FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Variational Reconstruction and GPU Ray-Casting of Non-Uniform Point Sets using B-Spline Pyramids

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Martin Kinkelin
Matrikelnummer 0326997

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Prof. Dr. Eduard Gröller
Mitwirkung: Dr. techn. Erald Vuçini

Wien, 03.05.2011 ________________________ ________________________
(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

In this work, we focus on the problem of reconstructing a volume (scalar 3D field) based on non-uniform point samples and then rendering the volume by exploiting the processing power of GPUs.

In the first part involving the reconstruction, we motivate our choice of tensor-product uniform B-splines for the discretized representation of the continuous volume. They allow for highly efficient, scalable and accurate reconstruction at multiple scales (resolution levels) at once. By subdividing the volume into blocks and reconstructing them independently, current desktop PCs are able to reconstruct large volumes and multiple CPU cores can be efficiently exploited. We focus on linear and cubic B-splines and on how to eliminate otherwise resulting block discontinuities.

Once we have reconstructed the volume at multiple scales, we can derive different Levels of Detail (LoDs) by subdividing the volume into blocks and selecting a suitable scale for each block. We present a fusion scheme which guarantees global C^0 continuity for linear LoDs and C^2 continuity for cubic ones. The challenge here is to minimize visual block interscale discontinuities.

A LoD, consisting of a hierarchical spatial subdivision into blocks and an autonomous B-spline coefficient grid for each block, is then rendered via a GPU ray-caster. We achieve interactive frame-rates for qualitative Direct Volume Renderings (DVRs) and real-time frame-rates for iso-surface renderings.

Kurzfassung

Diese Arbeit behandelt die Problemstellung, ein Volumen (skalares 3D-Feld) anhand von unstrukturierten, punktuellen Messwerten zu rekonstruieren und dieses Volumen anschließend mithilfe dedizierter Grafikchips (GPUs) zu visualisieren.

Der erste Schritt besteht in der Rekonstruktion des Volumens. Wir setzen das Tensor-Produkt von uniformen B-splines ein, um eine diskretisierbare Repräsentation des stetigen Volumens zu erhalten. Uniforme B-Splines eignen sich ideal zu diesem Zweck und erlauben eine hocheffiziente, skalierbare und akkurate Rekonstruktion in mehreren Skalen (Auflösungen) auf einmal. Eine Unterteilung des Volumens in Blöcke und die unabhängige Rekonstruktion der Blöcke ermöglichen es modernen Desktop-PCs, große/hochauflösende Volumina zu rekonstruieren und die parallele Architektur moderner Prozessoren auszunutzen. Wir konzentrieren uns auf lineare und kubische B-Splines und wie ansonsten auftretende Diskontinuitäten zwischen den Blöcken vermieden werden können.

Nachdem das Volumen in mehreren Skalen rekonstruiert wurde, können in einem weiteren Schritt verschiedene Detail-Ebenen (Levels-of-Detail, LoDs) davon abgeleitet werden, indem das Volumen wiederum in Blöcke unterteilt und eine geeignete Skala für jeden Block selektiert wird. Wir präsentieren ein Fusionierungsschema, welches eine globale C^0-Stetigkeit für lineare LoDs bzw. C^2-Stetigkeit für kubische LoDs garantiert. Die Herausforderung besteht hierbei darin, visuelle Diskontinuitäten zwischen Blöcken unterschiedlicher Auflösung zu minimieren.

Ein LoD, bestehend in einer hierarchischen räumlichen Unterteilung in Blöcke sowie einem autonomen B-Spline-Koeffizienten-Gitter für jeden Block, wird danach anhand von einem GPU Ray-Caster visualisiert. Wir erzielen interaktive Bildwiederholraten für qualitative Direct-Volume-Renderings (DVRs) und Echtzeit-Raten für entsprechende Isoflächen-Renderings.

Contents

CHAPTER 1

Introduction

In this work, we tackle the problem of visualizing and interactively exploring data given as unstructured scalar point cloud. The points are assumed to be samples of a continuous volume (scalar 3D field) and may be obtained by measurements (e.g., CT/MRT scans), simulations, non-affine transformations, data compression/loss etc. Our approach consists of two steps. We

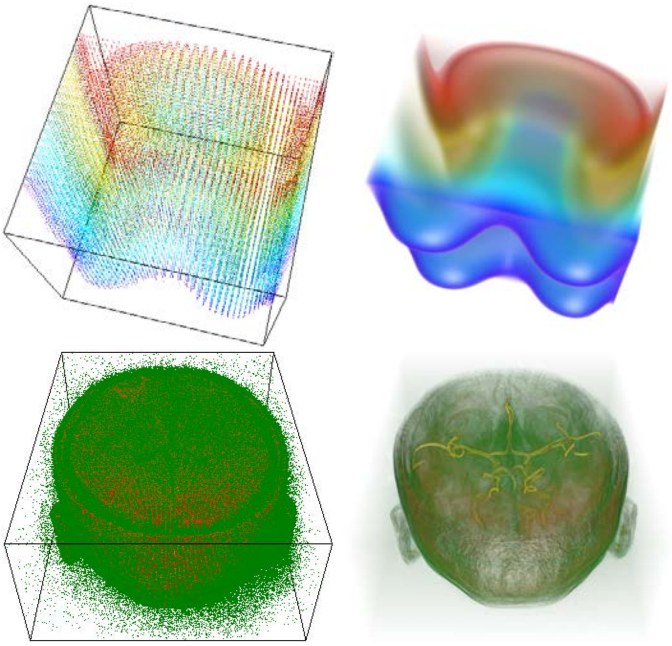


Figure 1.1: Based on unstructured point samples (left), we reconstruct a volume at multiple resolution levels at once and then render derived block-based levels-of-detail (right) on the GPU.

first reconstruct a continuous volume based on the given points, thereby estimating a value for each position in space. The volume can then be qualitatively rendered via ray-casting and enables advanced techniques involving gradient/curvature etc. To achieve the goal of interactive explorability, we exploit dedicated graphics hardware (the GPU) for ray-casting.

In the first part involving the volume reconstruction, we search for a continuous 3D field which connects/approximates the given points. The field needs to be discretizable for machine processing and its discretized representation should allow for efficient evaluation in order to enable efficient GPU ray-casting. Tensor-product uniform B-splines are perfectly suited for this task. We employ a variational approach to find a uniform B-spline discretization of the volume which approximates the point samples (least-squares) while being subject to a smoothness constraint and focus on linear and cubic B-splines. We exploit the interscale relation of B-splines of odd degree to reconstruct the volume efficiently and accurately at multiple scales (resolution levels) at once, yielding a pyramid of uniform B-spline coefficient grids.

In this context, we tackle the problem of reconstructing large volumes requiring huge amounts of RAM. Our approach consists in subdividing the volume adaptively into smaller blocks in a bottom-up fashion. The blocks are padded and independently reconstructed. Afterwards, we apply a fusion scheme which guarantees C^{n-1} continuity at block boundaries for B-splines of odd degree n, even across scales. This block-based reconstruction is ideally suited for parallelization and therefore allows to efficiently exploit multiple CPU cores. It also allows to adapt the resolution of each block and to skip empty blocks altogether.

Once we have a discretized representation of the volume at multiple scales, the most obvious advantage is the ability to derive different Levels-of-Detail (LoDs) by adapting the resolution. This is achieved by subdividing the volume again into adaptive blocks, selecting a matching scale for each block and then fusing the blocks to ensure global C^{n-1} continuity. We use a Bounding Interval Hierarchy (BIH), a simple extension of k-d trees, as flexible hierarchical structure for the blocks and map each block's coefficient grid into a single 3D texture. The GPU ray-caster then traverses the binary tree to identify hit blocks and samples the autonomous block B-splines along the ray. We will see that current GPUs provide enough power to deliver qualitative Direct Volume Renderings (DVRs) at interactive frame-rates and iso-surface renderings at real-time frame-rates.

1.1 Related Work

Visualizing point clouds depends primarily on what the points represent/on how they have been acquired. Points can be visualized directly if they represent geometric primitives such as spherical particles [9]. In this case, one renders a sphere at each given point and uses α-blending for a translucent view. The required visibility ordering can be efficiently implemented by ray-casting on the GPU and accelerating it by a hierarchical subdivision of the points/particles, e.g., via k-d trees, octrees, bounding sphere hierarchies etc.

In case the points represent samples of a surface, such as data acquired by laser scanners, one aims at visualizing an estimate of the original surface. Traditional approaches generate a mesh of polygons such as triangles to interpolate the points (used as vertices) [2] and rasterize the polygons for visualization. More advanced techniques use the given points directly as primitives and derive a fitting smooth surface, e.g., via a moving-least-squares (MLS) approach [1].

If the point cloud represents samples of a continuous field, the goal is to visualize an estimate of the original field. Analogous to surface reconstruction based on point samples, related approaches can be split into two categories. The first category is based on finite element analysis and aims at interpolating between neighboring points [15]. The required neighborhood/connectivity information is often computed by polyhedralizing the point cloud, e.g., by generating a tetrahedral mesh; the new structures can then be rendered via ray-casting [20, 6]. The second category aims at approximating the points by estimating a field composed of basis functions such as RBFs (Radial Basis Functions) [7].

Arigovindan et al. [3] proposed the usage of a tensor-product uniform B-spline basis to reconstruct grey-scale images (scalar 2D fields) based on non-uniform point samples. Vuçini et al. [18] ported the approach to 3D for scalar volume data and compared uniform cubic B-splines to classical approaches based on RBFs and EBFs (Ellipsoidal Basis Functions). They conclude that the uniform cubic B-spline basis provides higher quality (synthetic as well as visual) at a fraction of the required run-time for reconstruction and rendering. Recently, Morozov et al. [14] published a very similar approach, which aims at reducing memory requirements by expressing the linear system in tensor-notation. Based on these findings, we are also going to employ uniform B-splines to reconstruct scalar 3D fields. Contrary to Vuçini et al. [18] and Morozov et al. [14], we will not only focus on cubic B-splines, but also evaluate linear ones, analog to Arigovindan et al. [3] for 2D fields.

Reconstructions based on uniform B-splines result in a regular (Cartesian) grid of coefficients. The implicit connectivity allows for very efficient evaluation and hence good ray-casting performance. The disadvantage is that the resolution is uniform, i.e., the grid density is not adapted to the given point cloud. Vuçini et al. [18] proposed an adaptive top-down subdivision to adapt the resolution in a block-wise fashion to the local data. Each block is represented by a coarse B-spline coefficients grid and the most significant parts of 1-2 finer resolution levels (20% of the error volumes each, run-length encoded and thus requiring 40% of the full grid's size). Evaluating a block's B-spline therefore requires RLE decoding as well as summing up multiple B-splines (coarse volume + error volumes). Additionally, the blocks overlap to eliminate discontinuities at block boundaries, requiring to further sum up multiple (up to 8 in 3D) block B-splines near boundaries. This multi-resolution scheme is therefore not suited for efficient and scalable ray-casting. We are thus going to propose a simpler scheme based on an adaptive bottom-up subdivision and autonomous blocks to achieve interactive visualizations of volumes consisting in blocks of varying resolution levels while ensuring global C^0 and C^2 continuity for linear and cubic B-splines, respectively. The latter point regarding interblock continuity has already been studied for traditional DVR [4]. Most of the proposed approaches are designed for a trilinear interpolation of a given Cartesian grid. To obtain coarse blocks, the original grid region is downsampled. Commonly, a centered sampling pyramid [5] is used for higher downsampling accuracy; the resulting sample alignment complicates the task of smooth/continuous interblock boundaries, so most approaches only focus on C^0 continuity for trilinear filtering. Our proposed fusing scheme simplifies things by relying on aligned sampling pyramids and exploits the fact that coarser scales of multi-scale pyramids are more accurate than downsampled versions of the finest scale, yielding less visual divergence across scales.

Part I

Theory

CHAPTER 2

The Uniform B-Spline

This chapter gives a quick overview over uniform B-splines and introduces some terminology used later on.

2.1 Definition

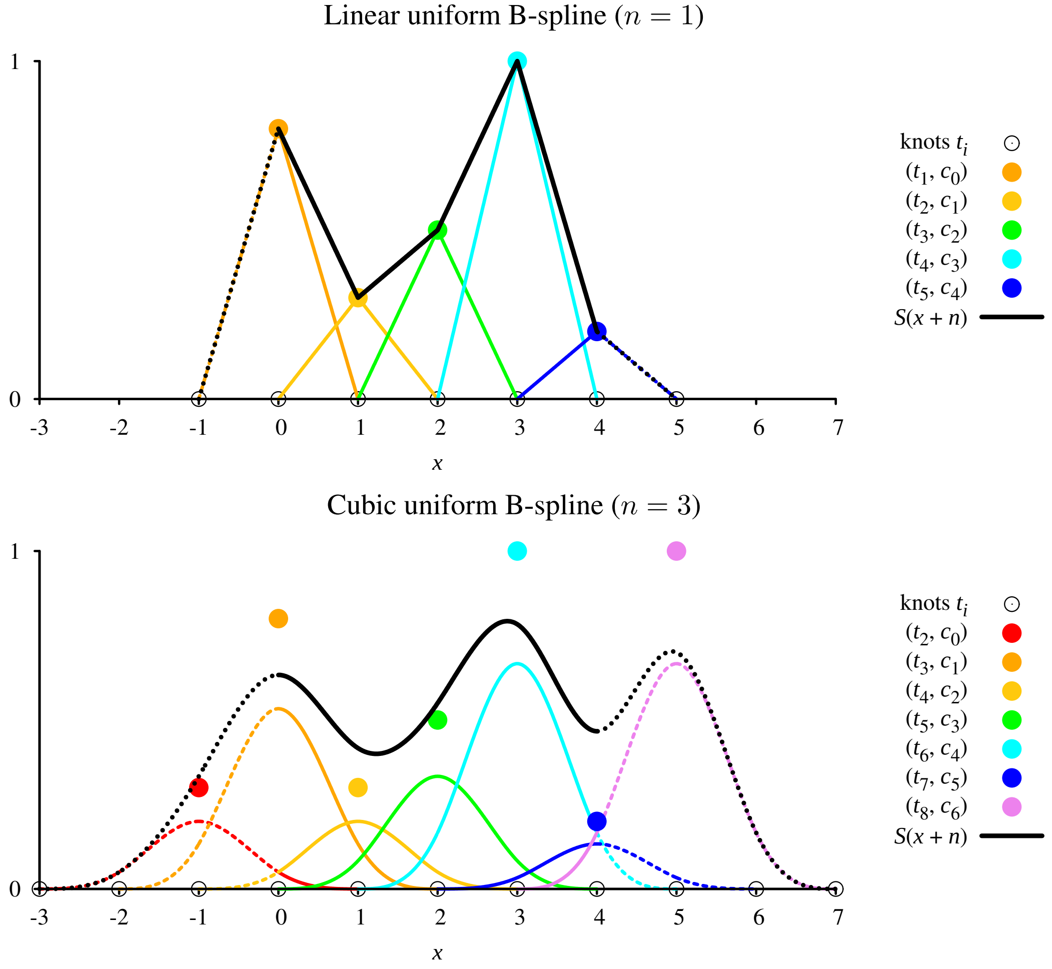
1D

A spline of degree n is a piecewise function composed of polynomials of degree n. The space of piecewise polynomials is a vector space and has a basis. The optimal basis in terms of efficiency and numerical stability, due to its minimal support with respect to a given degree and smoothness, is the family of so-called B-spline basis functions, defined by the Cox-de Boor recursion formula. B-splines are a subclass of splines and defined as linear combination of B-spline basis functions of the corresponding degree.

The shape of the basis B-splines is determined, apart from the degree n obviously, by M so-called knots $\{t_0, \ldots, t_{M-1}\}$ associated with the spline. t_0 represents the start of the parameterized spline, t_{M-1} its end and the other knots represent the joints between the polynomial pieces or, as we are going to call them, spline intervals. A basis B-spline of degree n has a support of $n+1$ intervals, i.e., it is zero everywhere except for $n+1$ intervals. Given the number of knots M, there are $N = M - (n+1)$ basis B-splines, one starting at every knot except for the last $n+1$ knots. The weights of these N basis B-splines are given by N coefficients c_i, which are also called control points (due to B-splines being a generalization of 2D Bezier curves). A B-spline $S : \mathbb{R} \to \mathbb{R}$ is defined as:

$$S(x) = \sum_{i=0}^{N-1} c_i B_i^n(x) \tag{2.1}$$

with support $\text{supp}(S) = [t_0, t_{M-1}] = [t_0, t_{N+n}]$; $B_i^n(x)$ denotes the basis B-spline of degree n with support $\text{supp}(B_i^n) = [t_i, t_{i+(n+1)}]$. Each basis B-spline $B_i^n(x)$ thus depends upon the placement of the local knots $\{t_i, \ldots, t_{i+(n+1)}\}$ only. As each coefficient c_i specifies the weight of a single basis B-spline, it also affects only these $n+1$ intervals.


Linear uniform B-spline ($n = 1$)

Cubic uniform B-spline ($n = 3$)

Figure 2.1: Examples of uniform B-splines for 1D signals. The weight of each compactly supported basis B-spline is given by a corresponding coefficient c_i located at the central knot $t_{i+\frac{n+1}{2}}$ (for odd degree n). The inner signal range of 4 intervals (each affected by $n+1$ basis B-splines and indicated by solid curves) requires $N = 4 + n$ coefficients/basis B-splines and leads to $M = N + (n + 1)$ knots.

Uniform B-splines illustrated in Figure 2.1 are a simplification because the knots are equidistantly placed. This results in all basis B-splines being shifted copies of each other and leads to the uniform B-spline definition

$$S(x) = \sum_{i=0}^{N-1} c_i \beta^n (x - i) \tag{2.2}$$

where $\beta^n(x)$ denotes the uniform basis B-spline of degree n with support $\operatorname{supp}(\beta^n) = [0, n{+}1]$. The parameter x indicates the position along the knots, e.g., $S(0)$ evaluates the B-spline at t_0 and $S(1.5)$ evaluates it halfway between knots t_1 and t_2. We are going to use linear ($n = 1$) and cubic ($n = 3$) uniform B-splines; their basis B-splines depicted in Figure 2.2 are given by:

$$\beta^1(x) = \begin{cases} x & \text{if } x \in [0,1], \\ 2 - x & \text{if } x \in [1,2], \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

and

$$\beta^3(x) = \frac{1}{6} \cdot \begin{cases} x^3 & \text{if } x \in [0,1], \\ -3x^3 + 12x^2 - 12x + 4 & \text{if } x \in [1,2], \\ 3x^3 - 24x^2 + 60x - 44 & \text{if } x \in [2,3], \\ (4 - x)^3 & \text{if } x \in [3,4], \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Uniform basis B-splines of odd degree n have an even support of $n + 1$ intervals, hence their centers lie exactly at a knot. It may be useful to think of a scalar coefficient c_i as point $(t_{i+\frac{n+1}{2}},\ c_i)$ whose y coordinate determines the weight of the uniform basis B-spline centered at its x coordinate (see Figure 2.1).

Figure 2.1 illustrates that the first n and last n intervals of a B-spline are linear combinations of less than the usual $n + 1$ basis B-splines. We therefore speak of the full signal in $[0, N + n]$ and of the inner signal in $[n, N]$. We refer to the $n - 1$ (for odd degree n) coefficients outside the inner signal as outer coefficients. We will focus on the inner signal and use the offset n for the B-spline parameter in $S(x + n)$, so that $x \in [0, N - n]$ denotes the position relative to the inner signal. An appropriate alternative uniform B-spline definition, emphasizing the local support, is

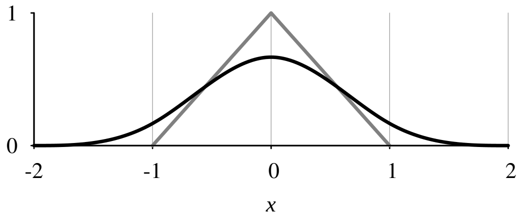


Figure 2.2: Linear ($n = 1$, grey) and cubic ($n = 3$, black) uniform basis B-splines shifted by $\frac{n+1}{2}$ to the left so that they are centered at $x = 0$, emphasizing the symmetry. The $n{+}1$ intervals are defined by different polynomials of degree n.

given by:

$$\begin{aligned} S(x+n) &= \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + n} c_i \beta^n((x+n)-i) \\ &= S^{(\lfloor x \rfloor)}(x - \lfloor x \rfloor) \end{aligned} \tag{2.5}$$

where $\lfloor \cdot \rfloor$ denotes the floor function and

$$S^{(i)}(t) = \sum_{j=0}^{n} c_{i+j} \beta^n(t+n-j) \tag{2.6}$$

evaluates the uniform B-spline in the ith inner signal interval and $t \in [0,1]$ denotes the relative position inside the interval. The special case $x = N - n$ is handled by $S((N-n)+n) = S^{(N-n)}(0) = S^{(N-n-1)}(1)$ to prevent the coefficient index $(i+j)$ from overflowing.

Higher Dimensions

B-splines $S : \mathbb{R}^d \to \mathbb{R}$ with $d > 1$ are commonly defined as tensor-product of d uni-dimensional B-splines, e.g., for a uniform B-spline and $d = 3$:

$$S(x,y,z) = \sum_{m=0}^{N_z-1} \sum_{l=0}^{N_y-1} \sum_{k=0}^{N_x-1} c_{k,l,m} \, \beta^n(x-k)\beta^n(y-l)\beta^n(z-m) \tag{2.7}$$

for $x \in [0, N_x + n], y \in [0, N_y + n], z \in [0, N_z + n]$. The corresponding alternative definition for the inner signal is given by:

$$S(x+n, y+n, z+n) = S^{(\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor)}(x - \lfloor x \rfloor, y - \lfloor y \rfloor, z - \lfloor z \rfloor) \tag{2.8}$$

for $x \in [0, N_x - n], y \in [0, N_y - n], z \in [0, N_z - n]$, with

$$\begin{aligned} S^{(i_x,i_y,i_z)}(t_x,t_y,t_z) = \sum_{j_z=0}^{n} \beta^n(t_z+n-j_z) \sum_{j_y=0}^{n} \beta^n(t_y+n-j_y) \cdot \\ \sum_{j_x=0}^{n} \beta^n(t_x+n-j_x) \, c_{i_x+j_x, i_y+j_y, i_z+j_z} \end{aligned} \tag{2.9}$$

The coefficients $c_{k,l,m}$ form a uniform Cartesian 3D grid of size (N_x, N_y, N_z).

There exist more exotic spline variants for higher dimensions such as Box splines for the Body Centered Cubic lattice [8]. While it is well known that the Cartesian lattice is not the most effective sampling lattice, it is simple and well-studied. The GPU is tailored for Cartesian lattices as well, offering hardware linear interpolation (in up to 3 dimensions). The Cartesian lattice also enables us to formulate a simple scheme for fusing adjacent blocks, as we will see later on. Another advantage of tensor-product uniform B-splines is the simple adaptation for arbitrary dimensions.

2.2 Interscale Relation of B-splines of Odd Degree

An important property of B-splines of odd degree is the so-called interscale relation. The Lane-Riesenfeld algorithm allows to duplicate a 1D B-spline at twice the resolution by inserting new knots at the centers between two existing knots. The coefficients $\mathbf{c}_{j-1}$ of the B-spline S_{j-1} in the finer scale $(j-1)$ can be easily derived from the coarser coefficients $\mathbf{c}_j$ by upsampling (inserting zeros inbetween the existing coefficients) followed by a discrete convolution with the appropriately scaled binomial filter of degree $(n+1)$ (i.e., the $(n+2)$th row of Pascal's triangle).

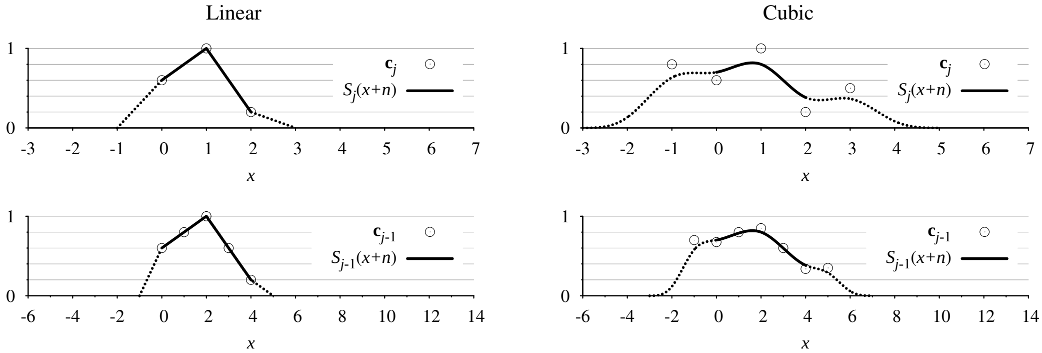


Figure 2.3: Lossless expansion of S_j's inner signal (top) to S_{j-1} (bottom).

Figure 2.3 illustrates the expansion process for linear and cubic uniform 1D B-splines. Let us assume that the ith coarse coefficient $c_{j;\,i}$ is aligned with the fine coefficient $c_{j-1;\,2i}$, i.e., that the coarse coefficient's index i is simply to be doubled to obtain the index of the finer coefficient at the same location. We then yield the following formula for a linear B-spline:

$$c_{j-1;\,i} = \begin{cases} \frac{1}{2} \cdot \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0 & c_{j;\,\frac{i}{2}} & 0 \end{pmatrix}^T & \text{if } i \text{ is even,} \\ \frac{1}{2} \cdot \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} c_{j;\,\frac{i-1}{2}} & 0 & c_{j;\,\frac{i+1}{2}} \end{pmatrix}^T & \text{if } i \text{ is odd,} \end{cases} \tag{2.10}$$

and the following one for a cubic B-spline:

$$c_{j-1;\,i} = \begin{cases} \frac{1}{8} \cdot \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \end{pmatrix} \begin{pmatrix} c_{j;\,\frac{i-2}{2}} & 0 & c_{j;\,\frac{i}{2}} & 0 & c_{j;\,\frac{i+2}{2}} \end{pmatrix}^T & \text{if } i \text{ is even,} \\ \frac{1}{8} \cdot \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \end{pmatrix} \begin{pmatrix} 0 & c_{j;\,\frac{i-1}{2}} & 0 & c_{j;\,\frac{i+1}{2}} & 0 \end{pmatrix}^T & \text{if } i \text{ is odd.} \end{cases} \tag{2.11}$$

The inner signal of S_{j-1} will then be a perfect copy of S_j's inner signal: $S_j(x+n) = S_{j-1}(2x+n)$ for $x \in [0, N_j - n]$. Due to the parameter transformation, a qth order derivative of S_{j-1} needs to be scaled by 2^q to obtain equivalence with S_j's derivative, i.e.,

$$\frac{\partial^q S_j}{\partial x^q}(x+n) = 2^q \cdot \frac{\partial^q S_{j-1}}{\partial (2x)^q}(2x+n) \tag{2.12}$$

for $x \in [0, N_j - n]$.

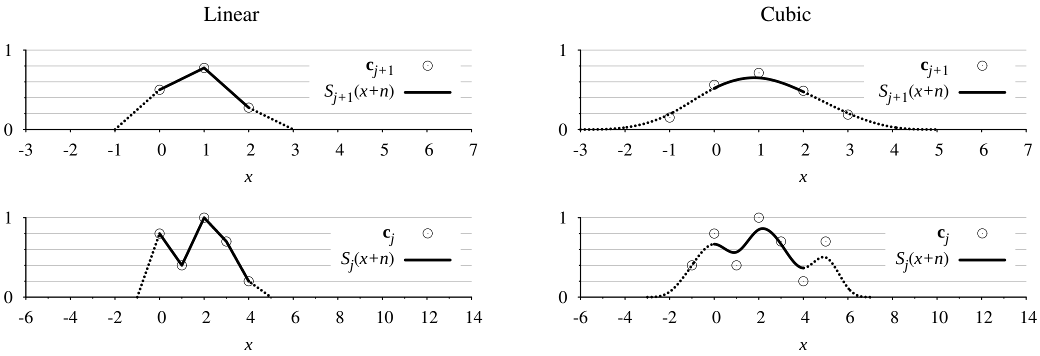


Figure 2.4: Lossy coarsening of S_j's inner signal (bottom) to S_{j+1} (top).

