate normal values (Figure 8(d)) due to poor approximation. When a proper offset value is used (Figure 8(c)), it results in accurate normal values. But the analytic formula always results in accurate normal values. Moreover, as can be seen in Table 3, the computation overhead of the analytic formula is lower since the data fetch overhead is lower.

Table 3: Performance (in fps: frames per second) comparison of two normal computation methods.

dataset	analytic	finite difference	speed-up
MRI-Head	65.2	51.7	1.3
VisMale	68.9	55.1	1.3
CT-Head	54.6	45.1	1.2
Carp	22.3	22.2	1.0

5.3. GPU Pre-processing

Table 4 shows the performance improvement of two pre-processing stages, quasi-interpolation prefiltering and min/max cell construction, using OpenCL kernels. For quasi-interpolation, which is relatively light-weight, the performance improvement is up to $\approx$ 19-fold. For min/-max cell construction, the performance gain is up to $\approx$ 26-fold. In any case, the gain is roughly proportional to the size of the dataset.

Table 4: Computation time (in seconds) for pre-processing using CPU and GPU.

dataset	quasi-interpolation			min/max construction		
	CPU	GPU	speed-up	CPU	GPU	speed-up
MRI-Head	2.412	0.140	17.2	95.542	3.555	26.9
VisMale	1.204	0.096	12.6	47.598	1.777	26.8
CT-Head	0.958	0.089	10.8	41.320	1.723	24.0
Carp	4.654	0.251	18.6	185.502	7.146	26.0

5.4. Empty Space Skipping

Table 5: Performance (in fps) comparison according to the number of min/max cells.

dataset	not used	8^3	16^3	32^3	64^3	speed-up
MRI-Head	27.2	43.4	57.9	65.2	31.5	≤ 2.40
VisMale	30.0	60.0	66.6	68.9	40.0	≤ 2.30
CT-Head	14.9	31.5	43.0	54.6	35.3	≤ 3.66
Carp	4.5	11.5	17.6	22.3	22.1	≤ 4.96

Our raycasting kernel reduces the computational overhead by using fine-grained min/max cells for efficient empty space skipping. Figure 9 illustrates the computational overhead with and without empty space skipping and Table 5 compares the performance. The performance gain is up to $\approx$ 4.95-fold. Again, the performance gain is relatively proportional to the size of the dataset. For optimal performance we need to consider the trade-off according to the number of min/max cells. A large number of small cells induces a tight bound of the isosurface, but comes with a large overhead due to the primitives. On the other hand, a small number of large cells reduces the overhead but bounds the isosurface loosely. In the experiments, 32^3 number of cells always resulted in the best performance (Table 5).

5.5. FCC Dataset Indexing Scheme

Table 6: Performance (in fps) comparison of two indexing schemes.

dataset	4-channel	1-channel	speed-up
MRI-Head	33.6	65.2	1.9
VisMale	31.5	68.9	2.2
CT-Head	25.3	54.6	2.2
Carp	7.3	22.3	3.1

Since the suggested four-channel 3D texture compactly stores the FCC dataset, it is expected to reduce cache misses, but it also induces computational overhead due to index mapping, while it is composed of bitwise operations. But there is a more severe problem in adapting this scheme. Since current graphics hardware does not allow selective fetching of one texture channel, we have to fetch all four channels and select one using a dot product operation, which results in massive performance degradation. In other words, instead of

```
% texelFetch(tex,x,y,z)[idx]
  texelFetch(tex,x,y,z)[idx]
```

we have to make the following call:

```
  dot(texelFetch(tex,x,y,z),vec4(idx==0,idx==1,idx==2,↩
      idx==3))
```

Table 6 shows the performance comparison with respect to the indexing schemes. In all cases, the one-channel scheme required twice the memory footprint as the four-channel scheme. Note that all of the previous performance measurements were taken using a one-channel texture to obtain the best performance.

5.6. Quasi-Interpolation Prefilter

Figure 10 compares the reconstruction quality with and without quasi-interpolation prefilter applied. As can be seen, when the prefilter was not applied, the 'valleys' of the rims were shallower due to the oversmoothing aliasing.