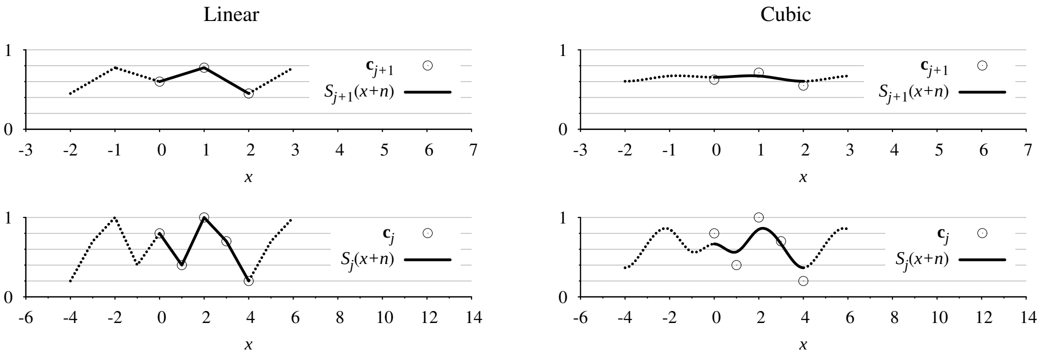


Figure 2.5: Lossy coarsening of S_j's periodic signal (bottom) to S_{j+1} (top).

In higher dimensions, the filter kernel for a tensor-product B-spline is the tensor-product of the 1D kernels. The linear separability can therefore be exploited by first convolving all rows horizontally, then the resulting rows vertically and finally the resulting slices along the z axis (the order of the axes is interchangeable).

The adjoint coarsening operation illustrated in Figure 2.4 consists in performing the convolution first, followed by a downsampling operation (discarding every other coefficient). The operation is obviously lossy and results in a low-pass filtered coarser signal. The convolution filter is the binomial one used for the expansion scaled by $\frac{1}{2}$ to obtain an overall weight of 1, as becomes evident in the following formula:

$$c_{j+1;\,i} = \begin{cases} \frac{1}{4} \cdot \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} c_{j;\,2i-1} \\ c_{j;\,2i} \\ c_{j;\,2i+1} \end{pmatrix} & \text{for } n = 1, \\ \frac{1}{16} \cdot \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \end{pmatrix} \begin{pmatrix} c_{j;\,2i-2} \\ c_{j;\,2i-1} \\ c_{j;\,2i} \\ c_{j;\,2i+1} \\ c_{j;\,2i+2} \end{pmatrix} & \text{for } n = 3. \end{cases} \tag{2.13}$$

When coarsening a signal, some fine coefficients inside the convolution window are unavailable for some of the first and last coarse coefficients. A common choice is to treat all missing fine coefficients as zeros (natural boundary conditions) as illustrated in Figure 2.4. Mirroring the coefficients is also a common choice, leading to a periodic signal. The left and right mirroring axes then need to be positioned at the beginning and the end of the inner signal so that the axes are not shifted across scales. The coefficients are therefore restricted to inner coefficients only, outer ones are obtained implicitly by mirroring. The coarsening operation for a periodic signal is illustrated in Figure 2.5 (using the same fine coefficients as in Figure 2.4 for a direct comparison).

This is just one of the simplest and most common schemes for expanding and coarsening B-splines. There exist other methods which try to minimize the error between a vector and the coarsened version of its expansion, requiring very wide convolution windows. In that respect, the centered pyramids approach [5] provides better accuracy than an aligned pyramid by placing the two fine coefficients at $\frac{1}{4}$ and $\frac{3}{4}$ between two coarse ones. Later on, we will see that the proposed scheme fits our purposes just fine.

CHAPTER 3

Variational Reconstruction

In this chapter, we give a formal overview of the variational reconstruction process. We start with a set of points in 3D, each associated with a scalar attribute: $\{(\mathbf{x}_i, f_i)\}, \mathbf{x}_i \in \mathbb{R}^3, f_i \in \mathbb{R}$. These samples may represent material densities (e.g., obtained by a CT scan), temperatures, pressures and so on, at some points in space. Our goal then is to construct a corresponding continuously defined field $S : \mathbb{R}^3 \to \mathbb{R}$ which maps every point inside a volume (e.g., the bounding box of the points) to a scalar value, filling all the gaps inbetween the samples.

3.1 Problem Formulations

This section covers the two fundamental problem formulations of the reconstruction process.

Interpolation Problem

The basic interpolation problem consists in constructing a continuously defined function S, such that $S(\mathbf{x}_i) = f_i$. This problem is ill-posed because there are no constraints for S at all other points $\notin \{\mathbf{x}_i\}$, apart from S being continuous. Hence additional constraints are needed to make the solution unique. In the variational approach, one aims at enforcing some smoothness on S by minimizing a squared semi-norm used as regularization term. We are going to use Duchon's semi-norms [7]. A detailed analysis thereof, touching also the related Lagrange's semi-norms, is given by Kybic et al. [12, 13].

Duchon's Semi-Norms and Corresponding Regularization Terms

Duchon's semi-norm of order p is defined as

$$\|S\|_{D^p} = \sqrt{\int_{\mathbb{R}^3} \|D^p S\|^2 \,\mathrm{d}\mathbf{x}} \tag{3.1}$$

where $\|\cdot\|$ denotes the Euclidean norm and D^p is a differential operator yielding the vector of all possible partial derivatives of order p of some function.

The corresponding regularization term is simply the square of the semi-norm:

$$\mathcal{M}_p(S) = \|S\|^2_{D^p} = \int_{\mathbb{R}^3} \|D^p S\|^2 \,\mathrm{d}\mathbf{x}. \tag{3.2}$$

In other words, $\mathcal{M}_p(S)$ is the integral of the sum of all squared partial derivatives of order p of S.

For the gradient ∇S, i.e., the vector of all first order partial derivatives, we obtain

$$D^1 S = \nabla S = \left(\frac{\partial S}{\partial x}, \frac{\partial S}{\partial y}, \frac{\partial S}{\partial z}\right)$$

$$\mathcal{M}_1(S) = \int_{\mathbb{R}^3} \|D^1 S\|^2 \,\mathrm{d}\mathbf{x} = \iiint\limits_{\mathbb{R}^3} \left(\frac{\partial S}{\partial x}\right)^2 + \left(\frac{\partial S}{\partial y}\right)^2 + \left(\frac{\partial S}{\partial z}\right)^2 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \tag{3.3}$$

If, more commonly, we are interested in minimizing the squares of the second order partial derivatives, we obtain

$$D^2 S = \left(\frac{\partial^2 S}{\partial x^2}, \frac{\partial^2 S}{\partial x \partial y}, \frac{\partial^2 S}{\partial x \partial z}, \frac{\partial^2 S}{\partial y \partial x}, \frac{\partial^2 S}{\partial y^2}, \frac{\partial^2 S}{\partial y \partial z}, \frac{\partial^2 S}{\partial z \partial x}, \frac{\partial^2 S}{\partial z \partial y}, \frac{\partial^2 S}{\partial z^2}\right)$$

$$\begin{aligned}\mathcal{M}_2(S) = \int_{\mathbb{R}^3} \|D^2 S\|^2 \,\mathrm{d}\mathbf{x} = \iiint\limits_{\mathbb{R}^3} &\left(\frac{\partial^2 S}{\partial x^2}\right)^2 + \left(\frac{\partial^2 S}{\partial y^2}\right)^2 + \left(\frac{\partial^2 S}{\partial z^2}\right)^2 + \\ &2\left(\frac{\partial^2 S}{\partial x \partial y}\right)^2 + 2\left(\frac{\partial^2 S}{\partial y \partial z}\right)^2 + 2\left(\frac{\partial^2 S}{\partial x \partial z}\right)^2 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z.\end{aligned} \tag{3.4}$$

Approximation Problem

Measurements are often noisy or lacking precision, and thus it may not be desirable to interpolate the samples f_i exactly. In that case, the interpolation problem is relaxed to an approximation problem by trading some closeness of fit against more smoothness on the solution. The solution for the approximation problem is the minimizer of the cost function

$$\mathcal{C}_p(S, \lambda_p) = \left(\sum_i [S(\mathbf{x}_i) - f_i]^2\right) + \lambda_p \mathcal{M}_p(S) \tag{3.5}$$

where $\lambda_p > 0$ is a so-called regularization parameter controlling the weight of the regularization term $\mathcal{M}_p(S)$ (Eq. 3.2) and hence the trade-off between closeness of fit and smoothness.

We are going to tackle this problem rather than the interpolation problem because it handles the case in which exact interpolation is not possible due to discretization issues, a frequent problem when translating mathematical findings to algorithms for computers with their finite memory and computational resources. The approximation problem also provides flexibility via the regularization parameter – e.g., in case the samples can be exactly interpolated, we can get back to the interpolation problem by choosing an arbitrarily small λ_p, although it must be nonzero for the problem to remain well-posed.

3.2 Classical Solution

Arigovindan et al. [3, page 2] refer to Duchon as one of the first to establish the exact solution for the approximation problem:

$$S(\mathbf{x}) = \left(\sum_i w_i \, \phi(\|\mathbf{x} - \mathbf{x}_i\|) \right) + P(\mathbf{x}). \tag{3.6}$$

It consists of two terms. The first one is a linear combination of a specific class of Radial Basis Functions (RBFs) called thin-plate splines, positioned at the sample locations $\mathbf{x}_i$. In 3D, the thin-plate spline is given by $\phi(r) = |r|^{2p-3}$, depending upon the order p of the semi-norm. The second term is a polynomial $P(\mathbf{x})$ satisfying $D^p P = \mathbf{0}$ and thus not contributing to the regularization term (Eq. 3.2). A system of linear equations needs to be solved to obtain the weights w_i for the RBFs and the coefficients for the polynomial P. For further details please refer to Arigovindan et al. [3] or directly to Duchon [7].

Due to the infinite support of thin-plate splines, the matrices of the linear system are dense. Additionally, in the common case in which the order p of the semi-norm is greater than 1, the 3D thin-plate spline's magnitude increases with the distance from its center, leading to poorly conditioned matrices. This means that a small change to the samples f_i may have a large impact on the resulting weights. These disadvantages lead to computationally inefficient as well as numerically unstable solutions if the number of samples is large, which is common in 3D. The infinite support also prevents an efficient and scalable evaluation after having computed the solution – evaluating S at an arbitrary location $\mathbf{x}$ requires the evaluation of the thin-plate spline for every sample.

3.3 Relation of the Approximation Problem's Solution to B-Splines

1D Problem

Arigovindan et al. [3, page 3] state that the solution for the approximation problem in 1D is a spline of degree $(2p - 1)$, p still being the order of the semi-norm, with knots at the sample locations x_i. By expressing the spline as B-spline based on compactly supported non-uniform B-spline basis functions, the classical solution's disadvantages mentioned earlier can be avoided – the matrices become sparse and well-conditioned. The well-known corresponding numerical technique is called smoothing splines.

The non-uniform B-spline basis may also be replaced by a uniform one. In that case, to enforce equivalence with the exact solution, the distance between the knots, i.e., the step-size a, needs to be chosen such that every sample location x_i is positioned exactly at a knot. This means that the 1D approximation problem can be discretized exactly by using uniform B-splines of degree $(2p - 1)$ and choosing a sufficiently high resolution. We focus on the semi-norms of order $p = 1$ and especially $p = 2$, leading to linear and cubic uniform B-splines, respectively.

Higher Dimensions

Arigovindan et al. [3, page 3] encourage the use of tensor-product uniform B-splines for higher dimensions, although the approximation problem's solution can only be approximated: "Unfortunately, for dimensions higher than one, there are no compactly supported functions that span the same space as the RBFs. Thus, a uniform B-spline discretization of the problem is not rigorously exact anymore. However, a B-spline basis with a degree that is matched to p remains the best possible choice among all tensor product shift-invariant bases, because it has a high-enough order of differentiation and it is compatible with the optimal one-dimensional solution. The slight discrepancy with the optimal analytical solution that this may generate is largely compensated by the computational advantages (sparse matrices, multiresolution) provided by this type of representation. Additionally, the error can be made arbitrarily small by decreasing the sampling step a. In this last respect, B-splines offer another advantage: for a given support size, they are the refinable functions that result in the smallest discretization error. [...] In other words, they provide the best quality for a given computational cost."

Vuçini et al. [18, page 6] compare their approach based on uniform cubic B-splines to classical approaches based on RBFs [11] and EBFs (Ellipsoidal Basis Functions) [10]. They achieve higher reconstruction quality, both in terms of RMS (root mean square) error measure as well as visual quality. At the same time, the approach based on uniform B-splines outperforms the RBF/EBF-based approaches in terms of required computation time by a factor ranging from 3 to more than 600.

3.4 Variational Reconstruction using Uniform B-Splines

In the previous section, we found that tensor-product uniform B-splines, although not able to solve the approximation problem in 3D exactly, are very well suited to approximate the exact solution. In this section, we search for the optimal solution within the space of uniform B-splines.

Uniform B-Spline Formulation

Given

- the order p of the semi-norm and a matching B-spline degree $n \geq p$ (in our case, $p \in \{1, 2\}$ and $n = 2p - 1$),
- the size (M_x, M_y, M_z), in B-spline intervals, of the volume to be reconstructed,
- the regularization parameter $\lambda_p > 0$ and
- the sample set $\{(x_i, y_i, z_i, f_i)\}$.

Reconstructing a volume of size (M_x, M_y, M_z) as inner signal of a uniform B-spline of degree n requires a uniform 3D grid of B-spline coefficients of size $(N_x, N_y, N_z) = (M_x + n, M_y + n, M_z + n)$. We assume that the sample coordinates (x_i, y_i, z_i) already lie in the B-spline's knot-space, so that they can be used directly as parameters for the B-spline – e.g., if we were working in 1D, a sample located at $x = 1.5$ would be positioned halfway between knots

t_1 and t_2. The sample locations therefore need to be adjusted according to the chosen resolution and B-spline degree n.

The approximation problem now consists in finding the uniform B-spline (Eq. 2.7)

$$S(x, y, z) = \sum_{m=0}^{N_z-1} \sum_{l=0}^{N_y-1} \sum_{k=0}^{N_x-1} c_{k,l,m}\, \beta^n(x-k)\beta^n(y-l)\beta^n(z-m)$$

which minimizes the cost function (Eq. 3.5)

$$\mathcal{C}_p(S, \lambda_p) = \left(\sum_i \left[S(x_i, y_i, z_i) - f_i \right]^2 \right) + \lambda_p \mathcal{M}_p(S).$$

So the unknowns to be determined are the $N_x N_y N_z$ coefficients $c_{k,l,m}$.

The first step consists in expressing the regularization term $\mathcal{M}_p(S)$ in terms of these coefficients. To this end, we split $\mathcal{M}_p(S)$ into multiple integrals. For $p = 1$ and $p = 2$, based on Equations 3.3 and 3.4 we obtain

$$\mathcal{M}_1(S) = D_{1,0,0} + D_{0,1,0} + D_{0,0,1} \tag{3.7}$$

$$\mathcal{M}_2(S) = D_{2,0,0} + D_{0,2,0} + D_{0,0,2} + 2(D_{1,1,0} + D_{0,1,1} + D_{1,0,1}) \tag{3.8}$$

where each term D_{q_1,q_2,q_3} represents the integral of a specific squared partial derivative of order $p = q_1 + q_2 + q_3$ of S over the B-spline's inner signal, i.e.,

$$D_{q_1,q_2,q_3} = \int_n^{N_z} \int_n^{N_y} \int_n^{N_x} \left[\frac{\partial^{q_1+q_2+q_3}\, S}{\partial x^{q_1}\, \partial y^{q_2}\, \partial z^{q_3}}(x, y, z) \right]^2 \mathrm{d}x\, \mathrm{d}y\, \mathrm{d}z. \tag{3.9}$$

Let

$$\kappa_{q,k_1,k_2}(x) = \kappa_{q,k_2,k_1}(x) = \partial^q \beta^n(x - k_1) \cdot \partial^q \beta^n(x - k_2) \tag{3.10}$$

denote the product of the qth order derivatives of a pair of uniform basis B-splines (corresponding to coefficients c_{k_1} and c_{k_2}) at x. After some transformations, we then yield the equivalent definition:

$$D_{q_1,q_2,q_3} = \sum_{m_1,m_2} \sum_{l_1,l_2} \sum_{k_1,k_2} c_{k_1,l_1,m_1}\, c_{k_2,l_2,m_2} \cdot \int_n^{N_x} \kappa_{q_1,k_1,k_2}(x)\, \mathrm{d}x \int_n^{N_y} \kappa_{q_2,l_1,l_2}(y)\, \mathrm{d}y \cdot \int_n^{N_z} \kappa_{q_3,m_1,m_2}(z)\, \mathrm{d}z. \tag{3.11}$$

So for each term D_{q_1,q_2,q_3}, we sum up the products of all coefficient pairs multiplied by the product of each dimension's integral. $\mathcal{M}_p(S)$ is therefore a quadratic form of the B-spline coefficients $c_{k,l,m}$.

Matrix Formulation

Let us now introduce the corresponding matrix formulation to express the system of linear equations for the solution in the usual form $\mathbf{Ax} = \mathbf{b}$. To this end, we serialize the unknown coefficients $c_{k,l,m}$ into the vector

$$\mathbf{c} = (c_{0,0,0},\ \ldots,\ c_{N_x-1,0,0},\ \ldots,\ c_{N_x-1,N_y-1,N_z-1})^T$$

and the sample values into

$$\mathbf{f} = (\ldots,\ f_i,\ \ldots)^T.$$

The cost function (Eq. 3.5) can then be expressed as

$$\mathcal{C}(S) = \|\mathbf{Sc} - \mathbf{f}\|^2 + \lambda_p \mathbf{c}^T\mathbf{Rc} \tag{3.12}$$

where, after introducing the auxiliary function

$$\mathrm{i}(k,l,m) = N_x N_y \cdot m + N_x \cdot l + k \tag{3.13}$$

mapping a coefficient's 3D index (k,l,m) to its corresponding flat index in $\mathbf{c}$, the matrix $\mathbf{S}$ is defined by:

$$\{\mathbf{S}\}_{i,\,\mathrm{i}(k,l,m)} = \beta^n(x_i - k)\beta^n(y_i - l)\beta^n(z_i - m). \tag{3.14}$$

The matrix $\mathbf{R}$, responsible for the regularization term, directly results from the D_{q_1,q_2,q_3} (Eq. 3.11) and, referring to Eq. 3.10, is given by:

$$\{\mathbf{R}\}_{\mathrm{i}(k_1,l_1,m_1),\,\mathrm{i}(k_2,l_2,m_2)} = \sum_{(q_1,q_2,q_3)\in\mathcal{Q}} \int_n^{N_x} \kappa_{q_1,k_1,k_2}(x)\,\mathrm{d}x \int_n^{N_y} \kappa_{q_2,l_1,l_2}(y)\,\mathrm{d}y \cdot \int_n^{N_z} \kappa_{q_3,m_1,m_2}(z)\,\mathrm{d}z \tag{3.15}$$

where the set $\mathcal{Q}$ directly relates to Eq. 3.7 and 3.8:

$$\mathcal{Q} = \begin{cases} \{(1,0,0),(0,1,0),(0,0,1)\} & \text{for } p = 1, \\ \{(2,0,0),(0,2,0),(0,0,2),(1,1,0),(1,1,0),(0,1,1),(0,1,1),(1,0,1),(1,0,1)\} & \text{for } p = 2. \end{cases}$$

The cost function (Eq. 3.12) is clearly quadratic in the unknown coefficients $c_{k,l,m}$. Via vector differentiation, the minimum is finally obtained as the solution of the following system of linear equations:

$$(\mathbf{D} + \lambda_p\mathbf{R})\,\mathbf{c} = \mathbf{b} \tag{3.16}$$

where $\mathbf{D} = \mathbf{S}^T\mathbf{S}$ and $\mathbf{b} = \mathbf{S}^T\mathbf{f}$, i.e., the entries of the vector $\mathbf{b} = (\ldots,\ b_{\mathrm{i}(k,l,m)},\ \ldots)^T$ are given by:

$$\begin{aligned} b_{\mathrm{i}(k,l,m)} &= \sum_i f_i\,\{\mathbf{S}^T\}_{\mathrm{i}(k,l,m),\,i} = \sum_i f_i\,\{\mathbf{S}\}_{i,\,\mathrm{i}(k,l,m)} \\ &= \sum_i f_i\,\beta^n(x_i - k)\beta^n(y_i - l)\beta^n(z_i - m) \end{aligned} \tag{3.17}$$

and the matrix $\mathbf{D}$, again referring to Eq. 3.10, is defined by:

$$\begin{aligned}\{\mathbf{D}\}_{\mathrm{i}(k_1,l_1,m_1),\,\mathrm{i}(k_2,l_2,m_2)} &= \sum_i \{\mathbf{S}^T\}_{\mathrm{i}(k_1,l_1,m_1),i}\,\{\mathbf{S}\}_{i,\mathrm{i}(k_2,l_2,m_2)} = \sum_i \{\mathbf{S}\}_{i,\mathrm{i}(k_1,l_1,m_1)}\,\{\mathbf{S}\}_{i,\mathrm{i}(k_2,l_2,m_2)}\\ &= \sum_i \kappa_{0,k_1,k_2}(x_i)\,\kappa_{0,l_1,l_2}(y_i)\,\kappa_{0,m_1,m_2}(z_i).\end{aligned} \tag{3.18}$$

3.5 Constructing the Linear System

Matrix Layout

κ_{q,k_1,k_2} (Eq. 3.10) is the building block of both matrices $\mathbf{D}$ (Eq. 3.18) and $\mathbf{R}$ (Eq. 3.15). $\kappa_{q,k_1,k_2}(x)$ is the product of two nth degree uniform basis B-splines (or derivatives thereof) corresponding to coefficients c_{k_1} and c_{k_2}. Its support is hence limited to the intersection of both basis functions' support:

$$\kappa_q(x,k_1,k_2) = \begin{cases} 0 & \text{if } |k_2 - k_1| \geq n+1, \\ \partial^q\beta^n(x-k_1)\cdot\partial^q\beta^n(x-k_2) & \text{else.}\end{cases} \tag{3.19}$$

In 1D, this means that a coefficient c_k only interacts with itself and its n neighbors on each side, i.e., with the $(2n+1)$ coefficients starting from c_{k-n} up to and including c_{k+n}. The combined matrix

$$\mathbf{A} = \mathbf{D} + \lambda_p\mathbf{R} \tag{3.20}$$

is thus band-diagonal, the bandwidth being $2n+1$, leading to a tridiagonal matrix for linear B-splines and a heptadiagonal matrix for cubic ones. The symmetry $\kappa_{q,k_1,k_2}(x) = \kappa_{q,k_2,k_1}(x)$ further results in a symmetric matrix, so that only the main diagonal and n outer diagonals need to be stored in memory ($\sum_{i=0}^{n} N - i \leq (n+1)\cdot N$ elements instead of all N^2 elements). For example, the matrix for a linear 1D B-spline featuring three coefficients looks like this:

$$\mathbf{A} = \begin{pmatrix} d_{0,0} & d_{1,0} & \\ d_{1,0} & d_{0,1} & d_{1,1} \\ & d_{1,1} & d_{0,2}\end{pmatrix},$$

where $d_{j,k} = \{\mathbf{A}\}_{k,k+j} = \{\mathbf{A}\}_{k+j,k}$ is the kth element of diagonal $\mathbf{d_j}$ and represents the symmetric interaction between coefficients c_k and c_{k+j}. The diagonals may be serialized like this:

$$(\ d_{0,0}\ \ d_{0,1}\ \ d_{0,2}\,\big|\,d_{\mathbf{1},0}\ \ d_{\mathbf{1},1}\ ).$$

Multiplying $\mathbf{A}$ by $\mathbf{c}$ boils down to a convolution of $\mathbf{c}$ with a varying kernel of width $2n+1$, i.e., for our linear example:

$$\begin{aligned}&\mathbf{Ac} = \mathbf{b}\\ &b_k = d_{1,k-1}c_{k-1} + d_{0,k}c_k + d_{1,k}c_{k+1}.\end{aligned}$$

For the general case, the convolution is given by

$$b_k = \sum_{j=\max(-n,\,-k)}^{\min(n,\,(N-1)-k)} d_{|j|,\,k+\min(0,j)}\; c_{k+j}, \tag{3.21}$$

emphasizing that the first n and last n elements of $\mathbf{b}$ need to be handled separately because not all n neighbors on each side are available.

In higher dimensions, the kernel is obviously expanded to the same number of dimensions as well. In 2D, the kernel size is $(2n+1)^2$, additionally including parts of the n neighboring rows on each side. Therefore, in addition to a matrix for every coefficients row, we need a matrix for each distinct interaction between a row and its neighboring rows, e.g., for a linear 2D B-spline featuring 3×2 coefficients, it may be serialized like this:

$$\left(\begin{array}{ccc|cc}
d_{(0,0),\,(0,0)} & d_{(0,0),\,(1,0)} & d_{(0,0),\,(2,0)} & d_{(\mathbf{1},0),\,(0,0)} & d_{(\mathbf{1},0),\,(1,0)} \\
d_{(0,0),\,(0,1)} & d_{(0,0),\,(1,1)} & d_{(0,0),\,(2,1)} & d_{(\mathbf{1},0),\,(0,1)} & d_{(\mathbf{1},0),\,(1,1)} \\
\hline
d_{(0,\mathbf{1}),\,(0,0)} & d_{(0,\mathbf{1}),\,(1,0)} & d_{(0,\mathbf{1}),\,(2,0)} & d_{(\mathbf{1},\mathbf{1}),\,(0,0)} & d_{(\mathbf{1},\mathbf{1}),\,(1,0)}
\end{array}\right),$$

where

$$\begin{aligned} d_{(j_x,j_y),\,(k,l)} &= \{\mathbf{A}\}_{\mathrm{i}(k,l),\,\mathrm{i}(k+j_x,l+j_y)} = \{\mathbf{A}\}_{\mathrm{i}(k+j_x,l+j_y),\,\mathrm{i}(k,l)} \\ &= \{\mathbf{A}\}_{\mathrm{i}(k+j_x,l),\,\mathrm{i}(k,l+j_y)} = \{\mathbf{A}\}_{\mathrm{i}(k,l+j_y),\,\mathrm{i}(k+j_x,l)}. \end{aligned} \tag{3.22}$$

For this linear example, we yield $\sum_{i=0}^{n} N_x - i = 5$ vertical matrices (columns) and $\sum_{i=0}^{n} N_y - i = 3$ horizontal matrices (rows). Multiplication by $\mathbf{c}$ results in a 2D convolution:

$$b_{k,l} = \sum_{j_y=\max(-n,\,-l)}^{\min(n,\,(N_y-1)-l)} \; \sum_{j_x=\max(-n,\,-k)}^{\min(n,\,(N_x-1)-k)} d_{(|j_x|,\,|j_y|),\,(k+\min(0,j_x),\,l+\min(0,j_y))}\; c_{k+j_x,\,l+j_y}. \tag{3.23}$$

The layout of a matrix for 3D B-splines as well as cubic B-splines is analogous.

Constructing $\mathbf{D}$ and $\mathbf{b}$

$\mathbf{D} = \mathbf{S}^T\mathbf{S}$ and $\mathbf{b} = \mathbf{S}^T\mathbf{f}$ are both based on $\mathbf{S}$, which depends upon the sample set. We thus implemented a parallel construction of $\mathbf{D}$ and $\mathbf{b}$ by processing the samples in a serial fashion. In 1D, a sample only affects the $(n+1)$ surrounding coefficients directly, so in 3D it contributes a kernel of size $(n+1)^3$ to $\mathbf{S}$ (Eq. 3.14). This kernel is computed for each sample and then used to compute the sample's contributions to $\mathbf{b}$ (Eq. 3.17) (in form of the $\mathbf{S}$ kernel multiplied by f_i) as well as to $\mathbf{D}$ (Eq. 3.18) (a squared form of the $\mathbf{S}$ kernel).

The computational complexity for the construction of $\mathbf{D}$ and $\mathbf{b}$ is hence proportional to the number of samples.