5.7. Comparison with Other Reconstruction Schemes

Table 7: Configurations and performance comparison for Figure 11.

	size	speed (fps)
(b)	$80 \times 80 \times 160 = 1,024,000$	61.14
(c)	$64 \times 64 \times 128 \times 2 = 1,048,576$	10.66
(d)	$50 \times 50 \times 101 \times 4 = 1,010,000$	25.11

Figure 11 compares the quality of three reconstruction schemes with comparable dataset sizes (Table 7). For the Cartesian reconstruction, the tri-cubic B-spline filter was used and its GPU implementation was as described by Sigg and Hadwiger [20]. For the BCC reconstruction, the eight-direction box-spline was used and its GPU implementation was as described by Finkbeiner et al. [8]. They are the fastest GPU raycasters on each lattice so far. Note that they have higher approximation order than our scheme and therefore require more data for evaluation in general. For a fair comparison, no min/max octree was used and the normal computation was done entirely by the finite difference method. Also, the BCC and FCC datasets were stored in unpacked form to reduce the data fetching overhead. Therefore, while the sizes of the BCC and FCC datasets that were actually used for the evaluations were as shown in Table 7, the sizes of the 3D texture are quadruple and twice as the Cartesian scheme, respectively.

When reconsructed on each lattice with the corresponding quasi-interpolation prefilter applied, reconstruction on the BCC lattice using the eight-direction box-spline showed the lowest aliasing, and reconsruction on the Cartesian lattice using the tri-cubic B-spline showed the highest aliasing. Table 7 compares the performance of the three schemes. While the Cartesian reconstruction scheme requires many data samples ($4^3 = 64$) for evaluation, Sigg and Hadwiger [20] proposed a novel approach that requires only 16 data fetches, and hence results in superb performance. Comparing the BCC and FCC schemes, the FCC scheme showed better performance due to the small number of data fetches and lower degree, and hence lower complexity, of the spline formula. In summary, the method can be considered a practical reconstruction scheme that compromises between high quality and performance.

6. Conclusion and Future Work

We implemented a real-time GPU raycaster for FCC datasets. Superior performance was achieved by GPU pre-processing, efficient evaluation of the spline value and its gradient, and efficient empty space skipping. A novel indexing scheme for FCC datasets was proposed, but it showed a poor performance in current hardware due to the limitation of conditional channel selection.

In future work, we plan to improve the raycaster with an adaptive stepsize. Also, it may be possible to find the exact root for isosurface values using the closed-form of cubic polynomial roots. While our implementation adopts gradients that exactly match the underlying spline, it has been shown that this strategy is not necessary for visualization and we can improve rendering quality using different gradient estimation schemes [10]. Finding such a scheme is another future work. Finally, we consider developing an efficient direct volume raycaster.

7. Acknowledgments

The author deeply appreciates professor Jörg Peters and professor Alireza Entezari for their valuable comments on this work. This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0024007).

References

[1] John Horton Conway and Neil J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag New York, Inc., New York, NY, USA, 3rd edition, 1998.

[2] Balázs Csébfalvi and Markus Hadwiger. Prefiltered B-spline reconstruction for hardware-accelerated rendering of optimally sampled volumetric data. In *Workshop Vision, Modeling, and Visualization*, pages 325–332, 2006.

[3] Carl de Boor, Klaus Höllig, and Sherman Riemenschneider. *Box splines*. Springer-Verlag New York, Inc., 1993.

[4] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. A K Peters, Ltd., July 2006. ISBN 1568812663.

[5] Alireza Entezari. *Optimal Sampling Lattices and Trivariate Box Splines*. PhD thesis, Simon Fraser University, 2007.

[6] Alireza Entezari, Ramsay Dyer, and Torsten Möller. Linear and cubic box splines for the body centered cubic lattice. In *Proceedings of the IEEE Conference on Visualization*, pages 11–18. IEEE Computer Society, 2004.

[7] Alireza Entezari, Dimitri Van De Ville, and Torsten Möller. Practical box splines for reconstruction on the body centered cubic lattice. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):313–328, March 2008.

[8] Bernhard Finkbeiner, Alireza Entezari, Dimitri Van De Ville, and Torsten Möller. Efficient volume rendering on the body centered cubic lattice using box splines. *Computers & Graphics*, 34(4):409–423, August 2010.