Constructing R

Due to the compact support of basis B-splines, the integral of κ_{q,k_1,k_2} (Eq. 3.19) over the inner signal can be written as

$$\int_n^N \kappa_{q,k_1,k_2}(x)\,\mathrm{d}x = \begin{cases} 0 & \text{if } |k_2 - k_1| \geq n+1, \\ \int_{\max(n,\,\max(k_1,k_2))}^{\min(N,\,\min(k_1,k_2)+(n+1))} \kappa_{q,k_1,k_2}(x)\,\mathrm{d}x & \text{else.} \end{cases} \tag{3.24}$$

Since $\kappa_{q,k_1,k_2}(x) = \kappa_{q,k_1+s,k_2+s}(x+s)$ for any s, we obtain the equivalent definition in case $k_1 \leq k_2$, with $s = -k_1$ and $\delta_k = k_2 - k_1$:

$$\int_n^N \kappa_{q,k_1,k_2}(x)\,\mathrm{d}x = \begin{cases} 0 & \text{if } \delta_k \geq n+1, \\ \int_{\max(n-k_1,\,\delta_k)}^{\min(N-k_1,\,n+1)} \kappa_{q,0,\delta_k}(x)\,\mathrm{d}x & \text{else.} \end{cases} \tag{3.25}$$

This leads to the shortcut

$$\begin{aligned} \int_n^N \kappa_{q,k_1,k_2}(x)\,\mathrm{d}x &= \sum_{i=\max(\delta_k,\,n-k_1)}^{\min(n,\,(N-1)-k_1)} \eta_i(q,\,\delta_k) \\ \eta_i(q,\delta_k) &= \int_i^{i+1} \kappa_{q,0,\delta_k}(x)\,\mathrm{d}x = \int_i^{i+1} \partial^q\beta^n(x)\cdot\partial^q\beta^n(x-\delta_k)\,\mathrm{d}x, \end{aligned} \tag{3.26}$$

i.e., the integral for a given order q only depends upon the distance δ_k between both basis B-splines for most coefficient pairs, exceptions being most pairs involving the first n and last n coefficients, because some of their basis B-spline intervals lie outside the inner signal and are hence not covered by the integral.

The few per-interval integrals $\eta_i(q,\delta_k)$ can therefore be precomputed for all $q \in \{0\ldots p\}$, $\delta_k \in \{0\ldots n\}$ and $i \in \{\delta_k \ldots n\}$. Constructing $\mathbf{R}$ (Eq. 3.15) then consists in processing each partial derivative separately by summing up the involved η_is for each dimension's integral, multiplying the integrals and adding the product to the corresponding matrix coefficient. The computational complexity is thus proportional to the total number of B-spline coefficients as well as to the number of partial derivatives and independent from the samples set. The computation of a partial derivative's contributions is easily parallelizable.

3.6 Solving the Linear System via a Multi-Scale Pyramid

Arigovindan et al. [3, pages 5-6] propose to employ a common iterative solution scheme such as the the Gauss-Seidel, Jacobi or conjugate-gradient methods and to combine it with a multigrid approach to exploit the interscale relation of B-splines. The core of this approach consists in expressing the linear system in coarser scales by iteratively halving the B-spline resolution along each dimension and thus yielding a multi-scale pyramid of uniform B-splines. So if we have a linear system in scale j:

$$\mathbf{A}_j \mathbf{c}_j = \mathbf{b}_j,$$

the idea consists in approximating it in the coarser scale $(j+1)$ by deriving a corresponding linear system:

$$\mathbf{A}_{j+1}\mathbf{c}_{j+1} = \mathbf{b}_{j+1}.$$

Let us express the expansion of a B-spline coefficients vector from scale $(j+1)$ to j as matrix $\mathbf{U}_{j+1}$ and the adjoint coarsening from scale j to $(j+1)$ as matrix $\mathbf{D}_j$. $\mathbf{c}_{j+1}$ denotes the unknown coefficients of the coarser B-spline. The coarser right-hand-side vector $\mathbf{b}_{j+1}$ is obtained by coarsening the finer one: $\mathbf{b}_{j+1} = \mathbf{D}_j\mathbf{b}_j$. The coarser matrix $\mathbf{A}_{j+1}$ represents the synthesis of three operations: expanding $\mathbf{c}_{j+1}$ to $\tilde{\mathbf{c}}_j$, multiplying $\mathbf{A}_j$ by $\tilde{\mathbf{c}}_j$ to obtain $\tilde{\mathbf{b}}_j$ and then coarsening the result to $\mathbf{b}_{j+1}$. The coarser matrix is hence given by $\mathbf{A}_{j+1} = \mathbf{D}_j\mathbf{A}_j\mathbf{U}_{j+1}$. For a 1D B-spline, this boils down to a 2D convolution of the finer matrix $\mathbf{A}_j$ with a filter consisting of the tensor-product of the expansion and coarsening filters, followed by 2D downsampling (discarding every other row and column). So the filter for linear 1D B-splines is given by:

$$\left[\frac{1}{2}\cdot\begin{pmatrix}1 & 2 & 1\end{pmatrix}\right] \otimes \left[\frac{1}{4}\cdot\begin{pmatrix}1\\2\\1\end{pmatrix}\right] = \frac{1}{8}\cdot\begin{pmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{pmatrix}$$

and the filter for cubic 1D B-splines by:

$$\left[\frac{1}{8}\cdot\begin{pmatrix}1 & 4 & 6 & 4 & 1\end{pmatrix}\right] \otimes \left[\frac{1}{16}\cdot\begin{pmatrix}1\\4\\6\\4\\1\end{pmatrix}\right].$$

It is important to note that the expansion and coarsening matrices need to be related by $\mathbf{D}_j = s\cdot\mathbf{U}_{j+1}^T$ for an arbitrary scale factor $s \neq 0$ (in our case, $s = \frac{1}{2}$), otherwise $\mathbf{A}_{j+1}$ would generally not be symmetric anymore, see the attached proof. In practice, this means that coefficients cannot be mirrored (representing a periodic signal as illustrated in Figure 2.5), and that unavailable fine matrix coefficients in the convolution window need to be treated as zeros. These natural boundary conditions are hence required when expanding/coarsening vectors $\mathbf{c}_j$ and $\mathbf{b}_j$ too (see Figures 2.3 and 2.4).

In higher dimensions, coarsening $\mathbf{A}_j$ to $\mathbf{A}_{j+1}$ is analog to coarsening a vector in higher dimensions – each dimension's matrices are coarsened separately by using the intermediate results from the previously coarsened dimension.

A derived coarser linear system offers multiple usages for a finer scale. Firstly, its solution $\mathbf{c}_{j+1}$ can be expanded to $\tilde{\mathbf{c}}_j$ and then used as initial estimate for the finer scale to improve the convergence of the iterative solution method. Secondly, it can be used for so-called coarse-grid correction: after obtaining an estimate for the fine solution after k iterations $\mathbf{c}_j^{(k)}$, the residue $\mathbf{r}_j^{(k)} = \mathbf{b}_j - \mathbf{A}_j\mathbf{c}_j^{(k)}$ is computed. The residue is coarsened to $\mathbf{r}_{j+1}^{(k)}$ and used to form a new linear system in the coarser scale:

$$\mathbf{A}_{j+1}\mathbf{e}_{j+1}^{(k)} = \mathbf{r}_{j+1}^{(k)}.$$

The solution $\mathbf{e}_{j+1}^{(k)}$ of this linear system is a coarse and thus low-frequent version of the estimate's error $\mathbf{e}_j^{(k)} = \mathbf{c}_j - \mathbf{c}_j^{(k)}$. By expanding $\mathbf{e}_{j+1}^{(k)}$ to $\tilde{\mathbf{e}}_j^{(k)}$ and adding it to $\mathbf{c}_j^{(k)}$, we obtain a refined estimate for the fine solution $\mathbf{c}_j = \mathbf{c}_j^{(k)} + \mathbf{e}_j^{(k)}$.

Algorithms 1 and 2 summarize our implementation. We employ the Successive Over Relaxation (SOR) method as underlying solver. This method is a simple extension of the Gauss-Seidel method and extends it by a so-called over-relaxation factor $\omega \in (0, 2)$. All of our results are obtained by using $\omega = 1$, resulting in the Gauss-Seidel method.

Most of the involved operations are easily parallelizable. These include all coarsening and expansion operations, multiplying $\mathbf{A}$ by $\mathbf{c}$ ($N_x N_y N_z$ independent 3D convolutions) as well as adding/subtracting vectors and computing their norm. The Gauss-Seidel/SOR method is not inherently parallelizable (unlike the Jacobi method, for example). Since it is similar to multiplication, the computation of the $N_x N_y N_z$ dependent 3D convolutions can be optimized: when refining a slice, we can precompute the independent contributions from the $2n$ neighboring slices (for all $N_x N_y$ slice coefficients to be refined) in a parallel fashion.

Algorithm 1 Computing a pyramid of uniform B-splines consisting in l scales, based on the matrix $\mathbf{A}_0$ and right-hand-side vector $\mathbf{b}_0$ for the finest scale. *correct* enables/disables coarse-grid correction; if not stated otherwise, we disable it.

```
// derive the linear system at coarser scales, based on $\mathbf{A}_0$ and $\mathbf{b}_0$
for $j = 0$ to $l - 2$ do
    coarsen the matrix: $\mathbf{A}_{j+1} = \mathbf{D}_j \mathbf{A}_j \mathbf{U}_{j+1}$
    coarsen the right-hand-side vector: $\mathbf{b}_{j+1} := \mathbf{D}_j \mathbf{b}_j$
end for

initialize the coarsest solution with zeros: $\mathbf{c}_{l-1} := \mathbf{0}$
for $j = l - 1$ to 0 do
    Solve($j$, $l$, $correct$)
    if $j > 0$ then
        initialize the next finer solution: $\mathbf{c}_{j-1} := \mathbf{U}_j \mathbf{c}_j$
        if $correct$ then
            back up $\mathbf{c}_j$ (it will be replaced by $\mathbf{e}_j$ when correcting the next finer scale)
        end if
    end if
end for
```

Algorithm 2 Solve(j, l, $correct$): solve the linear system in scale j for a pyramid of l scales; $correct$ enables/disables coarse-grid correction. $convergenceThreshold$ controls the minimum residue norm improvement for a Successive Over Relaxation (SOR) iteration – e.g., a value of 0.1 terminates the refinement as soon as the residue norm has not improved by at least 10%. To balance the cost of a SOR iteration vs. the cost for computing the residue, we use blocks of N SOR iterations inbetween residue improvement checks. If not stated otherwise, we use $convergenceThreshold = 0.1$, $N = 2$, $correctRecursively =$ **false** and $numCorrections = 1$.

```
// compute an iteration block's convergence threshold
$blockThreshold := 1 - (1 - convergenceThreshold)^N$

// initial iteration block
perform $N$ SOR iterations
compute the residue vector: $\mathbf{r}_j := \mathbf{b}_j - \mathbf{A}_j \mathbf{c}_j$
compute the residue norm: $r_{\text{new}} := \|\mathbf{r}_j\|$

if $correct$ and $j < l - 1$ then
    // perform repeated coarse-grid corrections, each followed by an iteration block
    for $i = 0$ to $numCorrections - 1$ do
        replace the coarser right-hand-side vector with the coarsened residue: $\mathbf{b}_{j+1} := \mathbf{D}_j \mathbf{r}_j$
        initialize the coarser solution (representing $\mathbf{e}_{j+1}$) with zeros: $\mathbf{c}_{j+1} := \mathbf{0}$
        solve the linear system for the coarse error: Solve($j + 1$, $l$, $correctRecursively$)
        correct the solution estimate by adding the expanded coarse error estimate: $\mathbf{c}_j := \mathbf{c}_j + \mathbf{U}_{j+1} \mathbf{c}_{j+1}$

        perform $N$ SOR iterations
        update the residue vector $\mathbf{r}_j$
        $r_{\text{new}} := \|\mathbf{r}_j\|$
    end for
end if

repeat
    $r_{\text{old}} := r_{\text{new}}$
    perform $N$ SOR iterations
    update the residue vector $\mathbf{r}_j$
    $r_{\text{new}} := \|\mathbf{r}_j\|$
until $\frac{r_{\text{old}} - r_{\text{new}}}{r_{\text{old}}} < blockThreshold$
```

CHAPTER 4

Block-Based Reconstruction

In the previous chapter, we have seen how to reconstruct a volume based on point samples using a multi-scale pyramid of uniform B-splines. Instead of reconstructing the full volume globally, subdividing it into smaller blocks and reconstructing each block independently provides three key advantages:

- Temporary memory requirements can be reduced.
  This is best demonstrated by giving an example: reconstructing a volume of 256^3 inner cubic B-spline intervals requires $(256 + n)^3 = 259^3$ B-spline coefficients. The total length of the $n + 1 = 4$ diagonals of each 1D matrix will then be $\sum_{i=0}^{n} 259 - i = 1030$, leading to 1030^3 matrix coefficients in total. Assuming a single-precision floating-point format (4 bytes) for the coefficients, the matrix alone will have a foot-print of more than 4 Gigabytes. By doubling the resolution along each dimension (512^3), the matrix would require more than 32 Gigabytes. Reconstructing large/high-resolution volumes therefore tends to require more RAM than current desktop PCs offer.

- Blocks can be reconstructed in a parallel fashion instead of parallelizing only suited steps of the reconstruction process, leading to a better utilization of multiple CPUs/cores and thus decreasing the overall run-time. Of key importance here is that the construction of matrix $\mathbf{D}$ and right-hand-side vector $\mathbf{b}$ is not efficiently parallelizable.

- The block resolution can be adapted with respect to the local samples density. This allows to use coarser resolutions for sparse blocks and to skip empty/very sparse blocks altogether, speeding up the reconstruction.

These advantages come at a cost: when reconstructing a block, only the point samples lying inside its inner signal range can be regarded, outer ones are ignored. This generally leads to a discrepancy compared to the corresponding subregion of the globally reconstructed volume. This is due to two reasons:

- A block's borders would globally be also directly influenced by outer point samples lying inside the block's full signal range, according to the definition of matrix $\mathbf{S}$ (Eq. 3.14).
- Via regularization, we enforce some smoothness on the B-spline. This results in a coefficient being indirectly influenced by neighboring coefficients where the influence decreases with the distance. Since a block represents a subregion of the global volume, it only contains the coefficients needed to define its inner signal. The clipped coefficients neighborhood decreases the accuracy of block borders at boundaries shared by multiple blocks.

The discrepancy leads to discontinuities at block boundaries. We propose to combine two extensions to overcome this problem:

- Using temporary padding at shared block boundaries to extend the context by including outer point samples as well as additional coefficients. While this redundancy implies a computational overhead, it reduces the discrepancy between block-based and global reconstruction at block borders.
- Fusing adjacent blocks, i.e., enforcing C^{n-1} continuity at block boundaries.

4.1 Padding

By extending a block at shared boundaries, outer point samples lying in the new inner signal range of the padded block can be included. Additionally, the borders of the actual block are provided with an extended coefficients neighborhood. The increased context leads to more accurate results for the actual block, which is extracted after reconstructing the padded block.

The multi-scale pyramid needs to be aligned on the beginning of the actual inner signal in order to allow the extraction of the actual block from every padded scale (see Figure 4.1). If the left padding in a finer scale consists in an odd number of B-spline intervals, the coarser scale is offset by one fine interval to the left. The right padding is treated in an analogous way and affects

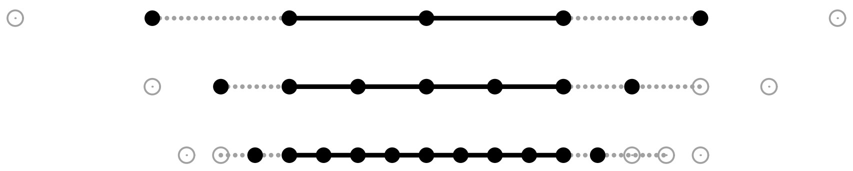


Figure 4.1: Schematics of a multi-scale pyramid of 3 uniform cubic 1D B-splines for a finest inner signal range of 8 intervals (bottom), padded by 2 intervals to the left and 3 intervals to the right. The actual inner signal range is indicated by solid lines, the extended inner signal range of the padded B-splines by dotted lines; the coefficients of the actual B-splines are filled, the additional ones of the padded B-splines are not. Coarser scales need to be aligned on the beginning of the actual inner signal, leading to potentially different offsets across scales.

the number of coarse coefficients. Each dimension's offset needs to be taken into account when expanding/coarsening vectors and when coarsening matrices. The motivation for this scheme is based on coarsening B-spline coefficients using natural boundary conditions (treating missing coefficients as zeros): we include all coarse coefficients affected by at least one fine coefficient, and exclude all outer ones which are implicitly zero.

4.2 Single-Scale Blocks Fusion

1D

A B-spline can be losslessly split into two autonomous parts by duplicating some of its coefficients (see Figure 4.2). Here, autonomous means that each part contains all coefficients needed to autonomously represent its own range as inner B-spline signal. We therefore use redundant outer coefficients for each part. For a split at an arbitrary location, the $(n + 1)$ coefficients surrounding the boundary need to be included by both parts. A split aligned at a knot is a special case and requires an overlap of only n coefficients.

The adjoint operation depicted in Figure 4.3 consists in fusing two adjacent parts so that their shared boundary retains C^{n-1} continuity of the underlying B-spline. Based on the splitting operation, it is sufficient to make sure that the n-wide overlapping coefficient regions of both parts are equal to satisfy this constraint. The question which arises here is how to merge the left and right regions to a single one. We propose a simple and intuitive scheme for linear and cubic B-splines to be fused at a shared knot:

- For linear B-spline parts, the single overlapping coefficient (located at the boundary) is simply merged to the average of both parts' corresponding coefficient.

- The overlapping region of cubic B-spline parts contains $n = 3$ coefficients. The first one is taken from the left part, the last one from the right part and, analogous to linear fusion, the central one is replaced by the average of each part's central coefficient. This scheme is motivated by the fact that outer coefficients (actually lying in the other part) are replaced by the corresponding inner ones from the other part, where they had more context during reconstruction, hence being more accurate.

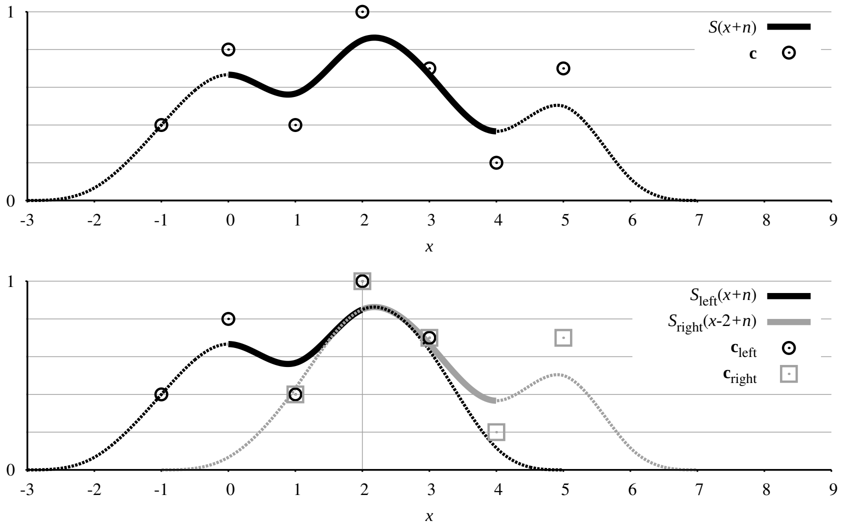


Figure 4.2: Splitting a uniform cubic 1D B-spline (top) at a knot into two autonomous parts (bottom) by duplicating $n = 3$ coefficients.

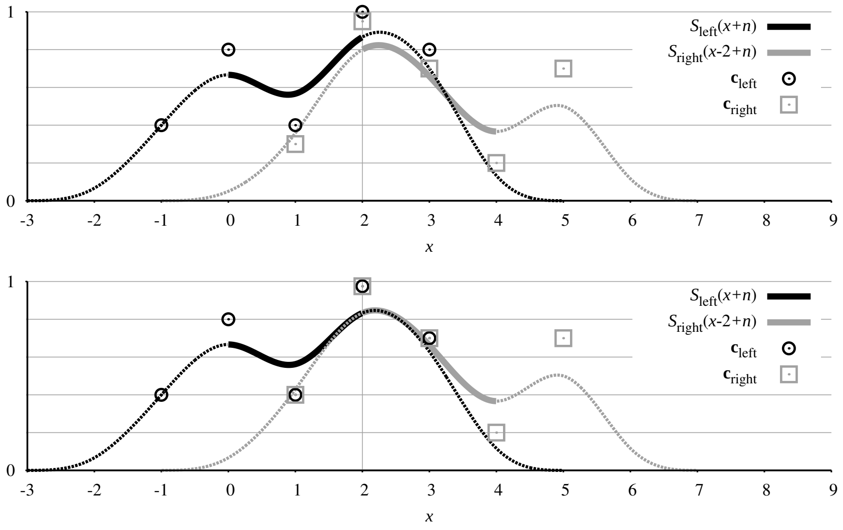


Figure 4.3: Fusing two adjacent uniform cubic 1D B-splines (top) to simulate a single B-spline (bottom) by merging the $n = 3$ overlapping coefficients.

Higher Dimensions

In higher dimensions, the n-wide border regions of a block's coefficients grid may overlap with multiple other blocks. In 2D, these regions are the corners (the intersection of 4 adjacent blocks); in 3D, the edges (4 blocks) and the corners (8 blocks). To enforce C^{n-1} continuity, all overlapping coefficients need to be equal, just as for the 1D case. The 1D fusing scheme can be extended to higher dimensions and is best demonstrated visually (see Figure 4.4).

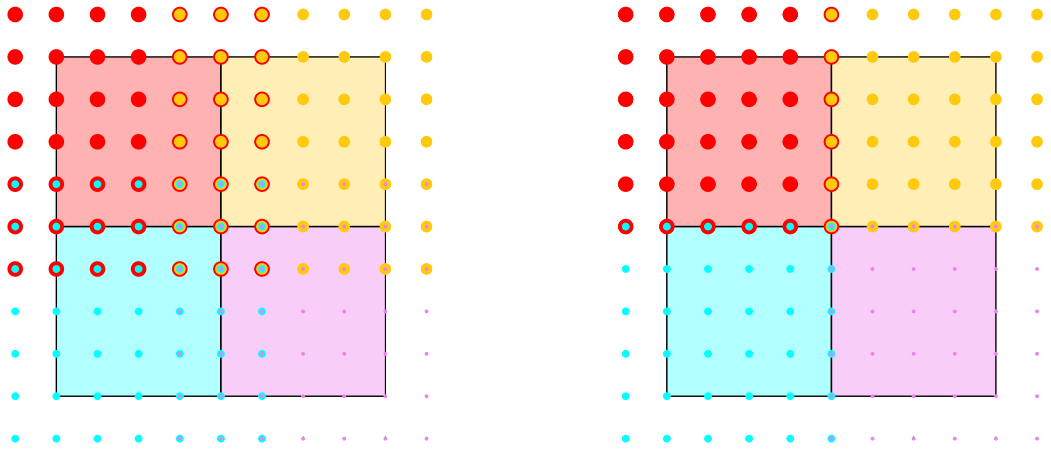


Figure 4.4: Example for fusing in 2D. The left image represents 4 adjacent blocks (each encoding an inner cubic B-spline signal of 4^2 intervals and associated with a specific color) and illustrates overlapping coefficient regions. The blocks are fused by making sure all overlapping coefficients are equal. According to the simple scheme used for fusing cubic 1D B-splines, we want to merge them as depicted in the right image (where overlapping coefficients represent their average).

A simple algorithm implementing the fusing scheme for arbitrary dimensions can be formulated if the blocks form a Cartesian lattice, i.e., if there is exactly one neighboring block at every inner block side. It consists in fusing all inner block sides, analog to the 1D case and ordered by the corresponding axis – i.e., fusing all blocks along x first, then all blocks along y and finally all blocks along z (see Figure 4.5; the order of the axes is interchangeable).

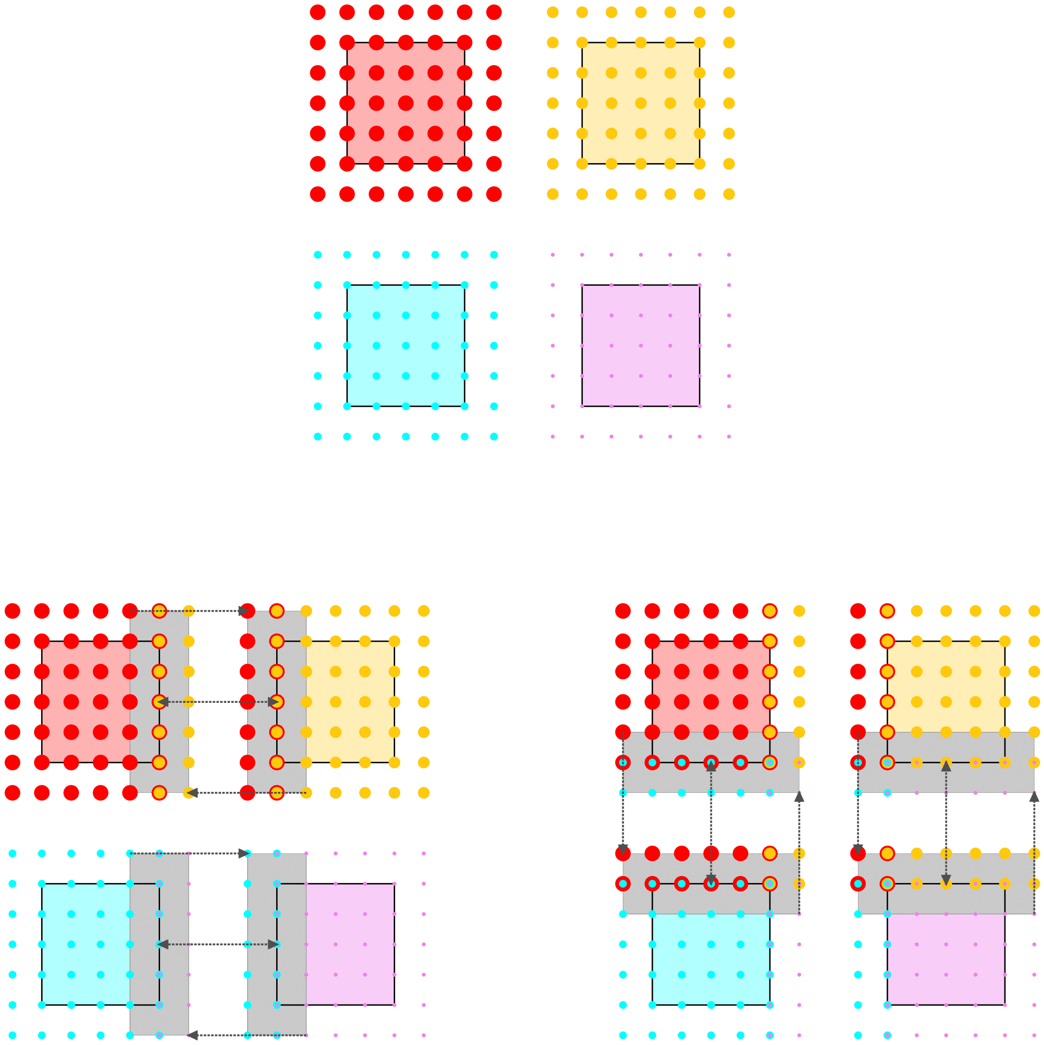


Figure 4.5: Illustration of the fusing algorithm for higher dimensions using the 2D example from Figure 4.4. The top image shows the 4 autonomous blocks and their coefficients. The first step consists in fusing all blocks along the x axis (every block with its neighbor to the right, analog to 1D fusion), leading to the bottom left image (where overlapping coefficients represent their average). Afterwards, all blocks are fused along y (every block with its upper neighbor), finally resulting in the bottom right image which corresponds to the target in Figure 4.4.

4.3 Interscale Fusion

In the previous section, we explained how adjacent blocks of identical resolution can be fused to ensure C^{n-1} continuity at block boundaries. We will now tackle the problem of fusing adjacent blocks whose resolutions differ by an arbitrary power of two. The block boundaries still need to be aligned on shared knots. The reason for these restrictions is the exploitation of the interscale relation of B-splines.

1D

Given a B-spline of odd degree n in a coarse scale $(j + 1)$, the interscale relation allows to duplicate the B-spline in a finer scale j by expanding the coefficients as explained in chapter 2. The value of a B-spline at an inner knot is a linear combination of only n basis B-splines (not the general $n+1$ ones) and hence only influenced by the surrounding n coefficients. So to duplicate the coarse B-spline at a boundary knot in a finer scale, it is sufficient to replace the surrounding n fine coefficients by the ones obtained by expanding the surrounding n coarse coefficients. By doing so, we trade accuracy for a C^{n-1} continuous blending between coarse and fine signals at the fine border (see Figure 4.6). The scheme only affects the first or last n inner B-spline intervals of the fine signal and enforces the shortest possible blending between the signals.

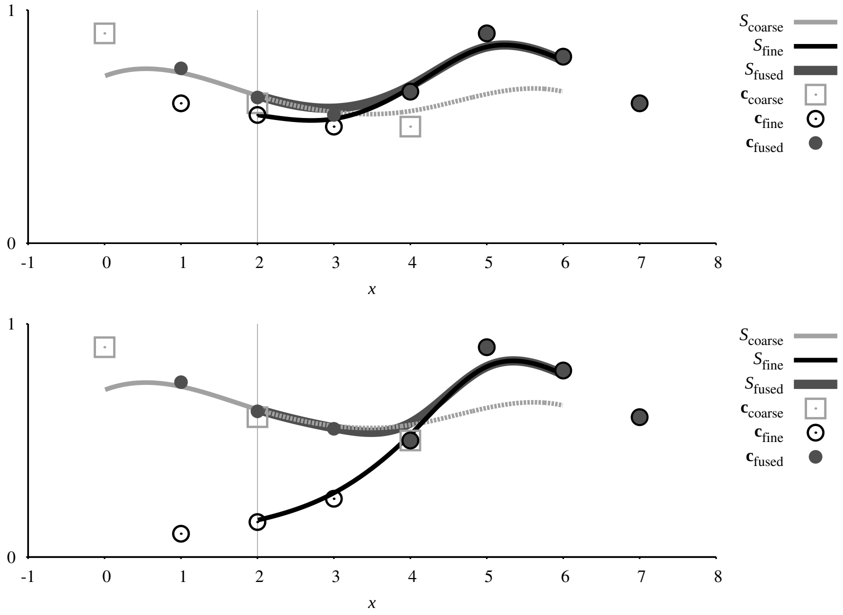


Figure 4.6: Two examples for fusing a fine uniform cubic 1D B-spline (black curve) with an adjacent coarser one (light-grey) by replacing the n fine coefficients surrounding the boundary by the ones obtained by expanding the surrounding n coarse coefficients, resulting in the fused curve (dark-grey). Only the inner signals are visualized.

This observation can also be derived formally. Let us define the uniform cubic B-spline interval (Eq. 2.6) explicitly:

$$\begin{aligned} S^{(i)}(t) &= \sum_{j=0}^{3} c_{i+j}\beta^3(t+3-j) \\ &= \frac{(-t^3+3t^2-3t+1)c_i + (3t^3-6t^2+4)c_{i+1} + (-3t^3+3t^2+3t+1)c_{i+2} + (t^3)c_{i+3}}{6} \end{aligned} \tag{4.1}$$

At the beginning of the ith inner signal interval of a B-spline in an arbitrary scale j, we thus yield the equations

$$S_j^{(i)}(0) = \frac{1}{6}\left(c_{j;\,i} + 4\,c_{j;\,i+1} + c_{j;\,i+2}\right) \tag{4.2}$$

$$\frac{\partial S_j^{(i)}}{\partial t_j}(0) = \frac{1}{2}\left(-c_{j;\,i} + c_{j;\,i+2}\right) \tag{4.3}$$

$$\frac{\partial^2 S_j^{(i)}}{\partial t_j^2}(0) = c_{j;\,i} - 2\,c_{j;\,i+1} + c_{j;\,i+2}. \tag{4.4}$$

An analog set of equations is obtained for a coarser scale $(j+1)$. For the beginning of a fine interval (index $2i$) to be congruent with the beginning of a corresponding coarser one (index i), we yield the following system of equations, considering the fact that the fine interval is half as long as the coarser one and thus $t_j = 2t_{j+1}$:

$$S_{j+1}^{(i)}(0) = 2^0 \cdot S_j^{(2i)}(0) \tag{4.5}$$

$$\frac{\partial S_{j+1}^{(i)}}{\partial t_{j+1}}(0) = 2^1 \cdot \frac{\partial S_j^{(2i)}}{\partial t_j}(0) \tag{4.6}$$

$$\frac{\partial^2 S_{j+1}^{(i)}}{\partial t_{j+1}^2}(0) = 2^2 \cdot \frac{\partial^2 S_j^{(2i)}}{\partial t_j^2}(0) \tag{4.7}$$

Assuming the coarse boundary coefficient has an index of i and the aligned finer one an index of $2i$, the solution of this linear system for the fine coefficients is

$$\begin{aligned} c_{j;\,2i-1} &= \frac{1}{8}\left(4\,c_{j+1;\,i-1} + 4\,c_{j+1;\,i}\right) \\ c_{j;\,2i} &= \frac{1}{8}\left(c_{j+1;\,i-1} + 6\,c_{j+1;\,i} + c_{j+1;\,i+1}\right) \\ c_{j;\,2i+1} &= \frac{1}{8}\left(4\,c_{j+1;\,i} + 4\,c_{j+1;\,i+1}\right), \end{aligned}$$

which is equivalent to expanding the n coarse coefficients to the central n finer ones.

The accuracy of this interscale fusing scheme is increased if the finer block is available in the coarser scale too, so that the blocks can be fused in the shared coarser scale beforehand, leading to more accurate coarse B-spline coefficients to be expanded.

Higher Dimensions

The scheme for the 1D case can be extended to higher dimensions by expanding n-wide overlapping coefficient regions. See Figure 4.7 for a visual explanation.

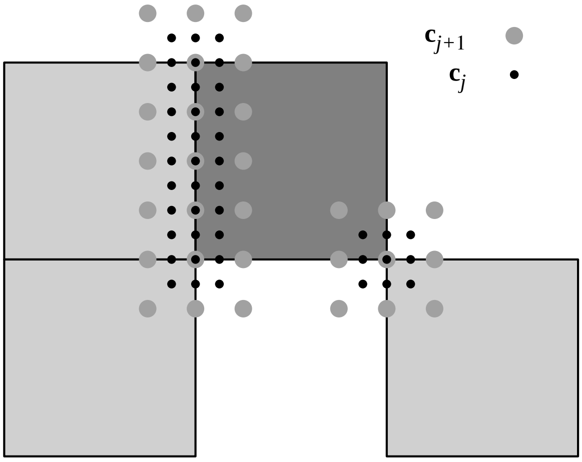


Figure 4.7: Fusing a fine block in scale j (dark-grey) with 3 neighboring coarser blocks in scale $(j+1)$ (light-grey), ignoring the other 5 neighbors for a moment. The affected fine coefficient regions are the bottom-left 3×3 corner, the bottom-right 3×3 corner and the left 3×11 side. The fine coefficients in these regions need to be replaced by the ones obtained by expanding the corresponding coarse regions.

Figure 4.7 illustrates that in higher dimensions, a block's border regions may overlap with multiple neighboring blocks – the bottom-left 3×3 corner of the fine block in scale j is affected by both the left and the bottom-left neighbors. So to obtain identical expansions, the overlapping coarse coefficients (of the left and bottom-left blocks) need to be equal. This is guaranteed by fusing blocks of identical resolution before fusing blocks across scales.

If a block's neighborhood of $(3^d - 1)$ other blocks features multiple coarser scales, we need to order the replacements of the fine coefficients such that in the end, a fine coefficient region affected by one or more coarser neighboring blocks is an expansion of the corresponding region of the coarsest neighbor. So for each block, we first identify its border regions to be replaced by scanning the $(3^d - 1)$ neighbors for coarser blocks; as we have seen, the regions may not be disjoint. These expansion jobs are then processed ascendingly by the scale difference $\delta_j = j_{\text{neighbor}} - j_{\text{block}}$. So if the bottom-left neighbor in Figure 4.7 was given in scale $(j+2)$, the left 3×11 side of the block in scale j would be first replaced by the expansion of the corresponding 3×7 side of the left neighbor (in scale $(j+1)$). The bottom-left 3×3 corner would then later be overridden by expanding the corresponding 3×3 corner of the bottom-left neighbor (in scale $(j+2)$) twice.

4.4 Fusing Multi-Scale Pyramids

Fine blocks need to be available in coarser scales too for our higher-dimensional single-scale fusing algorithm to work properly, since it requires both blocks for a side, all 4 blocks for an edge and all 8 blocks for a corner to achieve a correct merging of the overlapping coefficient regions. By choosing a common minimum resolution for each block's multi-scale pyramid and allowing for an arbitrary number of finer scales, fusing all blocks in the common coarsest scale yields global C^{n-1} continuity for the base scale. For each finer scale, we apply the single-scale fusing algorithm, fusing sides if both blocks are available in that scale. This means that if a block's pyramid only includes coarser scales, all adjacent sides, edges and corners in the current scale will not be merged correctly; all other block boundaries will be continuous. This problem is handled separately by interscale fusion – all of these diverging coefficient regions in the up to $(3^d - 1)$ neighboring finer blocks will later be replaced by the expansion of the corresponding coarse regions.

4.5 Holes

If a block contains no or only very few point samples, it may be treated as a hole, skipping the reconstruction and implying a null-signal instead. In this case, we need to make sure that the signals of all neighboring blocks fade out to zero at adjacent boundaries in order to achieve C^{n-1} continuity. This is accomplished by replacing the virtually overlapping n-wide coefficient regions of the up to $(3^d - 1)$ neighboring blocks with zeros, for all scales.

4.6 Approximating Continuous Gradients for Linear B-Splines and Associated Implications

A linear B-spline $S(x)$ is C^0 continuous, so its first derivative is constant for each interval and hence generally not continuous at a knot. A well-known work-around to obtain continuous gradient approximations is central differencing: $\partial S(x) \approx S(x + 0.5) - S(x - 0.5)$.

By employing this scheme, the signal needs to be padded by one interval at each side to allow the approximation of the gradient at the inner signal extremities. We therefore need to include two outer coefficients along each dimension, yielding the same number of total coefficients as for a cubic B-spline. This obviously also affects our fusing scheme. For the initial outer coefficients, we use mirroring. We then apply the same single-scale fusing scheme as for cubic B-splines, replacing outer coefficients by inner ones from neighboring blocks. This results in adjacent linear B-splines sharing two identical intervals at the boundary and thus ensures continuous block boundaries as well as continuous gradient approximations. The interscale fusing scheme is analog to the one for cubic B-splines too: overlapping coefficient regions of finer blocks are replaced by (linearly) expanding corresponding coarser ones, leading to a partial duplication of the two coarse intervals in the finer scale, again yielding continuous boundaries and continuous gradient approximations.

4.7 Relation of Resolution and Lambda

The resolution at which the linear system is constructed has an effect on the square of Duchon's semi-norm of order p (Eq. 3.2):

$$\mathcal{M}_p(S) = \int_n^{N_z} \int_n^{N_y} \int_n^{N_x} \|D^p S\|^2 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z.$$

If we duplicate a signal in scale j in a finer scale $(j-1)$ (by expanding the coefficients), the squared semi-norm will be scaled by 2^{d-2p} due to two reasons:

1. According to equation (Eq. 2.12), derivatives of order p are scaled by 2^{-p} in a finer scale. $\|D^p S\|^2$ is hence scaled by 2^{-2p}.

2. The range of the integral, the inner signal range, is doubled along each dimension in a finer scale. The range is therefore scaled by 2^d.

The regularization term in a coarser scale $j + \delta_j$ will hence be scaled by $2^{\delta_j(2p-d)}$ compared to scale j. In order to enforce an identical global smoothness constraint, the regularization parameter λ_p needs to be adjusted by the inverse factor $2^{\delta_j(d-2p)}$. This is important in case the finest resolutions of the block multi-scale pyramids differ.

CHAPTER 5

Adaptive Subdivision

In the previous chapter, we discussed the advantages of block-based reconstruction and how to remedy its disadvantages. We will now focus on how to adaptively subdivide a volume into blocks and on how to derive various levels-of-detail of the reconstructed volume.

5.1 Primary Subdivision into Cubes

The fusing scheme presented in the previous chapter requires the blocks to be aligned on shared knots and the resolutions to differ by powers of two. To satisfy these constraints, we first subdivide the volume into a Cartesian lattice of cubes and use powers of two as resolution levels for each cube (level l: 2^l inner B-spline intervals along each dimension). These cubes represent atomic block primitives and enable the higher-dimensional single-scale fusing algorithm presented in the previous chapter. The cubes are generally of identical size, although they may be trimmed to cuboids at the volume borders. The edge length of the cubes and hence the granularity of the subdivision is defined by an arbitrary parameter $s > 0$ indicating the number of cubes along the volume's longest axis. Figure 5.1 illustrates a 2D example using $s = 3$ squares along the longest axis. Note that if we chose $s = 2$, we would obtain a 2×2 grid, and the squares in the top row would be trimmed to rectangles.

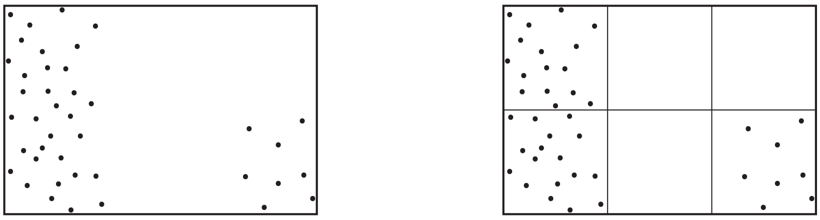


Figure 5.1: Example for primary subdivision in 2D: the bounding box of the given point cloud (left) is subdivided into a Cartesian lattice of 3×2 squares (right).

5.2 Hierarchical Subdivision into a Bounding Interval Hierarchy

A hierarchical subdivision in form of a tree on top of the cubes grid allows to merge cubes adaptively to larger cuboid blocks, according to some criteria. A binary partitioning scheme such as the widely used k-d tree allows for more flexibility than octrees, another common choice in 3D. We are going to subdivide the volume hierarchically into cubes by constructing a Bounding Interval Hierarchy (BIH) [19], an extension of k-d trees. The extension consists in storing a separate bounding plane for each child instead of a single splitting plane per inner node. This is particularly useful for sparse data.

A visual overview in 2D is given in Figure 5.2. The top image depicts the $3{\times}2$ grid from Figure 5.1. The axis-aligned bounding box (AABB) of the BIH root node (G) hence corresponds to $3{\times}2$ squares. We select the longest axis as splitting axis and the split offset at the ceiled center along that axis, yielding the splitting plane $x = \lceil \frac{3}{2} \rceil = 2$. The contained points are then assigned to either the left or right child (by in-place sorting, analog to quick-sort). During the assignment, we also compute the children's bounding planes along the splitting axis, round them to the appropriate boundaries ($x_{\text{left}} = 1$ and $x_{\text{right}} = 2$) and then store them in the parent node G. The empty region of $1{\times}2$ squares inbetween the children is therefore implicitly skipped – this represents the advantage of a BIH compared to a k-d tree. After splitting the root node G, we obtain the central image. The operation is recursively applied to both children L and R. Empty leafs are indicated by invalid bounding planes in the parent node and thus do not need to be explicitly stored as tree nodes, e.g., the right child of node R.

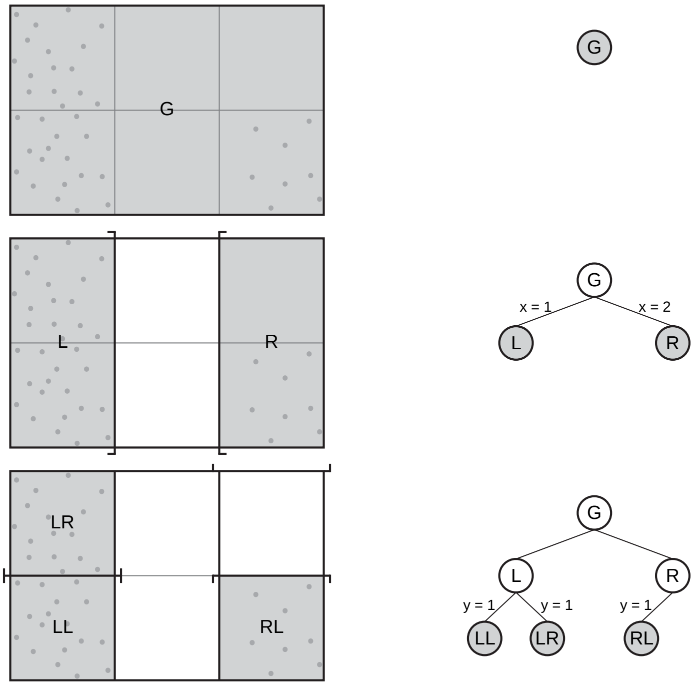


Figure 5.2: Constructing a hierarchical BIH subdivision on top of the $3{\times}2$ grid from Figure 5.1.

The resulting compact BIH tree consists of non-empty leaf nodes and approximately the same number of inner nodes (exactly $N-1$ inner nodes, N being the number of leafs, in case every inner node has exactly 2 children, i.e., if there are no empty leafs). At the same time, the points are sorted in a special way so that all points contained by a leaf or inner node form a sequential subset of the global set.

5.3 Normalizing Lambda

In the previous chapter, we have seen that the square of Duchon's semi-norm of order p in a finer scale $(j-1)$ is scaled by 2^{d-2p} compared to the squared semi-norm in scale j. The size of the B-spline intervals, relative to the full volume, is not only defined by a cube's resolution level, but also by the size of a cube relative to the full volume, i.e., by the parameter s. Analog to increasing the resolution, dividing the cube edge length by s results in $\mathcal{M}_p(S)$ being scaled by s^{d-2p}.

Let us introduce a parameter λ_p^{n} defining the actual regularization factor λ_p and used for a global smoothness constraint independent from chosen resolution and subdivision. λ_p^{n} specifies the weight of the regularization term $\mathcal{M}_p(S)$ assuming a single cube in resolution level 0 (one inner B-spline interval along the volume's longest axis). Using this reference, the actual regularization factor λ_p for resolution level l and a subdivision into s cubes (along the volume's longest axis) is given by:

$$\lambda_p = \lambda_p^{\mathrm{n}} \cdot (2^l s)^{2p-d}, \tag{5.1}$$

thereby neutralizing the implicit scaling of $\mathcal{M}_p(S)$. So using the same λ_p^{n} for different cube sizes and resolution levels yields a constant global smoothness constraint.

5.4 Block-Based Reconstruction

After subdivision, we reconstruct a multi-scale B-spline pyramid for each cube. As required by our fusing scheme, the cube pyramids start at a common coarsest resolution level l_{min}. The finest resolution level l_{max} for a cube is either fixed or can be adapted to the point samples density inside the cube.

A fine subdivision leads to many small cubes. Instead of reconstructing each cube separately, we merge cubes to larger blocks and reconstruct these blocks. This increases accuracy and decreases the computational overhead due to less redundant outer coefficients and paddings. Based on the cubes BIH, we merge an inner node's subtree to a block if:

- the total block volume V_{block}, in B-spline coefficients at maximum resolution (the maximum l_{max} of all contained cubes), is less than a threshold (thereby limiting RAM requirements), and
- the ratio $\frac{\sum V_{\text{cube}}}{V_{\text{block}}}$ is greater than another threshold. $\sum V_{\text{cube}}$ is the aggregate volume of all contained non-empty cubes (each at its maximum resolution l_{max}). This constraint is therefore used to merge only subtrees which contain no or only very few/small holes

and where the finest cube's maximum resolution $l_{\max}$ corresponds to the majority of other cubes' $l_{\max}$.

Figure 5.3 illustrates a 2D example for the merging of cubes (squares) to blocks for reconstruction.

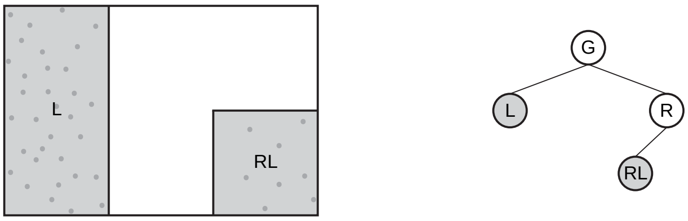


Figure 5.3: Merging squares (from Figure 5.2) to larger blocks for reconstruction. For this trivial example, only the LL and LR squares are merged to the larger block L.

To reconstruct a block by using some padding as motivated in the previous chapter, we also need to include point samples lying just outside the block. The samples potentially influencing the padded block can therefore be safely restricted to the union of all samples contained by the block itself and its neighboring cubes.

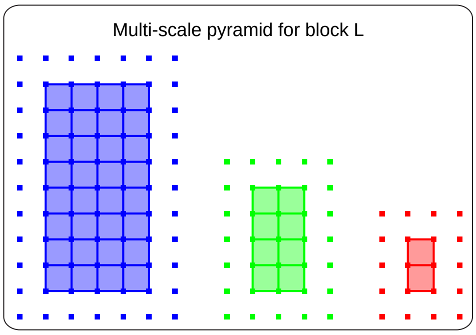


Figure 5.4: Example of a final (non-padded) multi-scale pyramid for a block consisting of three resolution levels (red: level 0, green: level 1, blue: level 2).

After reconstructing a block multi-scale pyramid (see Figure 5.4), it is split into a corresponding multi-scale pyramid for each contained cube. Once all blocks have been reconstructed and split into cubes, the cube pyramids can be fused in each scale using our single-scale fusing algorithm presented in the previous chapter.

5.5 Deriving a Level-of-Detail

Once we have a multi-scale pyramid for each cube, we may derive different Levels-of-Detail (LoDs). A common error measure associated with reconstructed volumes based on non-uniform samples is the Root-Mean-Square (RMS) error defined as:

$$RMS = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}[S(\mathbf{x}_i) - f_i]^2}, \tag{5.2}$$

assuming the values f_i of the N samples are in a normalized range $[0, 1]$. It is therefore a measure for the accuracy of the approximation at the point samples.

We developed a simple heuristic aiming at minimizing the RMS error for a specific memory budget/minimizing the memory requirements for a target RMS error. Firstly, all scales of each cube are associated with a local RMS error (taking into account only point samples inside the cube). The LoD is initialized using the coarsest scale for each cube. We then have the option to replace a cube's scale with a finer one, thereby decreasing the local and total RMS errors. Our goal, the minimization of the total RMS error, is equivalent to minimizing the total sum of squares:

$$\sum_{i=0}^{N-1}[S(\mathbf{x}_i) - f_i]^2 = \sum_{\text{cubes}} N_{\text{cube}}\, RMS^2_{\text{cube}} \tag{5.3}$$

If we denote the local RMS error of a cube i in scale j with $RMS_{i,\,j}$ and replace a cube's scale j with a finer scale $(j-1)$, the total sum of squares will decrease by $N_i\ (RMS^2_{i,\,j} - RMS^2_{i,\,j-1})$. So we generate a list of options (all finer scales of every cube) and associate each with a vote defined as sum-of-squares decrease per additional B-spline coefficient, compared to the cube's base scale j:

$$Vote_{i,\,\delta_j} = \frac{N_i\ (RMS^2_{i,\,j} - RMS^2_{i,\,j-\delta_j})}{V_{i,\,j-\delta_j} - V_{i,\,j}}. \tag{5.4}$$

The options are sorted descendingly by this vote and then applied (replacing the cube's current scale with the option's) if the memory budget is not exceeded. We optionally terminate the refinement as soon as the total RMS error falls below a target value. The minimum RMS error is obviously bounded by the cubes' finest scales though. A multi-pass approach would allow to extend a cube's multi-scale pyramid iteratively by a finer scale until the local RMS error satisfies a given threshold.

After selecting a scale for each cube, we need to take care of boundary continuity between adjacent cubes of differing scales. So we apply the interscale fusing scheme presented in the previous chapter. The replacement of fine coefficients with expanded coarser ones leads to a slight degradation of the total RMS error. Afterwards, the cubes can again be merged to blocks, similar to the merging to blocks for reconstruction. The merging results in a flatter BIH and less memory overhead due to less redundant outer coefficients (see Figure 5.5).

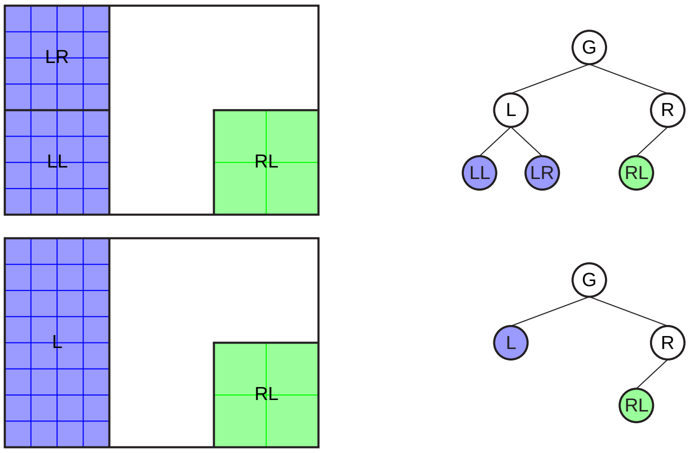


Figure 5.5: To derive a LoD, we first select a scale for each cube (square). After interscale fusion, the cubes (squares) are merged to larger LoD blocks. For this trivial example, the LoD blocks correspond to the blocks for reconstruction.

CHAPTER 6

GPU Ray Casting

So far, we have derived a volume LoD consisting of a BIH tree of blocks and a coefficients grid for each autonomous block. We now focus on how to project the LoD onto a 2D image plane by exploiting the GPU to allow for interactive visual exploration of the volume.

6.1 Our Variant of Direct Volume Rendering

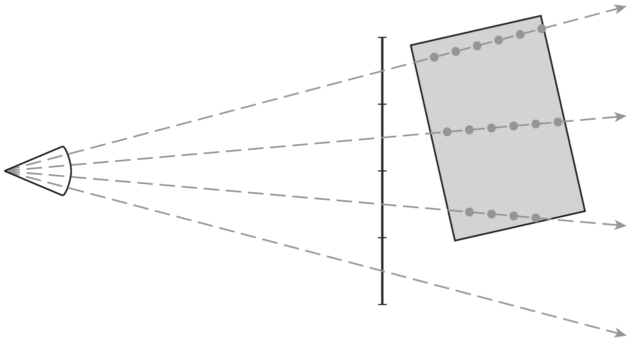


Figure 6.1: Ray casting schematics using a 2D example for 2D → 1D central projection. For each pixel (represented by the 4 vertical line segments), a ray is cast from the eye (left) through the pixel center. Ray segments intersecting the volume (indicated by the box) are then sampled (usually in front-to-back order for early ray termination) and the samples somehow combined to obtain a color for each pixel.

Direct Volume Rendering (DVR) consists in casting rays on the GPU. For each pixel, a ray is cast from the eye through the pixel center and the ray segment intersecting the volume is sampled equidistantly (see Figure 6.1). Traditional DVR uses a 3D texture to represent the volume and exploits the GPU's trilinear filtering capabilities to interpolate between the voxels – i.e., the 3D texture is treated as coefficients grid of a uniform trilinear B-spline.

Instead of trying to combine a ray's scalar samples directly to a pixel color, DVR uses a so-called Transfer Function (TF) mapping a scalar to a color (r, g, b) and an associated opacity $\alpha \in [0, 1]$. This allows for flexible renderings because modifications to the TF may yield completely different visual results, allowing to explore a volume interactively. DVR uses the emission-absorption model to allow for a translucent view into the volume: at each ray sample, the B-spline is evaluated and the resulting scalar mapped to (r, g, b, α). The sample absorbs an amount α of the light behind it (i.e., transmitting an amount of $(1 - \alpha)$) and emits the same amount α of new light (in the simplest form, directly its color, hence emitting $\alpha \cdot (r, g, b)$). To compute the light reaching the eye along a ray, the ray is usually sampled in a front-to-back manner and the samples accumulated like this:

$$(r, g, b, \alpha)_{\text{pixel}} := (r, g, b, \alpha)_{\text{pixel}} + (1 - \alpha_{\text{pixel}})\ \alpha_{\text{sample}}\ (r, g, b, 1)_{\text{sample}}$$

until α_{pixel} reaches a threshold near 1 (early ray termination). Note that for correct α-blending, the colors need to lie in a linear color space, so colors in the widely used sRGB color space need to be gamma-corrected.

For the generation of transfer functions, we use a cardinal cubic spline to smoothly interpolate a set of control points, each associated with color and opacity. The result is discretized into a 1D texture (we use 512 RGBA16 texels, 4 KB). The GPU renderer then performs a linear lookup for each ray sample. Figure 6.2 shows an example.

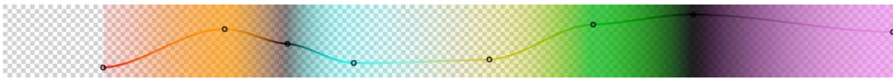


Figure 6.2: Example for a transfer function (TF). The x axis represents the signal range $[0, 1]$, the y axis the opacity $\alpha \in [0, 1]$.