[9] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24 (3):303–312, September 2005. ISSN 1467-8659.

[10] Zahid Hossain, Usman Alim, and Torsten Möller. Towards high quality gradient estimation on regular lattices. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):426–439, April 2011.

[11] Luis Ibáñez, Chafiaâ Hamitouche, and Christian Roux. Ray-tracing and 3-D objects representation in the BCC and FCC grids. *Lecture Notes in Computer Science*, 1347:235–241, 1997.

[12] Minho Kim and Jörg Peters. Fast and stable evaluation of box-splines via the Bernstein-Bézier form. *Numerical Algorithms*, 50(4):381–399, April 2009.

[13] Minho Kim and Jörg Peters. Symmetric box-splines on root lattices. *Journal of Computational and Applied Mathematics*, 235(14):3972–3989, May 2011.

[14] Minho Kim, Alireza Entezari, and Jörg Peters. Box spline reconstruction on the face-centered cubic lattice. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1523–1530, November-December 2008.

[15] Hans R. Künsch, Erik Agrell, and Fred A. Hamprecht. Optimal lattices for sampling. *IEEE Transactions on Information Theory*, 51(2):634–647, 2005.

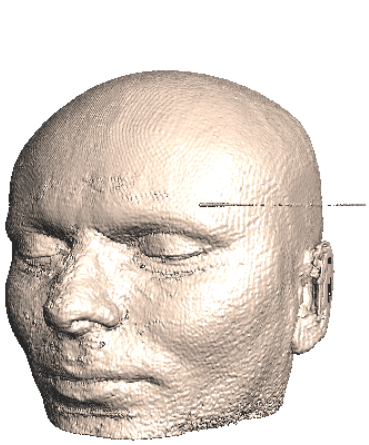
[16] MATLAB. *version 7.8.0 (R2009a)*. The MathWorks Inc., Natick, Massachusetts, 2012.

[17] Kaloian Petkov, Feng Qiu, Zhe Fan, Arie E. Kaufman, and Klaus Mueller. Efficient LBM visual simulation on face-centered cubic lattices. *IEEE Transactions on Visualization and Computer Graphics*, 15:802–814, September 2009. ISSN 1077-2626.

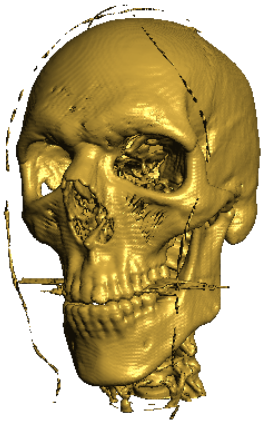
[18] Feng Qiu, Fang Xu, Zhe Fan, Neophytou Neophytos, Arie Kaufman, and Klaus Mueller. Lattice-based volumetric global illumination. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1576–1583, 2007.

[19] Stefan Roettger. Volume library (online), January 2012. URL `http://www9.informatik.uni-erlangen.de/External/vollib`.

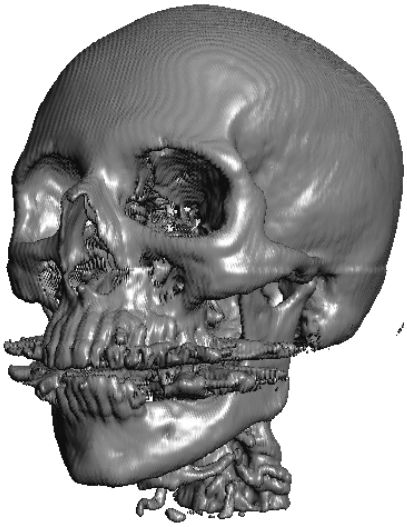
[20] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. In *GPU Gems 2*, chapter 20, pages 313–329. Addison-Wesley, 2005.



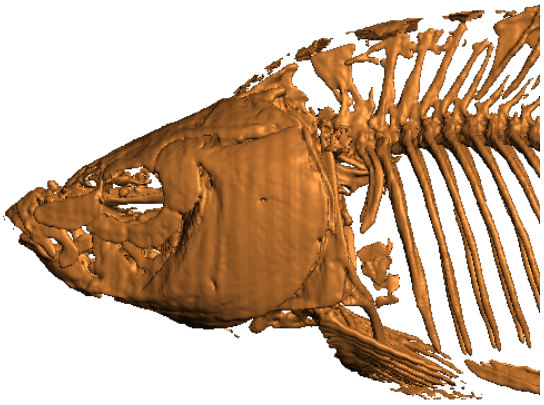
(a) `MRI-Head`



(b) `VisMale`

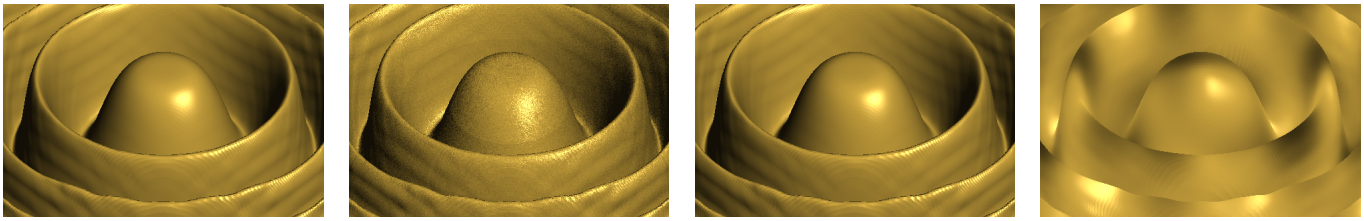


(c) `CT-Head`



(d) `Carp`

Figure 7: Rendered images. Experimental system: Intel® Core™ i7 860 @2.80 GHz, Windows 7 Professional (64 bit), 8GB memory, NVIDIA GeForce GTX 460 (driver 285.86).



(a) (b) (c) (d)

Figure 8: Rendering results where normals are calculated by (a) the analytic formula and finite difference methods with (b) $\delta = 10^{-7}$, (c) $\delta = 10^{-3}$, and (d) $\delta = 10^{-1}$.