Generating the Rays

We render the bounding box (12 triangles) of the volume as proxy geometry. The vertex shader projects the vertices into clip-space and computes the eye-space position of each vertex. This position is interpolated across a triangle's rasterized fragments and used as input for the fragment shader, where it is normalized and used as eye-space ray direction. To make sure that a pixel is covered by at most a single fragment, we only rasterize back-facing triangles. We then compute the ray segment intersecting the volume via algorithm 3.

Using the Gradient

The B-spline's gradient, i.e., the vector of all first partial derivatives, can be used to provide additional visual hints. For example, a sample's gradient can be treated as scaled normal vector of

Algorithm 3 Intersecting a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ ($\mathbf{o}$ being the ray origin and $\mathbf{d}$ the direction) with an axis-aligned bounding Box (defined by the Min and Max corners), yielding the $[t_{\text{in}}, t_{\text{out}}]$ interval for the ray equation.

```
$\mathbf{t}_{\text{min}} := \frac{Box.Min-\mathbf{o}}{\mathbf{d}}$
$\mathbf{t}_{\text{max}} := \frac{Box.Max-\mathbf{o}}{\mathbf{d}}$
$\mathbf{t}_{\text{near}} := \min(\mathbf{t}_{\text{min}}, \mathbf{t}_{\text{max}})$
$\mathbf{t}_{\text{far}} := \max(\mathbf{t}_{\text{min}}, \mathbf{t}_{\text{max}})$
$t_{\text{in}} := \max(0, t_{\text{near},x}, t_{\text{near},y}, t_{\text{near},z})$
$t_{\text{out}} := \min(t_{\text{far},x}, t_{\text{far},y}, t_{\text{far},z})$
```

the sample's iso-surface so as to allow to shade ray samples, giving the viewer more information about the volume on the 2D projection. We use the Blinn-Phong lighting model for shading and ignore the direction (sign) of the unit-length gradient, i.e., we do not differentiate between front- and back-side of the iso-surface. To differentiate between regions with stronger gradients and more homogeneous regions, the color used for a ray sample is a smooth linear combination of the unshaded and shaded colors depending upon the gradient magnitude. Regions with stronger gradients can be further emphasized by increasing the sample's opacity with the gradient magnitude. Algorithm 4 summarizes our implementation.

Algorithm 4 Computing a ray sample's contribution (r, g, b, α) for DVR. $\mathbf{x}$ denotes the sample location for the block's B-spline S (in a given resolution $level$) sampled using a given $stepLength$ (relative to a reference step length). $ShadingRefMag$ and $AlphaRefMag$ are parameters denoting the minimum gradient magnitude for a fully shaded sample/for the maximum sample opacity; α_{base} and α_{gradient} determine a sample's final opacity. lerp() denotes linear interpolation, smoothstep() a smoothly increasing weight $\in [0, 1]$, and both correspond to HLSL functions.

```
evaluate the B-spline: $s := S(\mathbf{x})$
map to color and opacity: $(r, g, b, \alpha) := TF(s)$
if $\alpha > 0$ then
    compute the resolution-normalized gradient: $\mathbf{g} := \nabla S(\mathbf{x}) \cdot 2^{level}$
    compute the gradient magnitude: $m := \|\mathbf{g}\|$
    compute the shaded contribution $(r, g, b)_{\text{shaded}}$ by using $-\frac{\mathbf{g}}{m}$ as normal vector
    // smoothly interpolate between unshaded and shaded colors depending upon the gradient magnitude $m$:
    $(r, g, b) :=$ lerp($(r, g, b)$, $(r, g, b)_{\text{shaded}}$, smoothstep($0, ShadingRefMag, m$))
    // modulate the opacity depending upon $m$ too:
    $\alpha := \max(0, \min(1,\ \alpha \cdot (\alpha_{\text{base}} + \alpha_{\text{gradient}} \cdot$ smoothstep$(0,\ AlphaRefMag,\ m))))$
    // perform opacity correction to decouple overall translucency from the sampling step length:
    $\alpha := 1 - (1 - \alpha)^{stepLength}$
end if
```

6.2 Adapting the Cubic B-Spline Evaluation to GPUs

The evaluation of a uniform cubic B-spline in 3D (Eq. 2.9) consists of a linear combination of $4^3 = 64$ coefficients. Sigg and Hadwiger [16] reformulated it as linear combination of $2^3 = 8$ trilinearly interpolated coefficients. This scheme is thus very well suited for GPUs with their hardware trilinear filtering capabilities and reduces the 64 nearest-neighbor texture lookups to 8 trilinear ones (for each ray sample). If the gradient is used, we also need 8 trilinear lookups for each partial derivative, resulting in a total of 32 trilinear lookups for each ray sample.

The lookup offsets and the weights (for the linear combination of the interpolated coefficients) can either be computed directly in the shader program or stored in a lookup-table texture. We have chosen the lookup-table as the shader is already complex enough. We need two lookup-tables: one for the 1D B-spline and another one for its first derivative (to compute the gradient). We use 512 RGBA16 texels (4 KB) for each table and perform linear lookups.

6.3 Mapping the Blocks into a Single 3D Texture

By employing the aforementioned acceleration scheme, we cannot simply serialize the block coefficient grids of a LoD into a flat buffer – to allow for trilinear filtering, the blocks need to retain their 3D structure. Since current GPUs do not support indexable 3D texture arrays, we need to pack the blocks into a single 3D texture while trying to waste as little space as possible. We chose the half-precision R16 texel format for the coefficients (2 bytes per coefficient).

To restrict the complexity and allow simple indexing of the blocks, we will not optimize the orientation of the blocks (e.g., rotating by 90 degrees about a primary axis). Instead, we focus on empty cuboid regions in the texture and try to fit blocks into these holes or, if no existing hole is large enough, try to extend the most appropriate hole.

The developed heuristic begins by initializing the texture size (N_x, N_y, N_z) with the maximum extents of all blocks, along each axis. The blocks are then sorted descendingly by their longest extent $\max(size_x, size_y, size_z)$, then descendingly by their volume in order to sort them by their probable influence on the fitting process. Afterwards, the blocks are inserted one after the other by using algorithm 5.

Note that the heuristic is not simply based upon a greedy minimization of the texture size. We also try to favor a compact cubic shape of the texture to prevent the texture from growing along a single axis, which would lead to small hole extensions along other axes to result in large volume increases (hence penalizing these hole extensions and extending the texture further along the dominant axis instead). The weight of the shape for the vote can be adjusted by raising the ratio to an arbitrary power.

Identification of Holes

The most important step of the algorithm is the identification of holes in the texture. We want to work with cuboid holes for the cuboid blocks, so the first step consists in identifying all disjoint empty cuboid regions in the texture. We achieve this by subdividing the texture into a k-d tree; all nodes of a k-d tree are cuboids and all nodes of a given tree depth are disjoint, hence each empty leaf represents a disjoint cuboid hole. Then we try to combine adjacent holes (see Figure 6.3). This is accomplished by traversing the k-d tree in a bottom-up manner. For each inner node, we select all left holes adjacent to the splitting plane as left candidates and similarly all adjacent right holes as right candidates. For each combination of left and right candidates, we try to combine the pair to obtain a new hole and keep it if it is not already fully contained by another new hole combination. After processing all pairs, the left and right candidates are scanned for

Algorithm 5 Mapping a block into the 3D texture.

```
for each of the global insertion candidates $\{(N_x, 0, 0),\ (0, N_y, 0),\ (0, 0, N_z)\}$ do
    compute the required texture size $(N_x^{\text{new}}, N_y^{\text{new}}, N_z^{\text{new}})$ if the block was inserted at the global candidate
    compute the ratio $\frac{\min(N_x^{\text{new}}, N_y^{\text{new}}, N_z^{\text{new}})}{\max(N_x^{\text{new}}, N_y^{\text{new}}, N_z^{\text{new}})} \in (0, 1]$, a measure for the cubicness of the new texture shape
    divide the required texture volume $N_x^{\text{new}} N_y^{\text{new}} N_z^{\text{new}}$ by above ratio to obtain a vote for the global candidate
end for
select the global candidate with the smallest vote as best global candidate

identify holes in the texture
for each hole do
    compute the required extension of the hole: $\delta = \max((0, 0, 0),\ size_{\text{block}} - size_{\text{hole}})$
    compute the probably required texture size: $(N_x^{\text{new}}, N_y^{\text{new}}, N_z^{\text{new}}) = (N_x, N_y, N_z) + \delta$ and the corresponding vote
    compute the fill-ratio of the extended hole: $\frac{V_{\text{block}}}{V_{\text{extended hole}}}$
end for
sort the hole candidates ascendingly by their vote, then descendingly by their fill-ratio, and select the first one as best hole candidate

if the best hole candidate's vote $\leq$ the best global candidate's then
    insert the block at the minimum corner of the hole
    if the hole had to be extended then
        shift blocks overlapping with the new block as well as blocks overlapping with newly shifted ones
        compute the smallest AABB containing all blocks and set it as new texture size $(N_x, N_y, N_z)$
    end if
else
    insert the block at the global candidate
    set the new texture size to the candidate's $(N_x^{\text{new}}, N_y^{\text{new}}, N_z^{\text{new}})$
end if
```

redundant ones (fully contained by a new hole combination). The holes of an inner node will then be the union of all non-redundant left and right holes as well as the new non-redundant hole combinations. We prefer the k-d tree over a BIH here since k-d trees are simpler: inner nodes always split the volume into two children (binary space partitioning), whereas an inner BIH node may split it into two children and an empty region inbetween them.

The depth of the k-d tree depends upon how we select the splitting planes. Intuitive candidates are all bounding planes of each contained block as long as they lie inside the node, thus splitting the node into two non-empty volumes. For each distinct candidate, we perform a hypothetical split and assign blocks to the left and/or right side (overlapping blocks need to be assigned to both children). At the same time, we compute the bounding planes of both children

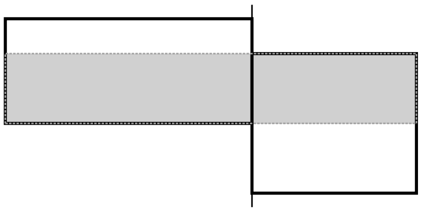


Figure 6.3: Combining 2 adjacent holes to a new cuboid hole (grey).

along the splitting axis, analog to constructing a BIH (see Figure 5.2). We then select the candidate which results in the smallest sum of both children's extents along the splitting axis, i.e., we try to minimize the children's overlap/maximize the empty space inbetween the children.

Shifting Overlapping Blocks

The other important aspect of the mapping algorithm regards shifting existing blocks when a hole is extended for a new block. To make sure there are no overlapping blocks, it would be sufficient to process each shifted bounding plane of the hole by shifting all blocks beginning at or after the hole's original bounding plane by the δ along the perpendicular axis. So a block lying on the right-hand-side of multiple shifted bounding planes would be shifted along multiple axes.

What we want though is to concentrate the blocks at the minimum corner of the texture and thereby favor a concentration of holes at the texture's right-hand-side borders. This is compatible with our choice for the global insertion candidates and leads to larger holes at the right-hand-side borders, especially at the maximum corner of the texture (being the intersection of the right-hand-side borders of each dimension). Additionally, we can restrict the translation to at most a single axis for each block by using algorithm 6.

Algorithm 6 Shifting overlapping blocks.

```
sort the shifted bounding planes of the hole ascendingly by their $\delta$
for each of these bounding planes do
    create a list of shifted blocks and initialize it with the newly inserted block
    identify all blocks lying on the right-hand-side of the original bounding plane and sort them ascendingly by their distance
    from the original bounding plane
    for each of these blocks do
        if the block overlaps with any of the shifted blocks then
            shift the block by the current $\delta$ along the axis perpendicular to the current bounding plane
            add the block to the shifted blocks list
        end if
    end for
end for
```

6.4 Traversing the BIH

BIH Memory Layout

An inner BIH node is defined by the index of the splitting axis $\in \{0, 1, 2\}$, the bounding plane offsets for both children and a reference to its children. By storing the tree in pre-order manner (root, left subtree, right subtree), an inner node's left child will be stored directly after the parent, requiring only a reference to the right child. A compact memory layout for inner nodes consists of two 16-bit bounding plane offsets, a 30-bit index for the right child and 2 bits for the splitting axis, partitioned into four 16-bit components. This is a GPU-friendly format, allowing to represent an inner node as RGBA16 texel (8 bytes).

Leaf nodes only need to indicate that they are leafs and may contain arbitrary attributes. We choose an identical layout for leaf nodes ($4 \cdot 16$ bits) and denote leafs by an invalid splitting axis

(3, which still fits into the 2 reserved bits). The other $(3 \cdot 16 + 14)$ bits encode the block's offset in the 3D texture as well as the block's resolution level.

Traversal Algorithm

To sample a ray segment inside a volume LoD, we need to iterate through the blocks hit by the ray and sample each block's autonomous B-spline, all in front-to-back order. To identify blocks hit by a ray, we have to traverse the LoD's BIH recursively. Algorithms 7 and 8 show how to do this by employing a per-ray stack whose capacity must not be smaller than the BIH depth.

Algorithm 7 Traversing the BIH tree for a given ray.

```
push the root node onto the empty stack
while there are nodes on the stack, i.e., subtrees yet to be traversed do
    pop the topmost node from the stack, representing the root of the deepest subtree
    while the node is an inner node do
        intersect the ray with the children
        if no child is hit then
            continue the outer loop with the next subtree
        else if only one child is hit then
            descend to that child
        else
            push the second hit child onto the stack
            descend to the first hit child
        end if
    end while
    render the leaf by sampling the block's B-spline along the ray segment
end while
```

Algorithm 8 Intersecting the children of an inner BIH node with a given ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ ($\mathbf{o}$ being the ray origin and $\mathbf{d}$ the direction). The node's axis-aligned bounding Box (defined by the Min and Max corners) as well as the entry and exit distances of the ray (t_{in} and t_{out}) are known (from the top-down traversal). a denotes the node's splitting axis, $PlaneOffsets$ the children's bounding plane offsets.

```
// compute the children's AABBs
$Box_0 := Box; Box_0.Max_a := PlaneOffsets_0$
$Box_1 := Box; Box_1.Min_a := PlaneOffsets_1$
// intersect the children's bounding planes with the ray
$t_0 := \frac{PlaneOffsets_0 - o_a}{d_a}$
$t_1 := \frac{PlaneOffsets_1 - o_a}{d_a}$
// determine the traversal order depending upon $\mathbf{d}$
$first := (d_a \geq 0\ ?\ 0 : 1)$
$second := 1 - first$
// compare the first child's intersection to the entry point of the parent
if $t_{first} > t_{\text{in}}$ then
    the first child enclosed by $Box_{first}$ is hit
    its entry distance is still $t_{\text{in}}$, its exit distance $\min(t_{\text{out}}, t_{first})$
end if
// compare the second child's intersection to the exit point of the parent
if $t_{second} < t_{\text{out}}$ then
    the second child enclosed by $Box_{second}$ is hit
    its entry distance is $\max(t_{\text{in}}, t_{second})$, its exit distance still $t_{\text{out}}$
end if
```

6.5 Adaptive Sampling

If a volume is represented by blocks of differing resolution levels, it may be useful to adapt the sampling step to each block's resolution level, thereby sampling lower-frequent coarser blocks more coarsely than higher-resolution ones and thus increasing rendering performance.

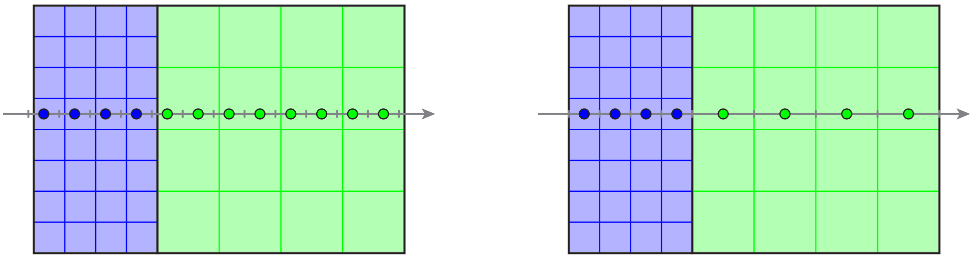


Figure 6.4: Equidistant ray sampling (left) and adaptive ray sampling (right) of a volume consisting of a fine left block (blue) and a coarse right block (green).

Figure 6.4 illustrates the difference between equidistant and adaptive ray sampling. In traditional DVR, the ray is sampled at equidistant steps. Each ray sample is treated as constant ray segment of a length corresponding to the sampling step, leading to a step-wise approximation of the rendering integral along the ray. For adaptive sampling on the other hand, we first select a sampling step length according to the block's resolution level (in the depicted example, the step length is set to the B-spline interval length, corresponding to 1x adaptive sampling). The whole ray segment inside the block is then sampled equidistantly by subdividing it into a corresponding number of smaller, uniformly sized segments. The sampling step length may be slightly decreased to accomodate for an integral number of small segments. In the depicted example, the ray is finally sampled at the center of each small segment.

The sampling step determines the accuracy of the step-wise approximation of the rendering integral. For DVR, this means that even if both the signal and the TF are composed of very low frequencies, we still need a rather high number of samples for visually pleasing results. Visual results for coarse sampling steps could be improved by techniques such as pre-integrated radiance transfer, where the rendering integral is precomputed for a small segment (of given length) by treating it as linear piece (instead of a constant one), yielding 2D transfer functions (for all combinations of left and right ray samples). Unfortunately, advanced transfer functions involving shading and gradient-based opacity lead to an infeasible number of degrees of freedom for pre-integration techniques.

Adaptive sampling therefore makes more sense for other rendering techniques such as iso-surface rendering, where the first intersection of a ray with a given iso-surface is rendered. In this case, we sample the ray until two consecutive samples lie on opposite sides of the iso-value. The location where the signal corresponds to the iso-value is then linearly refined by iterative interval shrinking. The required sampling accuracy (in order not to miss the first intersection) decreases for lower-frequent signals, and hence adaptive sampling may be employed to increase rendering performance. For iso-surface renderings, we sample the ray at the joints of the small segments instead of their centers in order not to miss intersections in the first half of the first segment and in the second half of the last segment.

Part II

Results

CHAPTER 7

Prerequisites

7.1 Test System

For our performance figures to make sense, we need to specify the key features of the employed hard- and software platform:

- CPU: AMD Phenom X4 9950, 4 cores @ 3.1 GHz
- RAM: 4 GB DDR2 @ 1,000 MHz
- GPU: ATI Radeon HD 5850 equipped with $18 \cdot 16$ 5D stream processors @ 900 MHz and 1 GB GDDR5 memory @ 1,150 MHz
- OS: Windows 7 x64

The key algorithms have been developed for the x64 platform using C++ and the Visual Studio 2010 compiler. To provide for good performance, we exploit parallelization (using OpenMP), SSE instructions and templated code. For rendering, we used the Direct3D 11 API and developed the GPU renderer as fragment shader in HLSL, targeting shader model 4 and thus supporting Direct3D 10 compatible hardware.

7.2 Data Sets

Analog to Vuçini et al. [18], our 3D data sets are based on true non-uniform point sets as well as on Cartesian grids. Most of the data sets have in fact been provided by Dr. Erald Vuçini; refer to his thesis for detailed descriptions [17]. Some additional Cartesian grids have been provided by Dr. Stefan Röttger's Volume Library (*http://www9.informatik.uni-erlangen.de/External/vollib/*). For Cartesian grids, we extract a percentage of the grid samples, selected based on their Laplacians – i.e., we approximate the Laplacian by computing the difference between a sample and the average of its 6 nearest neighbors (left and right neighbors along each dimension) and extract the samples with the highest absolute Laplacians. These Laplacian data sets are useful to assess the quality of the reconstructed volume globally since the original Cartesian grid represents the ground truth for comparison. Analog to Vuçini et al. [18] as well as Morozov et al. [14], we use an additional global RMS_g error measure for the Laplacian data sets, defined as the root-mean-square error at each original grid sample. The sample values f_i are normalized into the $[0, 1]$ range by dividing by the maximum value (for Laplacian data sets: by the maximum value in the original grid).

CHAPTER 8

In-Depth Reconstruction Analysis

In this chapter, we are going to analyze some key aspects of the reconstruction process. We do not tackle block-based reconstruction and subdivision yet.

8.1 Regularization

Tables 8.1 and 8.2 show some exemplary cubic 1D reconstruction results to illustrate the difference between Duchon's semi-norms of order $p = 1$ and $p = 2$, the smoothening impact of higher λ_p^{n} parameters as well as the relation with the chosen resolution. We see that the order of the semi-norm has a huge impact in regions where the point samples density is low because these regions are primarily defined by the regularization constraint, so that the difference is more notable in Table 8.2. For $p = 1$, the regularization aims at minimizing the gradient, whereas for $p = 2$, the aim is to keep the gradient constant.

By using a lower λ_p^{n} parameter, the B-spline's regularization constraint is relaxed and the signal fit more closely to the point samples. The RMS error therefore generally decreases with lower λ_p^{n}. Higher λ_p^{n} values are useful if the point samples are noisy and it is desirable to smooth the resulting B-spline, i.e., to perform a low-pass filtering of the signal's frequencies. A B-spline's resolution limits its frequencies, so higher λ_p^{n} values for coarse resolutions lead to more appropriate low-frequent signals. This observation is of key importance in the common case in which a band-limited signal is to be reconstructed. For our examples, we want the resulting signals to lie in the normalized $[0, 1]$ range. Exceedances of this range cannot be penalized via a quadratic cost function. As work-around, we try to minimize these exceedances by enforcing a higher smoothness via a higher λ_p^{n} value, especially for coarser resolutions.

Tables 8.3 and 8.4 give an overview of corresponding cubic 3D results for two challenging, very sparse Laplacian data sets. We see that while lower λ_p^{n} values generally lead to lower RMS errors, the global $\mathrm{RMS_g}$ errors at some point strongly increase due to exceedances of the $[0, 1]$ range in sparsely sampled regions. These regions are the borders, since both data sets contain quite a lot of empty/noisy padding. By selecting only 5% of the grid samples based on their

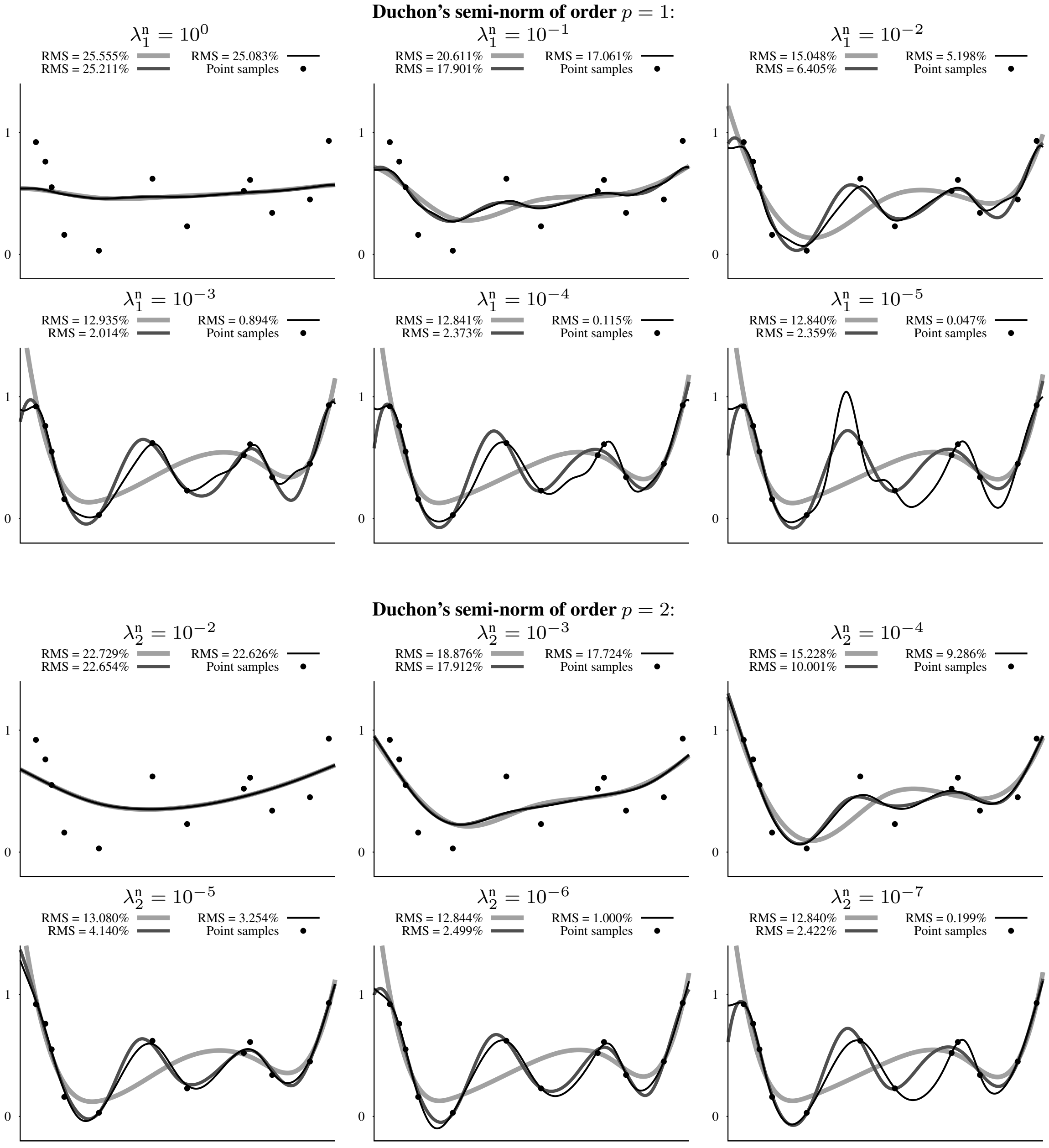


Table 8.1: Single-scale cubic 1D reconstruction results for three resolutions (4 (light-grey)/ 8 (dark-grey)/16 (black) inner B-spline intervals) and λ_p^n values of decreasing order of magnitude.

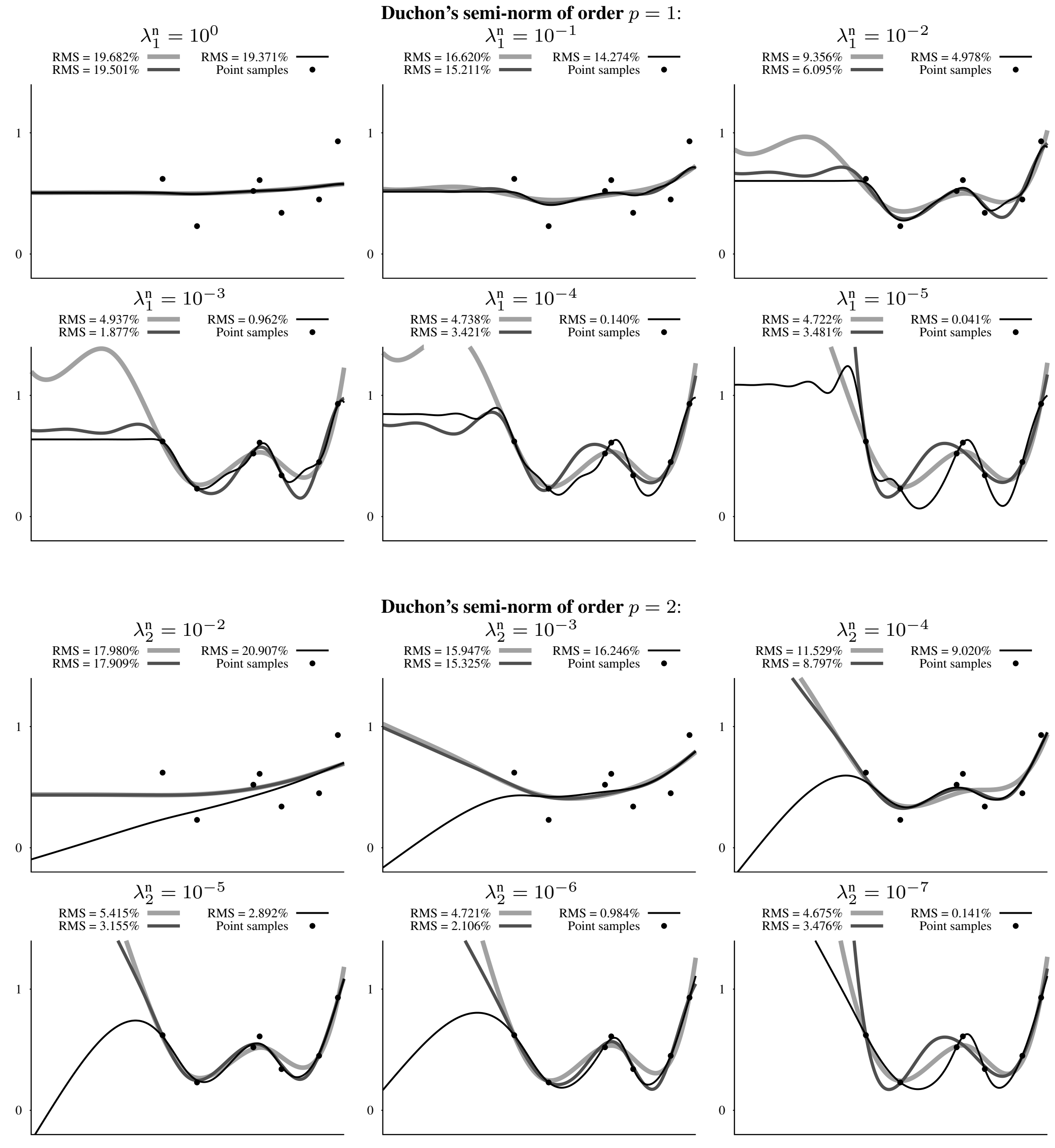


Table 8.2: Single-scale cubic 1D reconstruction results for three resolutions (4 (light-grey)/ 8 (dark-grey)/16 (black) inner B-spline intervals) and λ_p^{n} values of decreasing order of magnitude.

Laplacians, the extracted point cloud is concentrated at the actual object in the center, and we have large empty border regions, leading to situations comparable to Table 8.2. Here, it would obviously make more sense to either trim the reconstructed volume to the bounding box of the extracted point samples or to extract the samples randomly, leading to a more uniform samples distribution. Analog to Vuçini et al. [18], we clip the signal into the $[0, 1]$ range when computing the RMS$_{\text{g}}$ errors. For the RMS errors, we do not use clipping. We see that lower λ_p^{n} values are of key importance to avoid/minimize visible artifacts for coarser resolutions. The visibility of the artifacts depends upon the employed transfer function.

We conclude that λ_p^{n} generally needs to be adapted for each resolution level, using higher λ_p^{n} values for coarser scales. We therefore propose to scale λ_p^{n} by an interscale factor f_λ when transitioning to a coarser scale, based on a reasonable base λ_p^{n} for the maximum resolution.

Incorporating the Lambda Interscale Factor

To derive a linear system from a finer one while modifying λ_p^{n}, we need to split $\mathbf{A} = \mathbf{D} + \lambda_p \mathbf{R}$ (Eq. 3.20) into $\mathbf{D}$ and $\mathbf{R}$. $\mathbf{D}$ depends upon the point samples but not upon λ_p^{n}, so we can simply coarsen it for coarser scales. $\mathbf{R}$ on the other hand does not depend upon the point samples and is scaled by λ_p. So we have two options: either computing $\mathbf{R}$ in the finest scale and then coarsening it separately before scaling and adding it to separately coarsened $\mathbf{D}$, or computing $\mathbf{R}$ directly for each scale. We chose the latter option because it requires a single matrix for each scale (when computing $\mathbf{R}$, we scale it by λ_p and add it directly to coarsened $\mathbf{D}$). Since the matrix is responsible for almost all of the required memory foot-print, storing a single matrix allows to nearly halve temporary RAM requirements. Deriving a coarser linear system this way retains the advantages of being independent from the samples set, scaling with the total number of coefficients and allowing for straight-forward parallelization.

Care needs to be taken when computing λ_p for a derived coarser scale if $\mathbf{D}$ and $\mathbf{b}$ have been coarsened and $\mathbf{R}$ is to be added. Based on our definition for λ_p^{n} (Eq. 5.1), λ_p needs to be scaled by 2^{d-2p} when transitioning to a coarser scale for an identical λ_p^{n}. Our coarsening scheme for $\mathbf{D}$ and $\mathbf{b}$ implies that the sum of squared errors in the cost function (Eq. 3.5) is scaled by 2^{-d}. λ_p therefore needs to be additionally scaled by 2^{-d} when transitioning to a coarser scale, so that the overall scale factor for a transition is reduced to 2^{-2p} as long as we want to use an identical λ_p^{n}. Otherwise, we scale λ_p additionally by a lambda interscale factor f_λ, yielding a final overall scale factor of $f_\lambda \cdot 2^{-2p}$ for a derived coarser scale's λ_p.

Our implementation for matrix $\mathbf{R}$ requires the integral of the squared smoothness semi-norm to start and end at a knot, i.e., restricts the integral range to full B-spline intervals. An implication of this is that the integral range is padded to an appropriate knot in case the actual signal does not start and/or end at a knot. See Figure 4.1 for corresponding examples (where the inner signal for reconstruction corresponds to the inner signal of the padded B-spline, which may be further padded when transitioning to a coarser scale). The padding of the integral range leads to a slight difference compared to a strictly coarsened matrix, i.e., the resulting signal will slightly differ near the padded boundary due to the regularization constraint being applied to the extended range. To avoid this slight imprecision, it would be sufficient to use an extended implementation which does not rely solely on the $\eta_i(q, \delta_k)$ (Eq. 3.26).

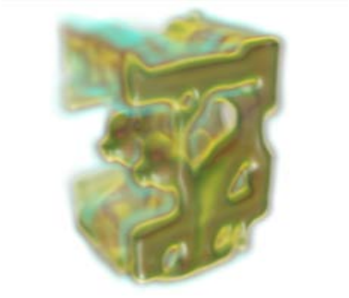
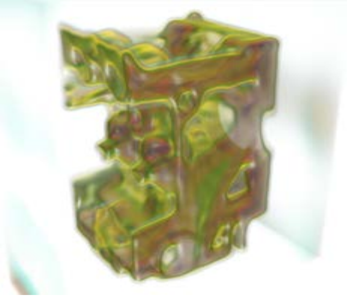
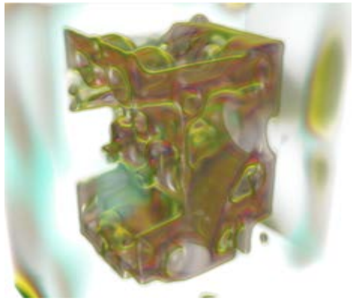
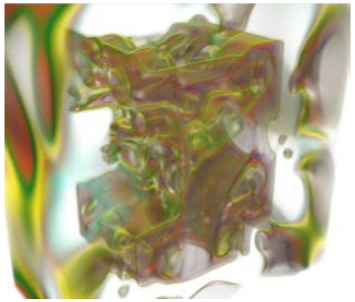
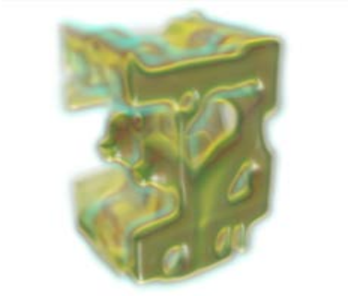
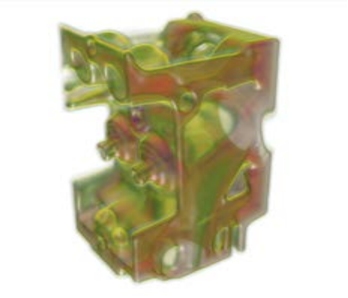
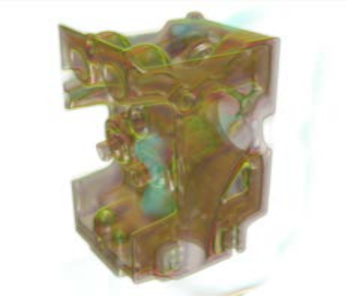
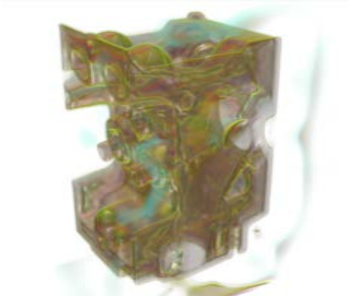
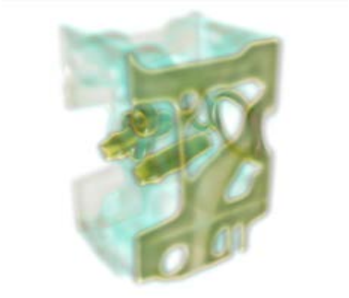
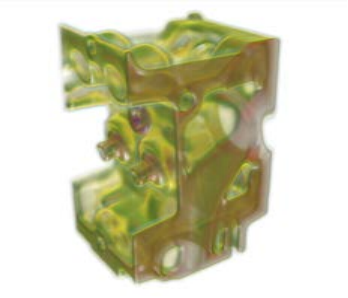
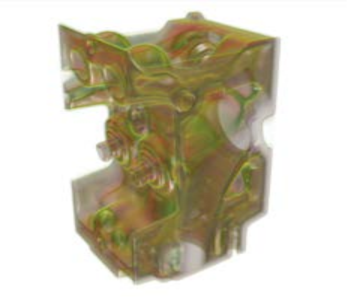
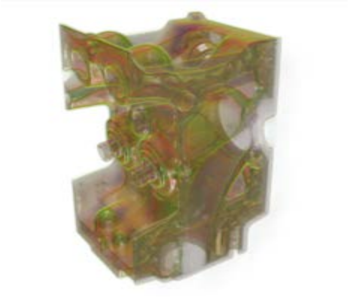
	$\lambda_2^{\mathrm{n}} = 10^{-1}$	$\lambda_2^{\mathrm{n}} = 10^{-2}$	$\lambda_2^{\mathrm{n}} = 10^{-3}$	$\lambda_2^{\mathrm{n}} = 10^{-4}$
32×32×16	RMS = 23.254%  $\mathrm{RMS_g}$ = 11.381% (13.242%)	RMS = 18.520%  $\mathrm{RMS_g}$ = 8.227% (60.738%)	RMS = 15.283%  $\mathrm{RMS_g}$ = 10.979% (196.766%)	RMS = 13.339%  $\mathrm{RMS_g}$ = 17.954% (551.935%)
64×64×32	RMS = 22.613%  $\mathrm{RMS_g}$ = 10.787% (10.903%)	RMS = 15.820%  $\mathrm{RMS_g}$ = 6.089% (16.741%)	RMS = 8.280%  $\mathrm{RMS_g}$ = 6.968% (68.710%)	RMS = 4.948%  $\mathrm{RMS_g}$ = 8.229% (105.414%)
128×128×64	RMS = 26.562%  $\mathrm{RMS_g}$ = 14.185% (14.202%)	RMS = 15.406%  $\mathrm{RMS_g}$ = 6.310% (6.814%)	RMS = 6.358%  $\mathrm{RMS_g}$ = 4.712% (12.904%)	RMS = 2.514%  $\mathrm{RMS_g}$ = 5.399% (26.125%)

Table 8.3: Single-scale cubic 3D reconstruction results for different resolutions and λ_2^{n} values using the sparse Engine (5%) data set. The $\mathrm{RMS_g}$ values in parenthesis denote the unclipped $\mathrm{RMS_g}$ values.

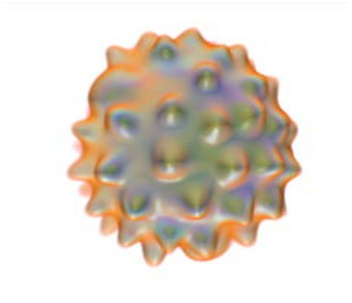
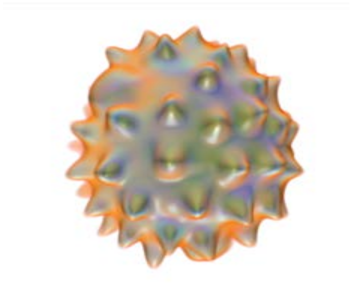
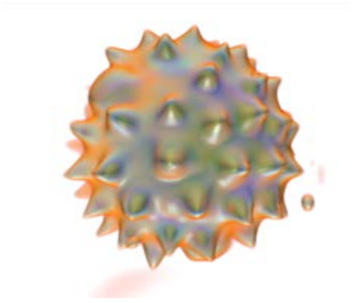
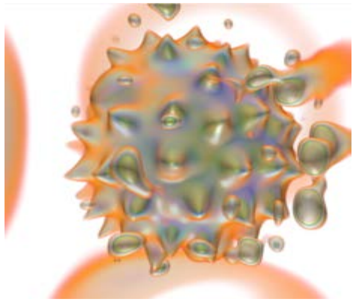
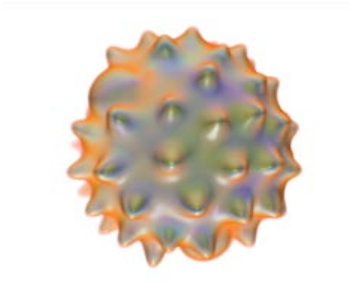
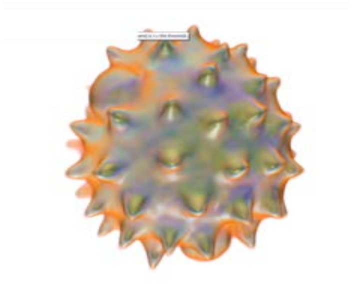
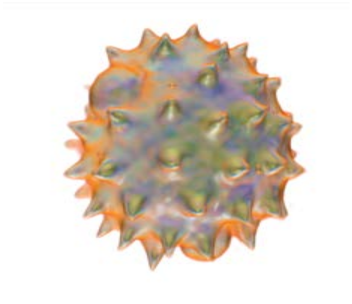
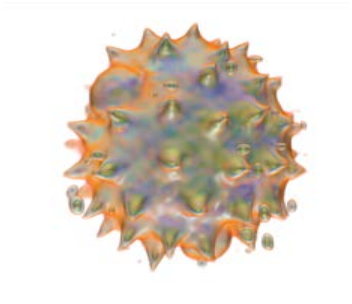
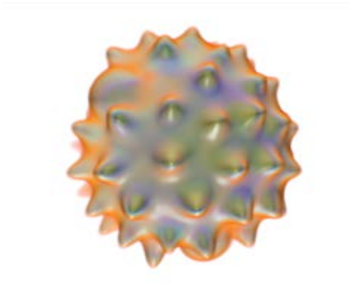
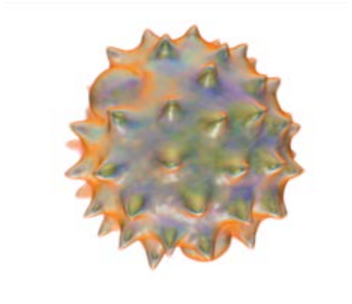
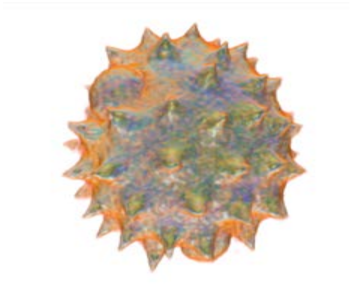
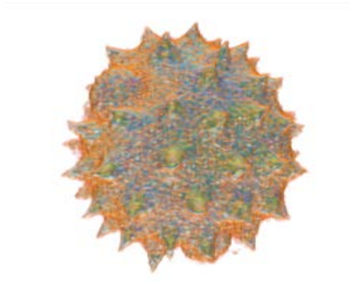
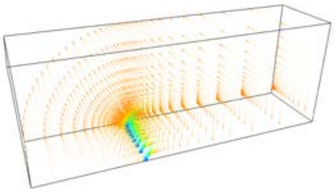
	$\lambda_2^n = 10^{-2}$	$\lambda_2^n = 10^{-3}$	$\lambda_2^n = 10^{-4}$	$\lambda_2^n = 10^{-5}$
32×30×28	RMS = 6.003%  RMS_g = 1.680% (6.660%)	RMS = 5.764%  RMS_g = 1.897% (12.853%)	RMS = 5.700%  RMS_g = 2.882% (24.635%)	RMS = 5.674%  RMS_g = 6.475% (50.302%)
64×60×56	RMS = 5.624%  RMS_g = 1.452% (2.212%)	RMS = 5.204%  RMS_g = 1.307% (4.380%)	RMS = 5.111%  RMS_g = 1.398% (8.428%)	RMS = 5.085%  RMS_g = 1.930% (17.221%)
128×120×112	RMS = 5.579%  RMS_g = 1.458% (1.481%)	RMS = 4.874%  RMS_g = 1.196% (1.526%)	RMS = 4.164%  RMS_g = 1.094% (3.070%)	RMS = 3.763%  RMS_g = 1.312% (5.041%)

Table 8.4: Single-scale cubic 3D reconstruction results for different resolutions and λ_2^n values using the sparse Daisy (5%) data set. The RMS_g values in parenthesis denote the unclipped RMS_g values.

8.2 Convergence

Iterative solvers such as the employed SOR method iteratively refine a solution estimate. The resulting signal therefore strongly depends on when the refinement is terminated, i.e., on the chosen convergence threshold for our implementation (see Algorithm 2 on page 26). The convergence rate on the other hand strongly depends on the initialization of the signal and generally decreases with higher resolutions. For instance, the diverging bottom-left result for the Engine data set in Table 8.3 is due to a premature termination of the refinement (we generally use a convergence threshold of 10% for 3D data sets; for the 1D results in Tables 8.1 and 8.2, we used 1%). See Table 8.5 for corresponding 3D results; finer scales of multi-scale pyramids are initialized with coarser results, leading to a greatly improved convergence rate as well as further extrapolation. Table 8.6 emphasizes the usefulness of a λ_p^{n} interscale factor $f_\lambda > 1$.

Blunt-Fin: 40,960 points



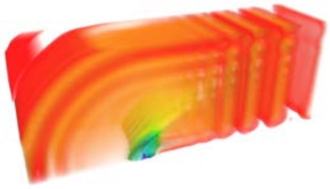
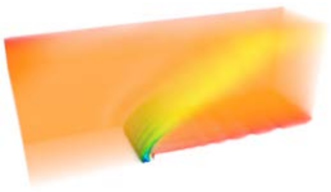
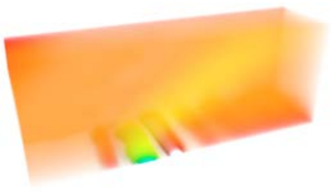
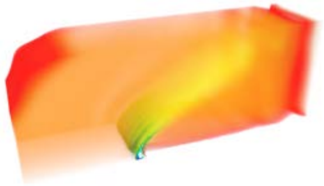
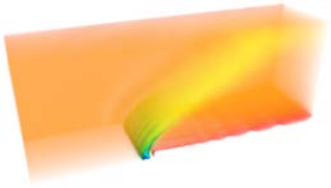
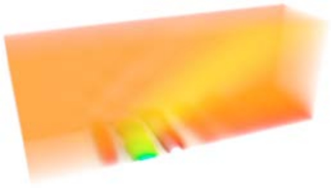
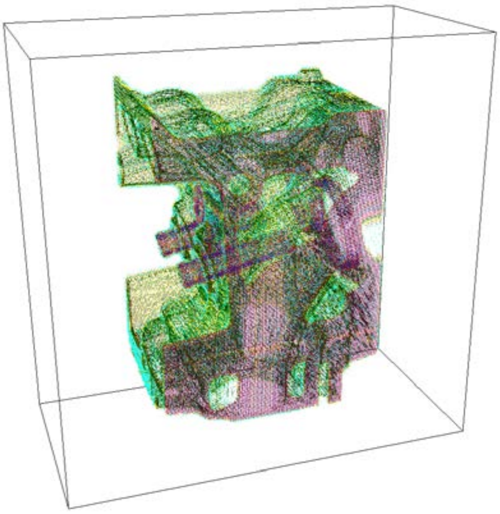
	Level 7 (128×49×34)	Levels 4-7 pyramid	
		Level 7 (128×49×34)	Level 4 (16×7×5)
10%	RMS = 1.049%  0.74 s	RMS = 0.858%  0.83 s	RMS = 8.484% 
1%	RMS = 0.807%  9.80 s	RMS = 0.859%  2.17 s	RMS = 7.885% 

Table 8.5: Dependency of the resulting signal on the convergence threshold (10% and 1%) and the signal's initialization. The specified timings represent the total reconstruction run-time. $\lambda_2^{\mathrm{n}} = 5 \cdot 10^{-5}$ has been used; for the pyramid, it has been scaled by $f_\lambda = 10$ for each coarser scale (yielding $\lambda_2^{\mathrm{n}} = 5 \cdot 10^{-2}$ for level 4).

Engine: 419,431 points (5%) extracted from 256×256×128 grid



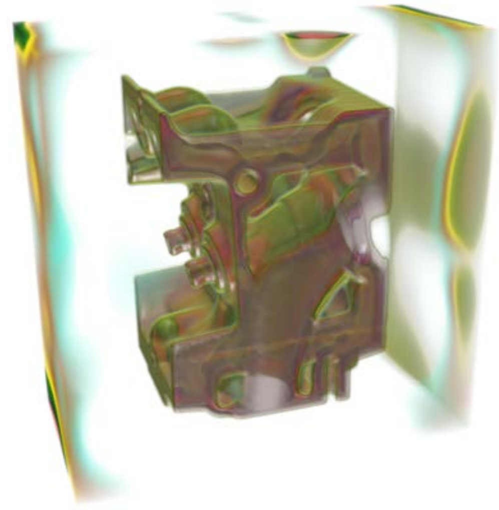
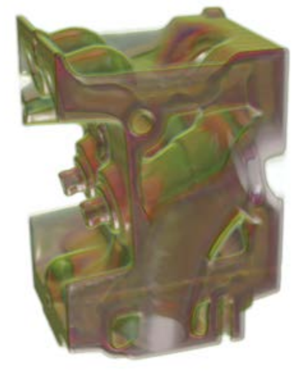
f_λ = 1: 10.69 s	f_λ = 10: 10.30 s
RMS = 5.415%	RMS = 6.255%
	
RMS$_g$ = 10.265% (194.595%)	RMS$_g$ = 5.750% (23.039%)

Table 8.6: Impact of scaling λ_p^{n} when transitioning to a coarser scale for multi-scale pyramids, illustrated by the sparse Engine (5%) data set at resolution level 7 (128×128×64). The results have been obtained by reconstructing a levels 5-7 pyramid with $\lambda_2^{\mathrm{n}} = 10^{-3}$. We see that artifacts in coarser scales (due to λ_2^{n} being too small for coarser resolutions, see Table 8.3 on page 63) are propagated to finer scales.

8.3 Accuracy of Derived Coarser Scales

Let us now assess the accuracy of coarser scales in multi-scale pyramids. These coarser scales differ from direct reconstructions at identical resolution in two aspects: firstly, they are initialized with the next coarser scale (if available) during solving, and secondly and more importantly, their linear systems (matrix $\mathbf{A}$ and right-hand-side vector $\mathbf{b}$) are not based directly on the given point samples, but derived from the finest scale's linear system by iterative coarsening. Let us focus on the latter by comparing the solution of a derived linear system to the solution of the dedicated linear system at identical resolution. By dedicated linear system, we mean the one obtained by constructing $\mathbf{A}$ and $\mathbf{b}$ directly from the point samples and using an identical λ_p^{n}. Corresponding 1D results are given in Figure 8.1, indicating an exact equivalence. Figure 8.2 illustrates the negligible imprecision introduced by padding the integral range of the squared semi-norm to the next appropriate knots, which is due to our implementation relying solely on the $\eta_i(q, \delta_k)$ (Eq. 3.26).

We conclude that constructing a dedicated linear system for each scale is not worth the effort, and that simply coarsening the matrices and right-hand-side vectors yields virtually equivalent linear systems for coarser scales. This is more efficient because the computation is independent from the samples set, easily parallelizable and governed by the total number of coefficients (which decreases considerably with each scale transition).

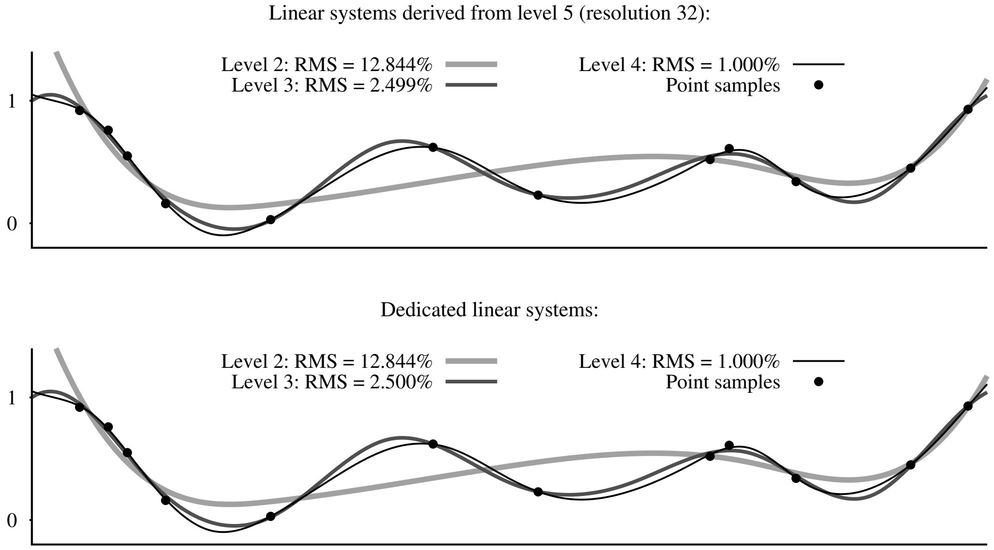
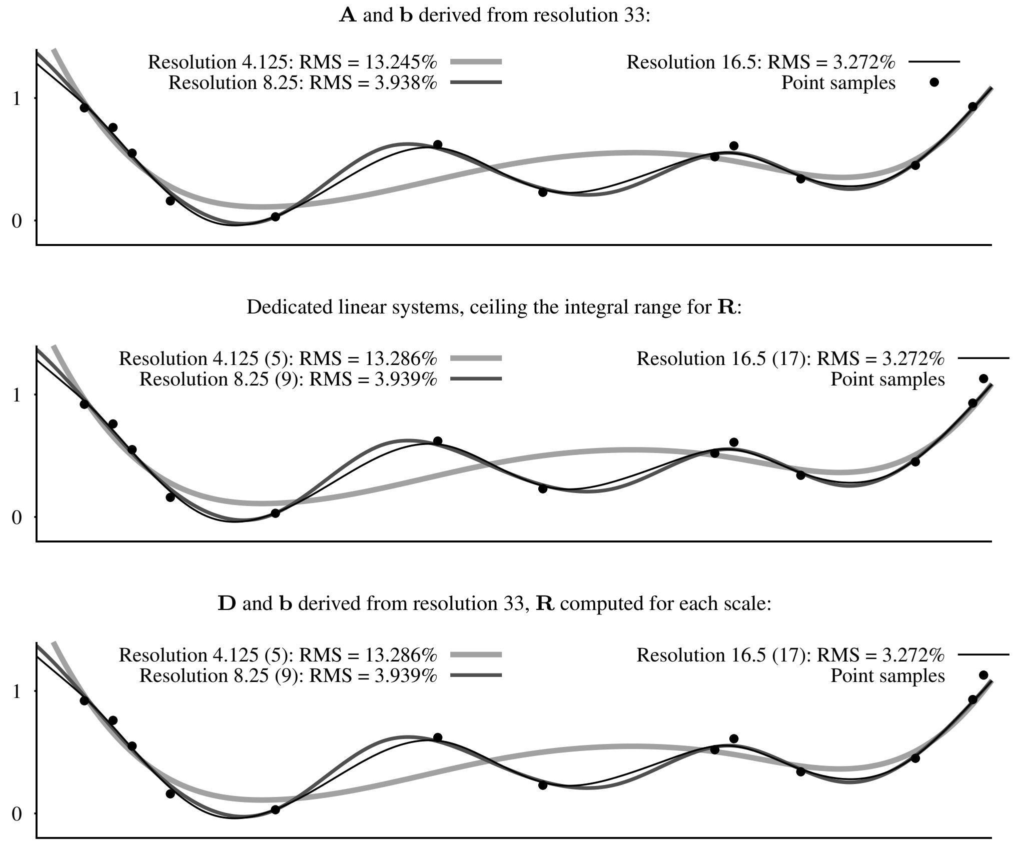


Figure 8.1: Equivalent results for derived and dedicated linear systems using $\lambda_2^{\mathrm{n}} = 10^{-6}$.