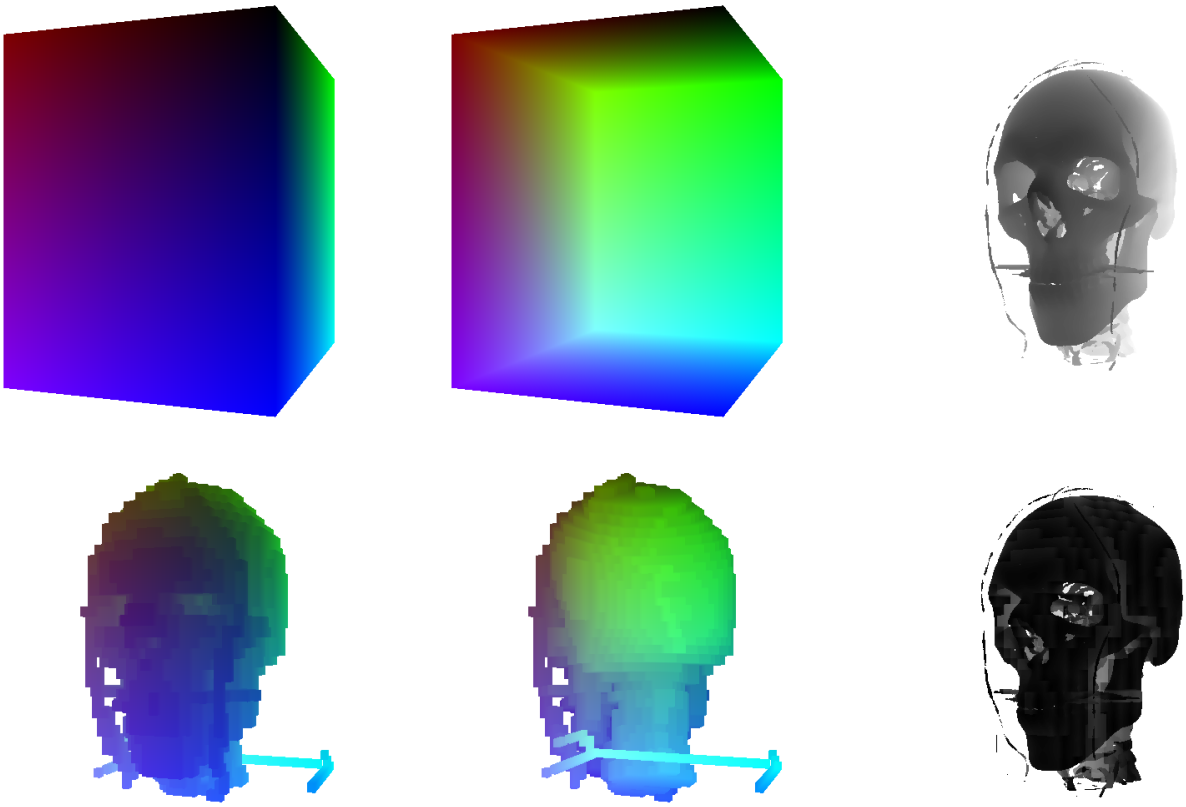


Figure 9: Visualization of the computation overhead for the `VisMale` dataset. (top) Without and (bottom) with the 32^3 min/max cubes. (left) Entry and (center) exit points are normalized to $[0,1]^3$ and color-coded. (right) The number of evaluations for each ray before hitting the isosurface is normalized to grayscale color.