$\mathbf{A}$ and $\mathbf{b}$ derived from resolution 33:

Dedicated linear systems, ceiling the integral range for $\mathbf{R}$:

$\mathbf{D}$ and $\mathbf{b}$ derived from resolution 33, $\mathbf{R}$ computed for each scale:

Figure 8.2: Almost equivalent results for derived and dedicated linear systems using $\lambda_2^{\mathrm{n}} = 10^{-5}$ and an odd finest resolution of 33 inner B-spline intervals, so that the coarser scales do not end at a knot. When constructing dedicated linear systems, the signal is extended to the next knot for the computation of the integral of the squared semi-norm, while a strictly coarsened linear system's integral covers the actual signal only. For the bottom figure, we compute $\mathbf{R}$ directly for each derived coarser scale and add it to coarsened $\mathbf{D}$. The integral's range is thus extended too, and we get equivalent results compared to the dedicated linear systems.

Coarsened Linear System vs. Coarsened Signal

Let us now compare the solution of a derived coarser linear system to the iteratively coarsened finest solution. Table 8.7 shows that simply coarsening a fine signal results in a way smoother coarse signal compared to the solution of a derived linear system, especially for cubic B-splines. The oversmoothing results in a way higher RMS error and a high discrepancy with respect to the finest solution, so that coarser scales obtained by simply coarsening finer scales are generally not suited for accurate LoD schemes. We suggest to use higher f_λ values in case a higher smoothness for coarser scales is desired.

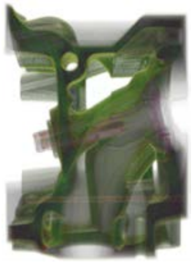
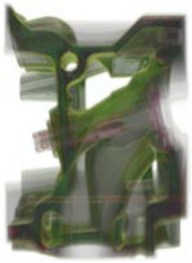
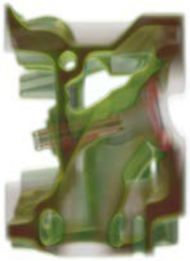
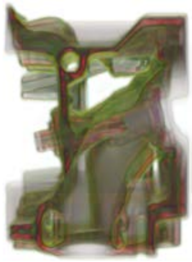
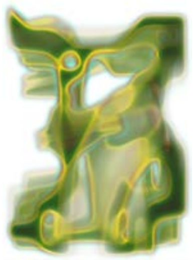
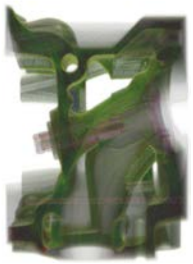
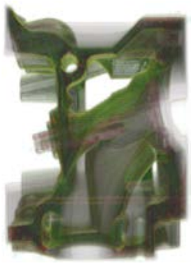
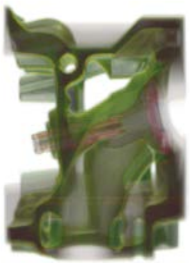
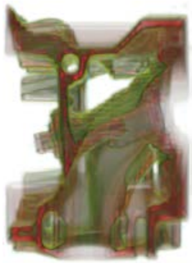
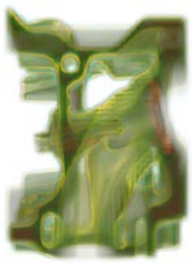
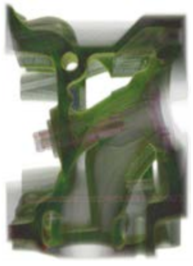
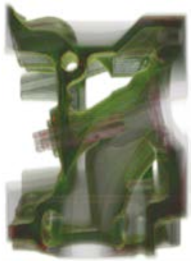
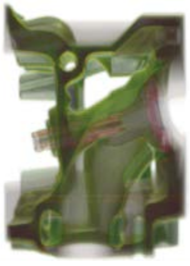
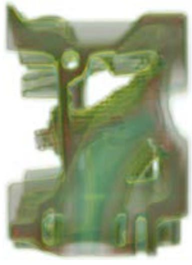
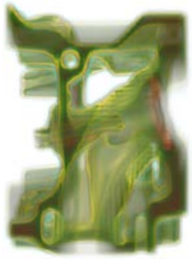
Level 8: 256×256×128	Level 7: 128×128×64		Level 6: 64×64×32	
	Coarsened linear system	Coarsened level 8 signal	Coarsened linear system	Coarsened level 8 signal
Cubic, $f_\lambda = 10$:				
RMS = 0.208%	RMS = 1.638%	RMS = 8.281%	RMS = 5.503%	RMS = 20.361%
				
Linear, $f_\lambda = 2$:				
RMS = 0.203%	RMS = 2.002%	RMS = 5.330%	RMS = 6.384%	RMS = 12.908%
				
Linear, $f_\lambda = 20$:				
RMS = 0.203%	RMS = 2.914%	RMS = 5.330%	RMS = 10.163%	RMS = 12.908%
				

Table 8.7: Comparison between the solution of a derived coarser linear system and the iteratively coarsened finest solution for cubic and linear multi-scale pyramids using the Engine (20%) data set.

8.4 Run-Time Analysis

Tables 8.8 to 8.14 present some performance numbers for different data sets and resolutions. We compare the reconstruction at different resolution levels to the reconstruction of multi-level pyramids and focus on the run-time performance of the critical parts:

1. Constructing $\mathbf{D}$ and $\mathbf{b}$ based on the point samples (only for the finest scale if reconstructing a pyramid). The run-time does not scale perfectly with the number of point samples – the performance also depends upon the resolution. This is due to the memory not being accessed in a cache-friendly way when accumulating the kernels of every sample. The accumulation also prevents a feasible parallelization, so that our implementation is strictly single-threaded.

2. Deriving $\mathbf{D}$ and $\mathbf{b}$ for coarser scales of multi-scale pyramids by iterative coarsening. The run-time scales as expected with the total number of coefficients, i.e., the run-time increases approximately by a factor of $2^d = 8$ for a finer scale. The run-time is rather small with respect to the total reconstruction time and way smaller than constructing a dedicated linear system for each coarser scale.

3. Adding $\lambda_p \mathbf{R}$ to each scale's $\mathbf{D}$. The run-time scales with the total number of coefficients and is also rather small.

4. Solving the linear system in each scale. The run-time of a SOR iteration and the computation of the residue norm scales with the total number of coefficients too. The run-time for solving the linear system usually does not scale by $2^d = 8$ for a finer scale though, which is caused by the adaptive termination due to the convergence threshold and a dependency upon λ_p^{n} – i.e., different scales usually require a different number of iterations. The convergence of multi-scale pyramids is improved by initializing finer scales with coarser results. The overhead of deriving and solving linear systems in coarser scales therefore generally pays off and leads to a smaller RMS error and/or a faster reconstruction of the whole pyramid compared to reconstructing the finest scale alone.

	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 2	0.8		7×7×7	0.01		0.00	0.01	**0.02**	4.29%
Level 3	0.4		11×11×11	0.01		0.00	0.01	**0.02**	0.98%
Level 4	0.2		18×19×18	0.01		0.00	0.04	**0.05**	0.56%
Level 5	0.1		33×35×33	0.01		0.01	0.07	**0.09**	0.26%
Levels 2-5	0.1	2		0.01	0.01	0.00	0.12	**0.14**	4.29% 0.98% 0.56% 0.10%
Cubic:									
Level 2	10^{-2}		7×7×7	0.09		0.00	0.01	**0.10**	3.05%
Level 3	10^{-3}		11×11×11	0.09		0.01	0.02	**0.12**	1.09%
Level 4	10^{-4}		18×19×18	0.11		0.01	0.05	**0.17**	0.54%
Level 5	10^{-5}		33×35×33	0.14		0.02	0.24	**0.40**	0.33%
Levels 2-5	10^{-5}	10		0.11	0.02	0.03	0.31	**0.47**	3.05% 1.08% 0.49% 0.11%

Table 8.8: **Oil**, 29,094 points: reconstruction timings, in seconds.

	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 2	8		5×5×5	0.02		0.00	0.01	**0.03**	5.96%
Level 3	4		9×9×9	0.02		0.00	0.02	**0.04**	3.04%
Level 4	2		17×17×17	0.02		0.00	0.04	**0.06**	0.89%
Level 5	1		33×33×33	0.02		0.01	0.13	**0.16**	0.23%
Level 6	0.5		65×65×65	0.03		0.01	0.54	**0.58**	0.06%
Levels 2-6	0.5	2		0.03	0.03	0.01	0.63	**0.70**	5.96% 3.04% 0.89% 0.23% 0.05%
Cubic:									
Level 2	10^{-1}		7×7×7	0.21		0.00	0.01	**0.22**	3.31%
Level 3	10^{-2}		11×11×11	0.20		0.01	0.02	**0.23**	0.96%
Level 4	10^{-3}		19×19×19	0.22		0.01	0.06	**0.29**	0.31%
Level 5	10^{-4}		35×35×35	0.23		0.02	0.35	**0.60**	0.07%
Level 6	10^{-5}		67×67×67	0.30		0.09	0.95	**1.34**	0.35%
Levels 2-6	10^{-5}	10		0.30	0.16	0.12	1.20	**1.78**	3.31% 0.89% 0.25% 0.03% 0.00%

Table 8.9: **Natural Convection**, 68,921 points: reconstruction timings, in seconds.

	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 4	$4 \cdot 10^{-3}$		17×17×17	0.02		0.00	0.05	**0.07**	6.46%
Level 5	$2 \cdot 10^{-3}$		33×33×33	0.03		0.00	0.09	**0.12**	0.97%
Level 6	10^{-3}		65×65×65	0.08		0.01	0.29	**0.38**	0.26%
Levels 4-6	10^{-3}	2		0.08	0.03	0.01	0.41	**0.53**	6.46% 0.95% 0.01%
Cubic:									
Level 4	10^{-5}		19×19×19	0.36		0.01	0.06	**0.43**	5.05%
Level 5	10^{-6}		35×35×35	0.96		0.02	0.29	**1.27**	0.37%
Level 6	10^{-7}		67×67×67	1.33		0.09	0.86	**2.28**	0.31%
Levels 4-6	10^{-7}	10		1.32	0.15	0.12	1.12	**2.71**	5.05% 0.14% 0.00%

Table 8.10: **Chirp**, 75,000 points: reconstruction timings, in seconds.

	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 6	20		61×65×17	0.34		0.00	0.14	**0.48**	5.31%
Level 7	10		120×129×32	0.64		0.01	0.61	**1.26**	2.67%
Level 8	5		239×257×63	1.08		0.07	5.46	**6.61**	0.56%
Levels 6-8	5	2		1.07	0.30	0.08	4.83	**6.28**	5.31% 2.67% 0.56%
Cubic:									
Level 6	$5 \cdot 10^{-4}$		63×67×19	9.44		0.03	0.26	**9.73**	5.12%
Level 7	$5 \cdot 10^{-5}$		122×131×34	14.39		0.15	1.86	**16.40**	2.27%
Level 8	$5 \cdot 10^{-6}$		241×259×65	19.57		1.07	14.85	**35.49**	0.31%
Levels 6-7	$5 \cdot 10^{-5}$	10		14.39	0.23	0.20	1.77	**16.59**	5.12% 2.28%
Levels 6-8	$5 \cdot 10^{-6}$	10		19.64	1.83	1.31	13.46	**36.24**	5.12% 2.28% 0.31%

Table 8.11: **Lobster**, 1,092,269 points (20%) extracted from anisotropic 301×324×56 grid: reconstruction timings, in seconds.

	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 6	160		65×9×17	2.01		0.00	0.03	**2.04**	3.92%
Level 7	80		129×17×33	2.00		0.00	0.13	**2.13**	3.26%
Level 8	40		257×32×65	2.04		0.01	0.77	**2.82**	2.14%
Level 9	20		513×62×129	2.17		0.07	6.36	**8.60**	0.80%
Levels 6-9	20	2		2.15	0.31	0.08	5.15	**7.69**	3.92% 3.26% 2.14% 0.80%
Cubic:									
Level 6	$5 \cdot 10^{-2}$		67×11×19	24.56		0.01	0.08	**24.65**	3.88%
Level 7	$5 \cdot 10^{-3}$		131×19×35	24.52		0.03	0.45	**25.00**	3.16%
Level 8	$5 \cdot 10^{-4}$		259×34×67	27.72		0.16	2.61	**30.49**	1.98%
Level 9	$5 \cdot 10^{-5}$		515×64×131	31.46		1.16	16.90	**49.52**	0.70%
Levels 6-8	$5 \cdot 10^{-4}$	10		27.79	0.28	0.19	1.86	**30.12**	3.88% 3.17% 1.99%
Levels 6-9	$5 \cdot 10^{-5}$	10		31.29	2.17	1.31	12.64	**47.41**	3.88% 3.17% 1.99% 0.70%

Table 8.12: **Bypass**, 7,929,856 points: reconstruction timings, in seconds.

	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 6	4		65×65×33	0.19		0.00	0.19	**0.38**	8.97%
Level 7	2		129×129×65	0.38		0.02	1.20	**1.60**	2.52%
Level 8	1		257×257×129	0.66		0.15	14.74	**15.55**	0.39%
Levels 6-7	2	2		0.38	0.07	0.02	1.11	**1.58**	8.97% 2.44%
Levels 6-8	1	2		0.66	0.63	0.17	8.48	**9.94**	8.97% 2.44% 0.36%
Cubic:									
Level 6	$2 \cdot 10^{-3}$		67×67×35	4.95		0.06	0.55	**5.56**	10.33%
Level 7	$1 \cdot 10^{-4}$		131×131×67	7.13		0.33	3.97	**11.43**	2.51%
Level 8	$5 \cdot 10^{-6}$		259×259×131	10.14		2.36	24.22	**36.72**	0.29%
Levels 6-7	$1 \cdot 10^{-4}$	20		7.14	0.44	0.43	4.45	**12.46**	10.33% 2.48%
Levels 6-8	$5 \cdot 10^{-6}$	20		9.96	10.17	5.54	29.16	**54.83**	10.33% 2.48% 0.25%

Table 8.13: **Engine**, 419,431 points (5%) extracted from 256×256×128 grid: reconstruction timings, in seconds. For the cubic levels 6-8 pyramid, we run into memory paging issues due to insufficient RAM when coarsening the finest matrix. We would expect a coarsening run-time of about $(1 + 8) \cdot 0.45 \approx 4$ seconds and about $0.06 + 0.33 + 2.36 = 2.75$ seconds for adding the $\mathbf{R}$s if we had more memory, decreasing the total run-time probably by at least 8 seconds.

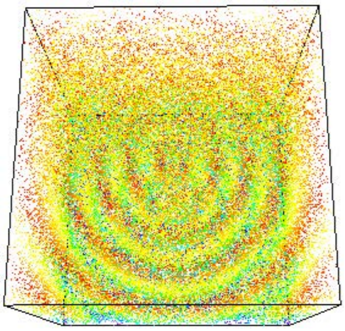
	λ_p^{n}	f_λ	Grid	$\mathbf{D}$ and $\mathbf{b}$	Coarsening	$\mathbf{R}$	Solving	**Total**	RMS error
Linear:									
Level 6	2.0		65×65×33	0.72		0.01	0.56	**1.29**	6.38%
Level 7	1.0		129×129×65	1.35		0.02	1.50	**2.87**	2.01%
Level 8	0.5		257×257×129	2.13		0.15	9.71	**11.99**	0.21%
Levels 6-7	1.0	2		1.33	0.08	0.02	1.53	**2.96**	6.38% 2.00%
Levels 6-8	0.5	2		2.14	0.70	0.17	9.57	**12.58**	6.38% 2.00% 0.20%
Cubic:									
Level 6	$5 \cdot 10^{-4}$		67×67×35	18.04		0.05	0.85	**18.94**	5.42%
Level 7	$5 \cdot 10^{-5}$		131×131×67	25.96		0.32	6.13	**32.41**	1.64%
Level 8	$5 \cdot 10^{-6}$		259×259×131	35.15		2.36	31.56	**69.07**	0.22%
Levels 6-7	$5 \cdot 10^{-5}$	10		25.83	0.45	0.40	5.48	**32.16**	5.42% 1.63%
Levels 6-8	$5 \cdot 10^{-6}$	10		35.06	14.16	8.13	32.89	**90.24**	5.42% 1.63% 0.20%

Table 8.14: **Engine**, 1,677,722 points (20%) extracted from 256×256×128 grid: reconstruction timings, in seconds. We run into the same memory paging issues for the cubic levels 6-8 pyramid.

8.5 Linear vs. Cubic Reconstruction

The previous results indicate that reconstructing cubic B-splines obviously requires a significantly higher run-time with respect to linear B-splines, but usually achieves lower RMS errors for coarser resolutions. When rendering the signals, the C^2 continuity of cubic B-splines leads to visually more pleasing results than the C^0 continuity of linear B-splines. The advantage (smooth signal vs. continuous signal, smooth analytical gradient vs. continuous gradient approximation) is more notable for lower resolutions and zoomed renderings. Tables 8.15 to 8.18 allow a direct comparison of corresponding linear and cubic pyramids. The specified scale sizes in Kilobytes represent the memory foot-print of the coefficient grids (including outer coefficients) assuming a single-precision format (4 bytes per coefficient). The timings denote the total run-time for pyramid reconstruction.

Synthetic Chirp: 75,000 points



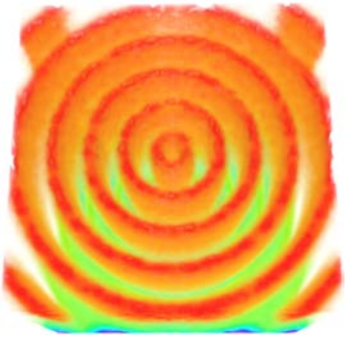
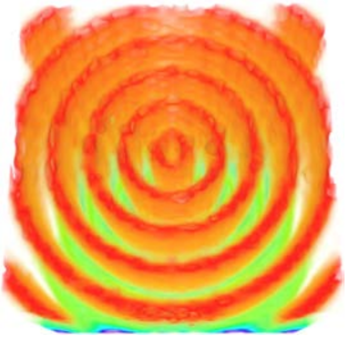
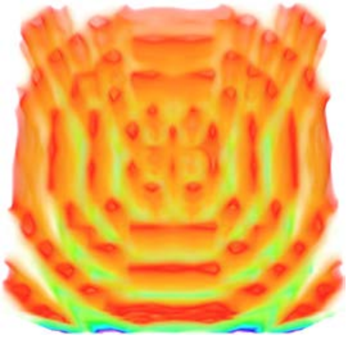
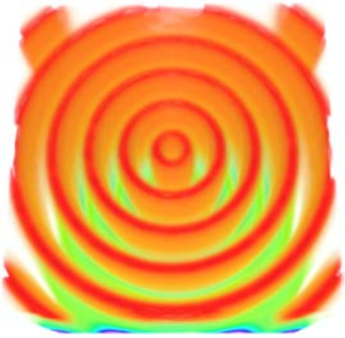
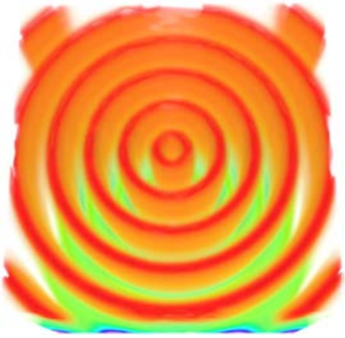
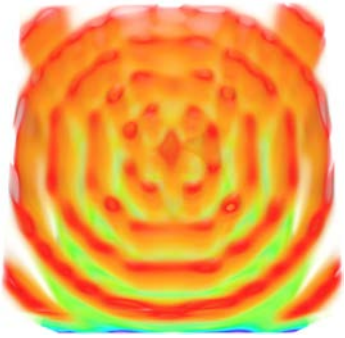
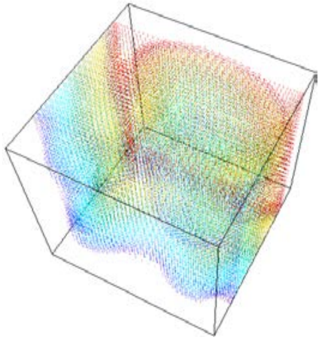
64×64×64: 1,175 KB	32×32×32: 167 KB	16×16×16: 27 KB
	Linear ($\lambda_1^{\mathrm{n}} = 10^{-2}, f_\lambda = 2$): 0.48 s	
RMS = 0.017%	RMS = 0.951%	RMS = 6.460%
		
	Cubic ($\lambda_2^{\mathrm{n}} = 10^{-6}, f_\lambda = 10$): 2.47 s	
RMS = 0.005%	RMS = 0.140%	RMS = 5.051%
		

Table 8.15: Comparison between linear and cubic levels 4-6 pyramids for the Synthetic Chirp data set.

Natural Convection: 68,921 points



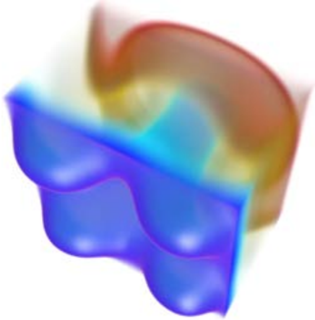
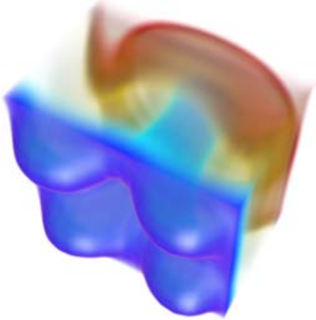
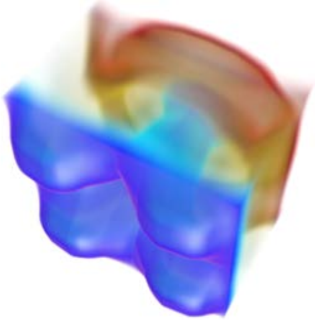
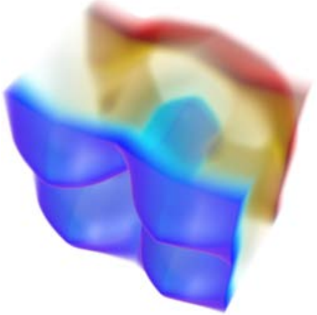
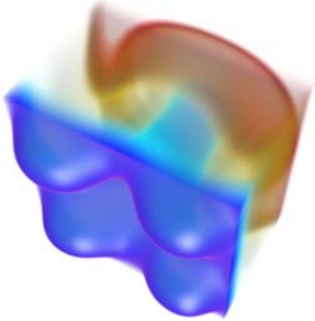
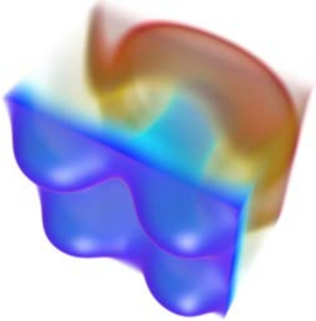
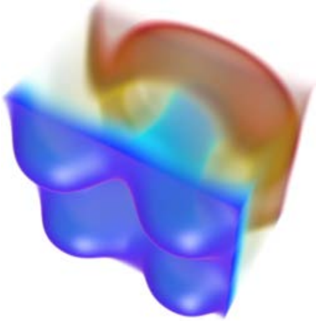
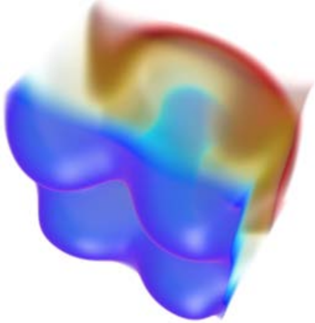
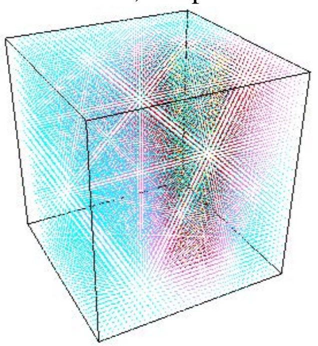
32×32×32: 167 KB	16×16×16: 27 KB	8×8×8: 5 KB	4×4×4: 1 KB
Linear ($\lambda_1^n = 1, f_\lambda = 2$): 0.21 s			
RMS = 0.229%	RMS = 0.887%	RMS = 3.040%	RMS = 5.961%
			
Cubic ($\lambda_2^n = 10^{-4}, f_\lambda = 10$): 0.67 s			
RMS = 0.026%	RMS = 0.249%	RMS = 0.892%	RMS = 3.315%
			

Table 8.16: Comparison between linear and cubic levels 2-5 pyramids for the Natural Convection data set. The rendering of the linear signal in resolution level 2 clearly shows that the signal is a trilinear interpolation of 5×5×5 coefficients.

Oil: 29,094 points



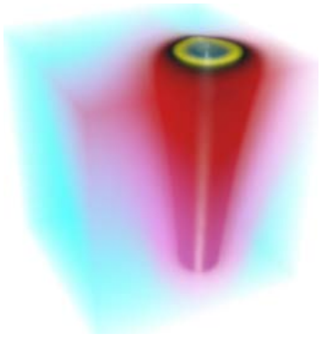
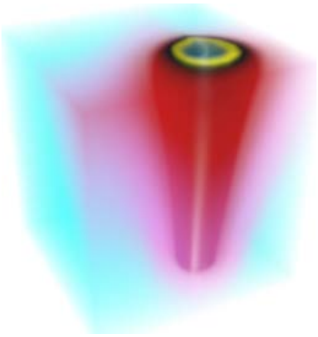
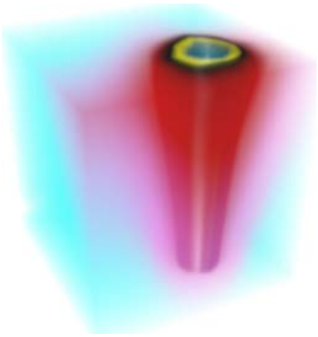
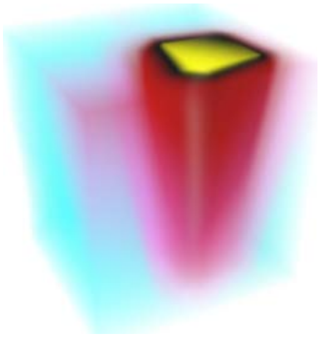
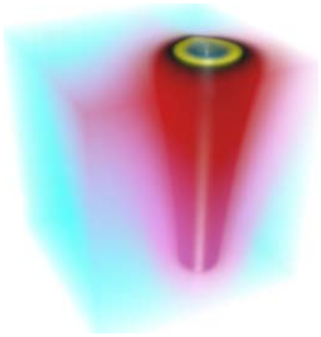
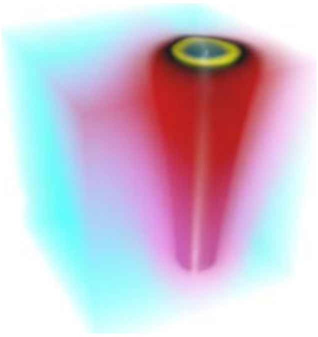
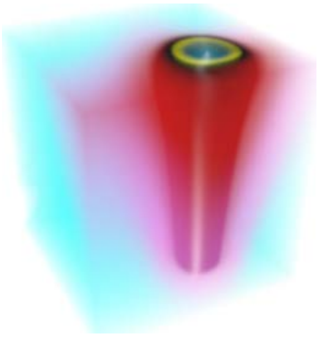
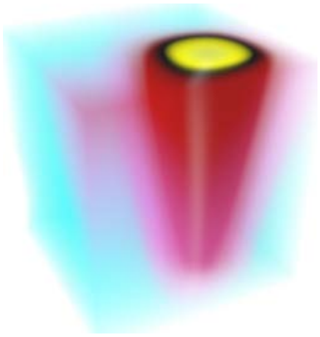
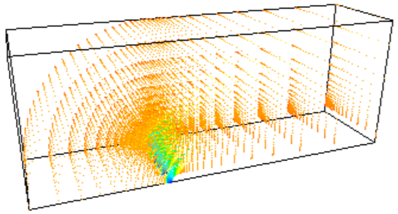
30×32×30: 149 KB	15×16×15: 24 KB	8×8×8: 5 KB	4×4×4: 1 KB
Linear ($\lambda_1^n = 10^{-1}, f_\lambda = 2$): 0.12 s			
RMS = 0.104%	RMS = 0.565%	RMS = 0.984%	RMS = 4.294%
			
Cubic ($\lambda_2^n = 10^{-4}, f_\lambda = 10$): 0.47 s			
RMS = 0.119%	RMS = 0.506%	RMS = 1.112%	RMS = 3.251%
			

Table 8.17: Comparison between linear and cubic levels 2-5 pyramids for the Oil data set.