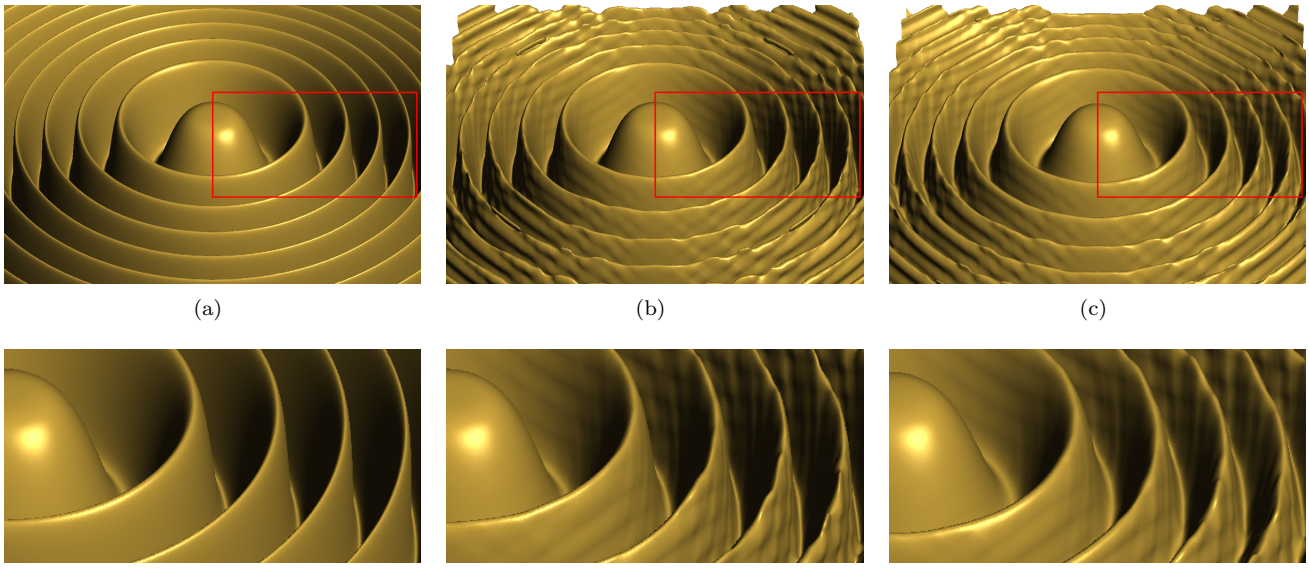
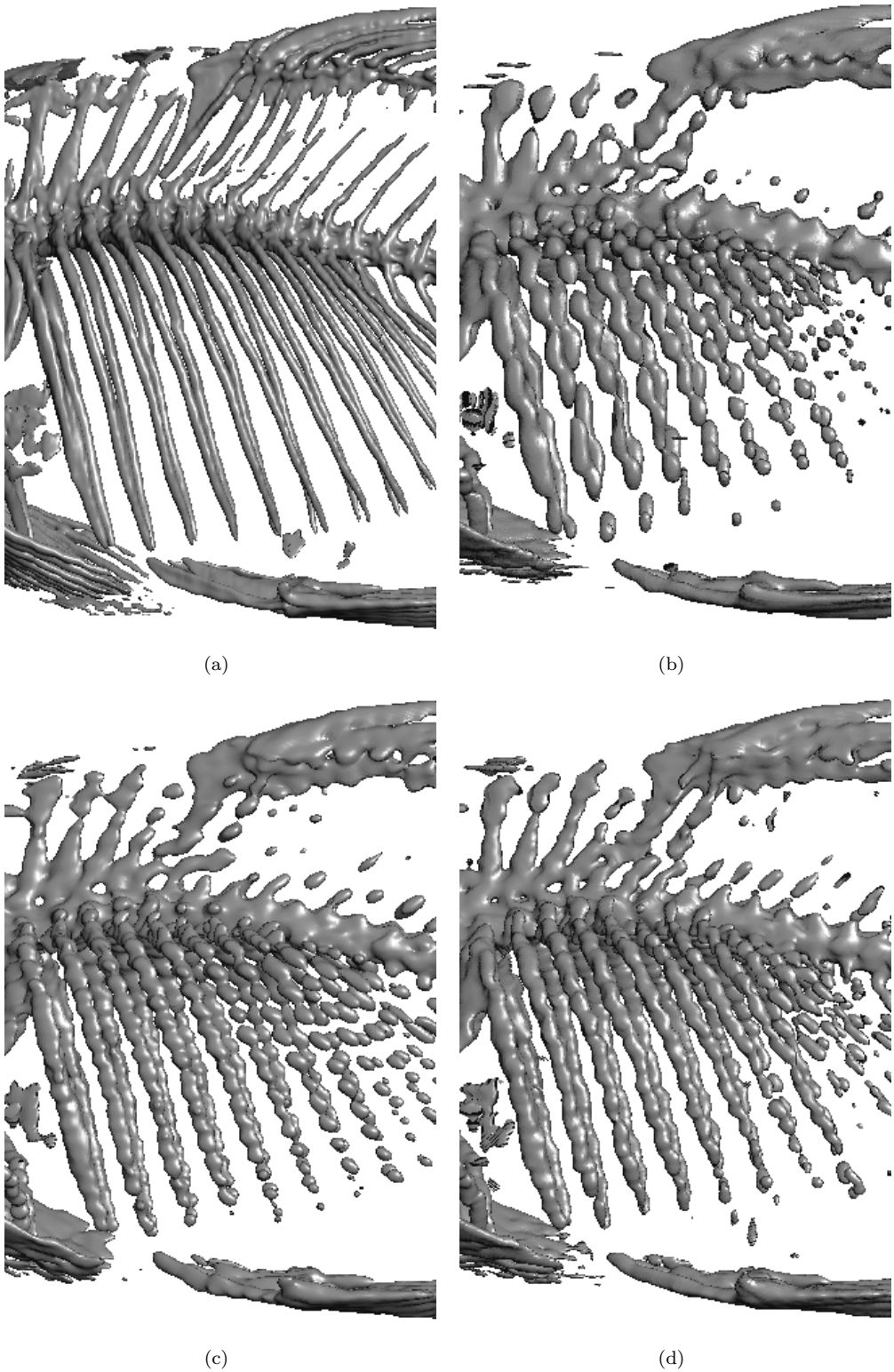


Figure 10: (Left) Original, (middle) with and (right) without a quasi-interpolation prefilter. The "valleys" in (f) are shallower than those in (e).



(a) (b)

(c) (d)

Figure 11: Reconstruction of `Carp` datasets. (a) M_{cc} on the original dataset ($256 \times 256 \times 512$), (b) M_{cc} on the Cartesian lattice, (c) M_{bcc} on the BCC lattice, and (d) M_{fcc} on the FCC lattice. For each dataset, the corresponding quasi-interpolation prefilter was applied.