Blunt-Fin: 40,960 points



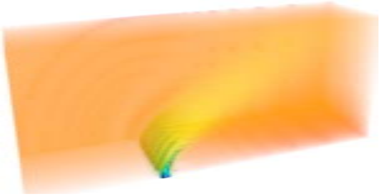
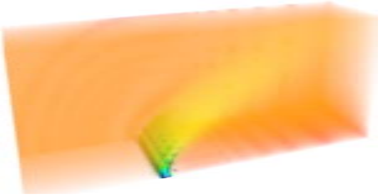
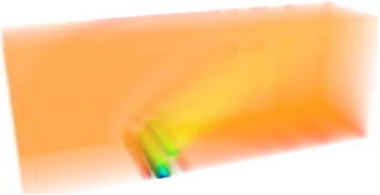
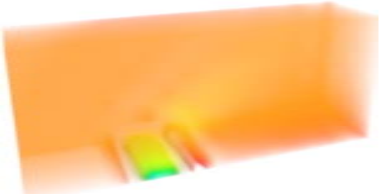
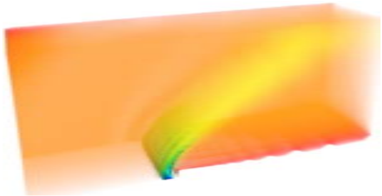
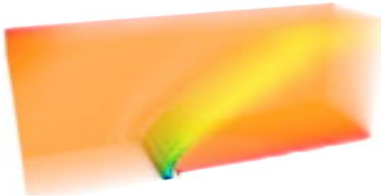
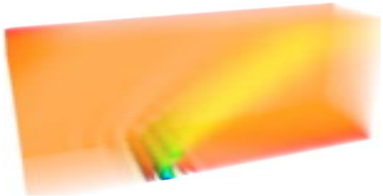
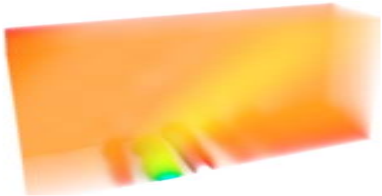
128×49×34: 985 KB	64×25×17: 147 KB	32×13×9: 26 KB	16×7×5: 6 KB
Linear ($\lambda_1^n = 10, f_\lambda = 2$): 0.39 s			
RMS = 1.260%	RMS = 2.226%	RMS = 6.197%	RMS = 10.643%
			
Cubic ($\lambda_2^n = 5 \cdot 10^{-5}, f_\lambda = 10$): 0.83 s			
RMS = 0.858%	RMS = 1.719%	RMS = 4.311%	RMS = 8.484%
			

Table 8.18: Comparison between linear and cubic levels 4-7 pyramids for the Blunt-Fin data set.

CHAPTER 9

Block-Based Reconstruction

9.1 Visual Block Discontinuities

Let us now compare block-based reconstruction to global reconstruction. To do so, we use powers of 2 for s, the number of cubes along the volume's longest axis. This allows to represent a volume of 256^3 inner B-spline intervals (single cube at resolution level 8) by 2^3 cubes (at level 7 each), 4^3 cubes (at level 6 each) etc. For the following comparisons, we reconstruct an independent multi-scale pyramid for each cube, i.e., we do not merge cubes to larger blocks yet, and then fuse the pyramids to enforce C^{n-1} continuity. Results in terms of RMS errors, run-time and renderings are given in Tables 9.1 to 9.4. The images representing the difference between a rendering and the reference rendering are generated by computing the absolute sRGB difference, which is then desaturated, scaled by 5, inverted, scaled by 0.8 and clamped into the normalized $[0, 0.8]$ range (background light-grey: no sRGB difference, black: desaturated absolute sRGB difference ≥ 0.2).

We see that using a padding at inner block sides is crucial to obtain virtually equivalent results to global reconstruction. The minimum useful padding amounts to $\frac{n+1}{2}$ B-spline intervals in order to include all point samples which directly influence non-padded inner coefficients; non-padded outer coefficients will be replaced by the fusing scheme. If the point samples are rather sparse, the impact of the regularization constraint and hence the indirect influence of excluded point samples increases. In this case, the block extremities to be fused may diverge strongly, leading to notable artifacts (see Figure 9.1). We need wider paddings for these cases, including more distant point samples (context) to increase the accuracy of the block extremities. The dependency upon the point samples density is clearly illustrated by the different results for both Engine data sets (5% of the grid samples in Table 9.3, 20% in Table 9.4). Higher resolutions obviously need wider paddings in order to include the same outer point samples because the B-spline interval sizes decrease. The results clearly show that the impact of the padding increases with finer subdivisions. Merging cubes to larger blocks (whose reconstruction memory footprint ideally fits into $\frac{\text{available RAM}}{\text{number of CPU cores}}$) thus increases accuracy and at the same time reduces the computational overhead – the padding is applied to the block borders only, not to every cube.

A subdivision leads to a general computation overhead, even if using no padding at all – for 256^3 inner B-spline intervals, a single cube needs $(256+n)^3$ coefficients whereas 4^3 cubes with $(64+n)^3$ coefficients each amount to a total of $(256+4n)^3$ coefficients. The overhead obviously increases with wider paddings (more total coefficients, and point samples may contribute to multiple blocks when constructing the linear systems). Our results confirm this. But we also see that the run-time can be significantly reduced by reconstructing blocks independently in a parallel fashion, thereby better exploiting multiple CPU cores. Of key importance here is that the single-threaded construction of the global linear system (matrix $\mathbf{D}$ and right-hand-side vector $\mathbf{b}$) is replaced by a simultaneous construction of multiple smaller linear systems.

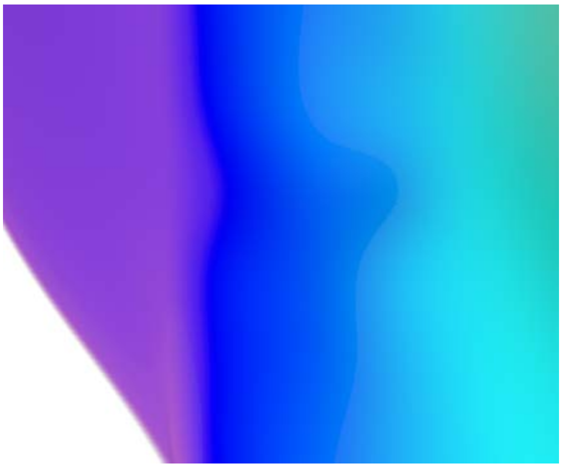


Figure 9.1: Zoomed rendering of a cubes boundary (from Table 9.1, using 4^3 cubes and no inner padding at all). Our fusing scheme results in a C^2 continuous boundary, but there is a visual discontinuity (bump) because the cube extremities diverge strongly.

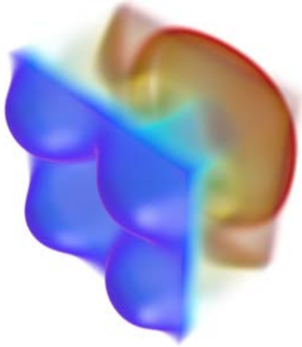
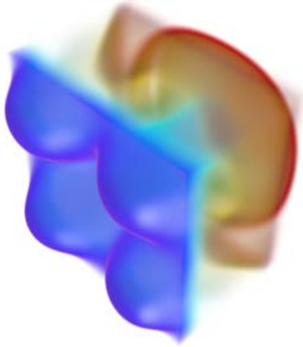
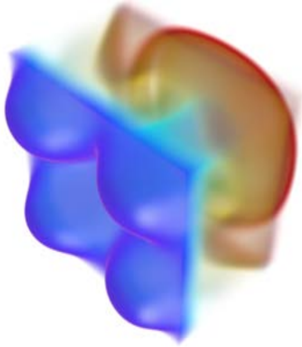
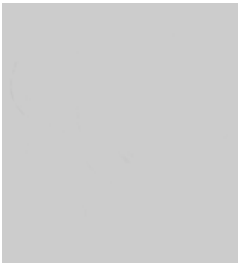
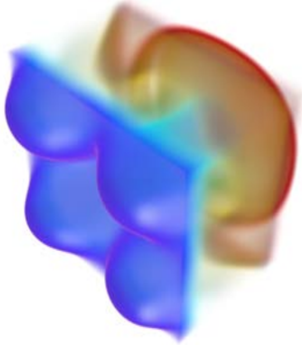
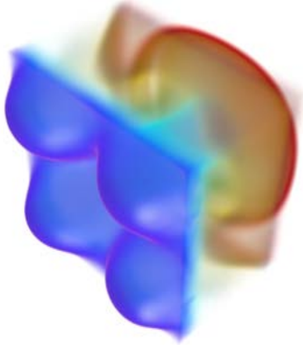
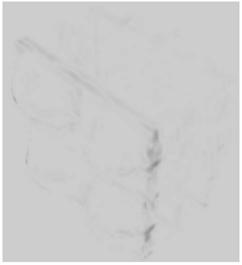
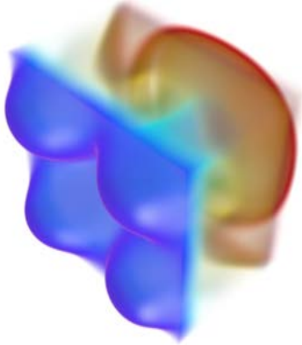
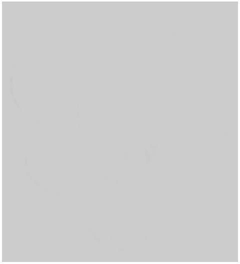
Cubes	Reference	No inner padding		2 intervals	
2×2×2	RMS = 0.026%, 0.61 s 	RMS = 0.071%, 0.37 s 	Difference 	RMS = 0.029%, 0.49 s 	Difference 
4×4×4	RMS = 0.026%, 0.61 s 	RMS = 0.097%, 0.60 s 	Difference 	RMS = 0.030%, 1.18 s 	Difference 

Table 9.1: Comparison of corresponding cubic multi-scale pyramids for the Natural Convection data set, two subdivisions and different block paddings (as number of additional B-spline intervals at each side, for the finest scale). The reference is a single cube reconstructed via a levels 4-5 pyramid ($\lambda_2^{\mathrm{n}} = 10^{-4}$, $f_\lambda = 10$) and rendered at the finest scale (32^3 inner B-spline intervals).

Reference: RMS = 0.701%, 50.08 s

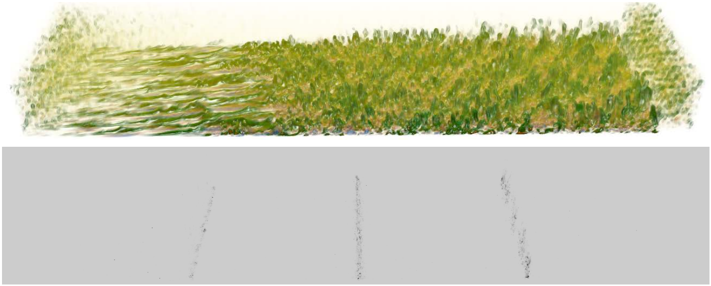
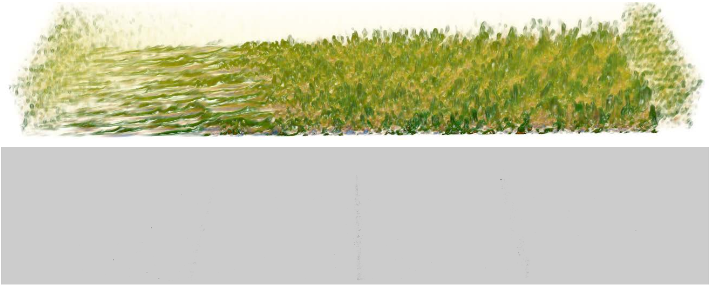
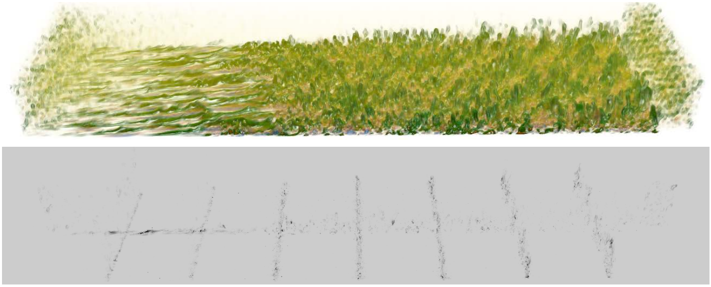
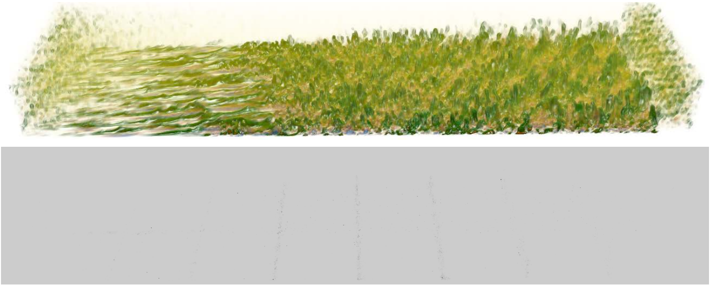
Cubes	No inner padding	2 intervals
4×1×1	RMS = 0.704%, 24.62 s 	RMS = 0.701%, 25.49 s 
8×1×2	RMS = 0.719%, 28.35 s 	RMS = 0.702%, 29.85 s 

Table 9.2: Comparison of corresponding cubic multi-scale pyramids for the Bypass data set, two subdivisions and different block paddings. The reference is a single cube reconstructed via a levels 6-9 pyramid ($\lambda_2^{\mathrm{n}} = 5 \cdot 10^{-5}$, $f_\lambda = 10$) and rendered at the finest scale (512×61×128 inner B-spline intervals).

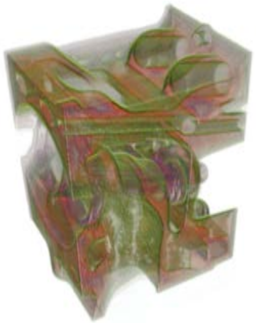
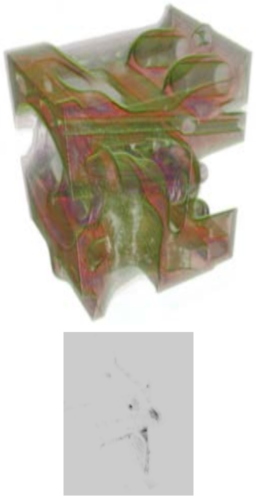
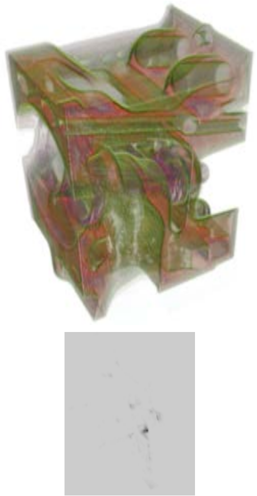
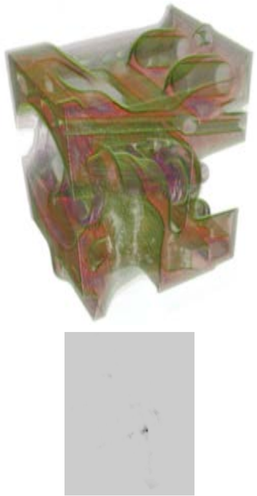
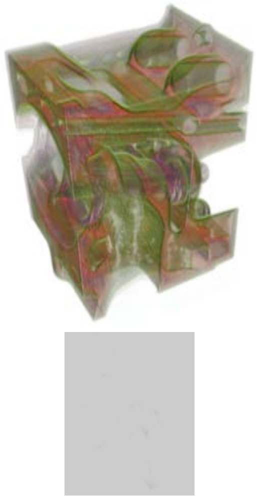
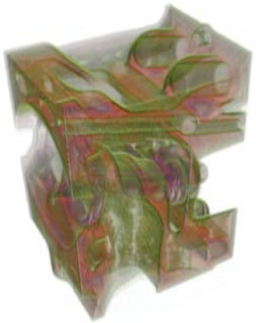
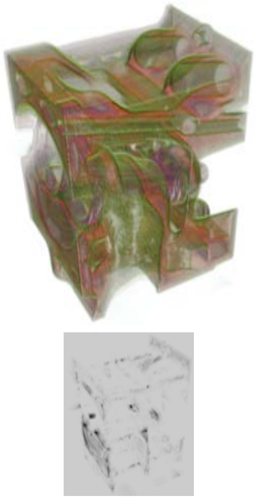
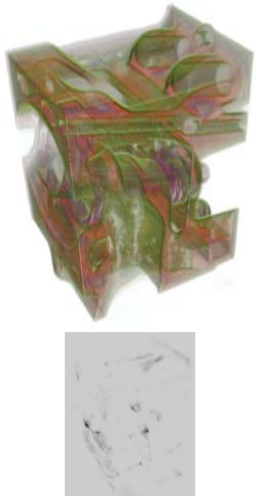
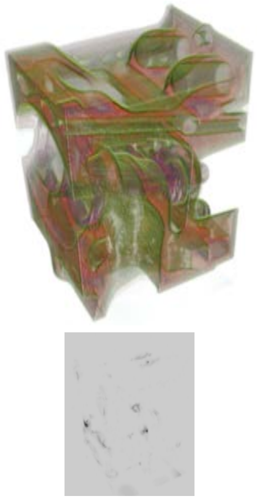
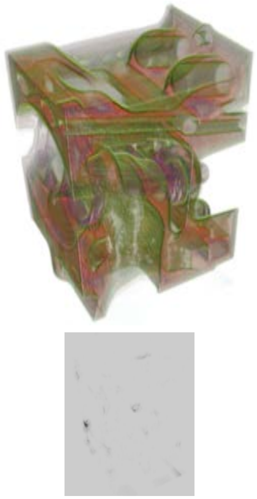
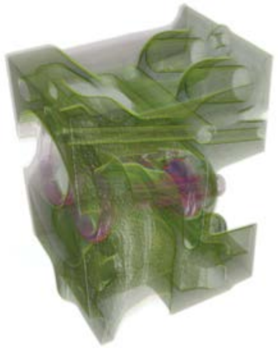
Cubes	Reference	No inner padding	2 intervals	4 intervals	6 intervals
2×2×1	RMS = 0.253%, 63.03 s 	RMS = 0.435%, 48.35 s 	RMS = 0.264%, 58.95 s 	0.255%, 68.08 s 	RMS = 0.253%, 75.64 s 
4×4×2	RMS = 0.253%, 63.03 s 	RMS = 1.214%, 42.49 s 	RMS = 0.295%, 47.95 s 	0.258%, 53.75 s 	RMS = 0.254%, 59.22 s 

Table 9.3: Comparison of corresponding cubic multi-scale pyramids for the Engine data set (5%), two subdivisions and different block paddings. The reference is a single cube reconstructed via a levels 6-8 pyramid ($\lambda_2^{\mathrm{n}} = 5 \cdot 10^{-6}$, $f_\lambda = 20$) and rendered at the finest scale (256×256×128 inner B-spline intervals). The run-time for the finer subdivision is significantly smaller than for the coarser one – this is due to memory paging issues when reconstructing multiple large cubes simultaneously.

Reference: RMS = 0.204%, 89.55 s



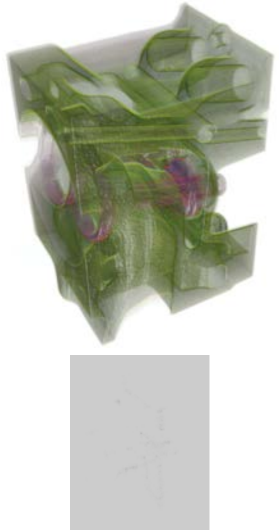
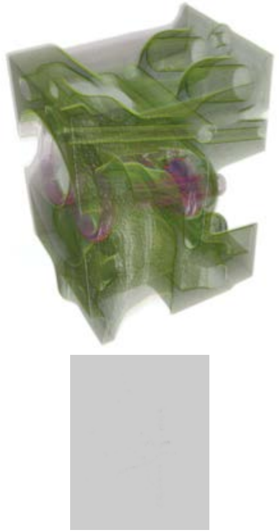
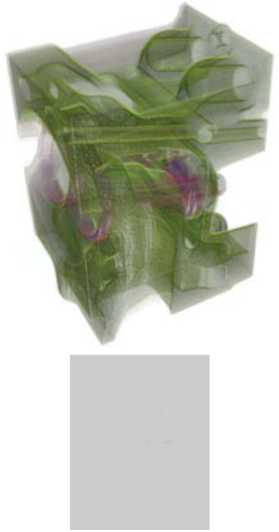
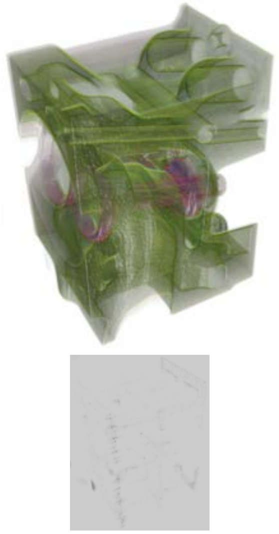
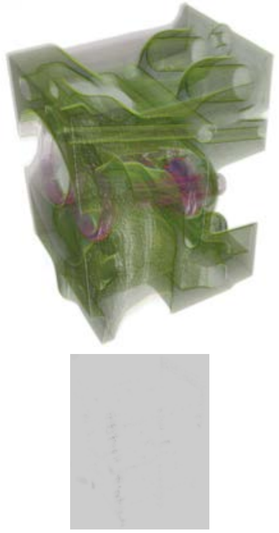
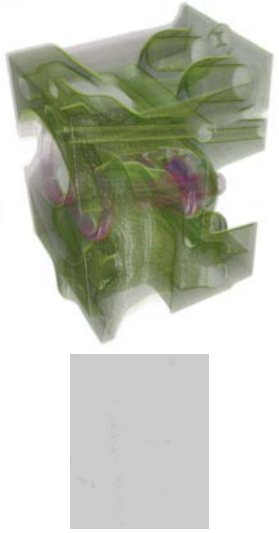
Cubes	No inner padding	2 intervals	4 intervals
2×2×1	RMS = 0.228%, 55.71 s 	RMS = 0.211%, 62.28 s 	RMS = 0.207%, 67.04 s 
4×4×2	RMS = 0.306%, 49.99 s 	RMS = 0.222%, 58.13 s 	RMS = 0.209%, 65.14 s 

Table 9.4: Comparison of corresponding cubic multi-scale pyramids for the Engine data set (20%), two subdivisions and different block paddings. The reference is a single cube reconstructed via a levels 6-8 pyramid ($\lambda_2^{\mathrm{n}} = 5 \cdot 10^{-6}$, $f_\lambda = 10$) and rendered at the finest scale (256×256×128 inner B-spline intervals). Memory paging issues again lead to higher run-times for the coarser subdivision.

9.2 Larger Data Sets

Besides allowing for efficient parallelization, the probably more important advantage of block-based reconstruction is the ability to reduce memory requirements, enabling to fit the reconstruction of large volumes into a certain memory budget.

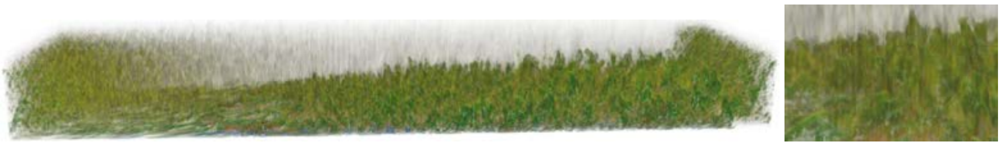
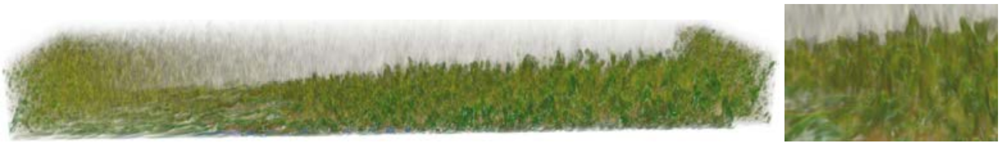
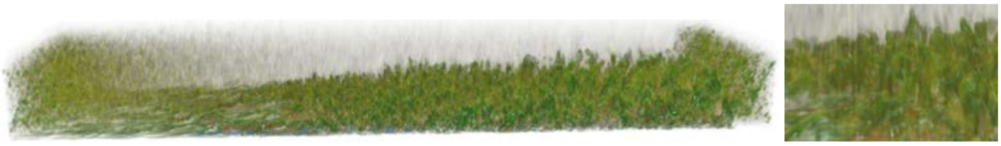
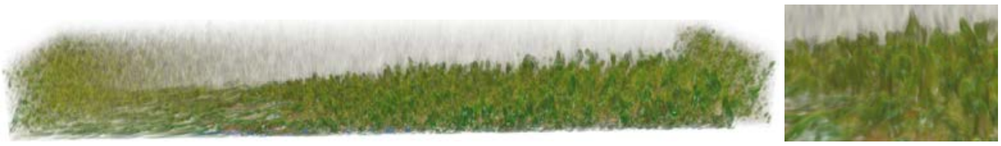
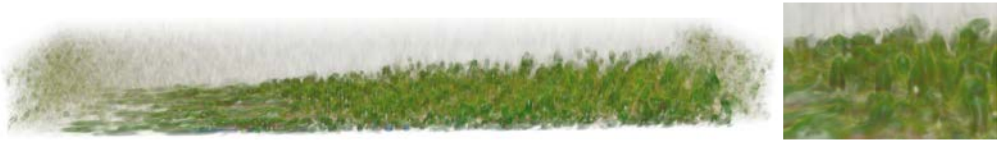
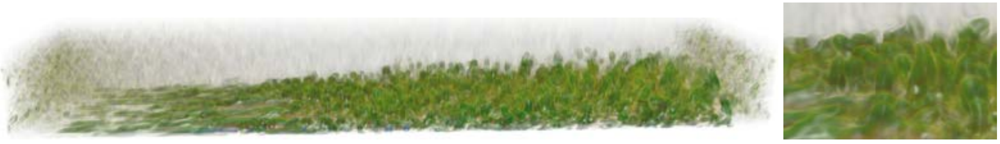
Most of the memory footprint during reconstruction is due to the linear system's matrix $\mathbf{A}$. The matrix for a cubic 3D B-spline with a total of $V = N_x N_y N_z$ coefficients requires $(4N_x - 6)(4N_y - 6)(4N_z - 6) \approx 64V$ elements (linear 3D B-spline: $(2N_x - 1)(2N_y - 1)(2N_z - 1) \approx 8V$ elements). When decrementing the resolution level, V is reduced to about $2^{-d} = \frac{1}{8}$th, and so are vectors and matrices, so that a multi-scale pyramid is not very costly in terms of additional memory: the total size of all scales approximately converges to $\frac{2^d}{2^d - 1} = \frac{8}{7}$ of the finest scale's size. What is costly though are temporary buffers when coarsening a matrix – our implementation exploits linear separability and so coarsens the matrix horizontally first (buffer size for the result: about $\frac{1}{2}$ of the fine matrix), then that buffer vertically (second buffer size: about $\frac{1}{4}$ of the fine matrix), and finally the second buffer along z to obtain the coarsened matrix. These two buffers for the intermediate results increase the memory footprint temporarily by $\frac{3}{4}$ of the fine matrix' one. We are thus able to derive a rough rule of thumb for the memory footprint of 3D B-spline multi-scale pyramids ($\mathbf{A}$, $\mathbf{x}$ and $\mathbf{b}$ for each scale), V being the total number of B-spline coefficients for the pyramid's finest scale:

- tricubic B-spline: roughly $[(64 + 2) \cdot \frac{8}{7}] \cdot V \approx 75V$ elements (peak: roughly $(75 + 64 \cdot \frac{3}{4}) \cdot V \approx 123V$ elements when coarsening the finest matrix)
- trilinear B-spline: roughly $[(8 + 2) \cdot \frac{8}{7}] \cdot V \approx 11.5V$ elements (peak: roughly $(11.5 + 8 \cdot \frac{3}{4}) \cdot V \approx 17.5V$ elements)

The peak footprint of a level 7 cube (128^3 inner B-spline intervals for the finest scale) therefore amounts to roughly 1 GB for tricubic B-splines if using the single-precision format (4 bytes per element), and about 0.14 GB for linear B-splines. For our test machine with its 4 GB RAM and 4 CPU cores, we want to retain a RAM reserve for the OS and other applications and then divide the remaining memory budget by 4 threads. It therefore makes sense to restrict the cubes' highest resolution level to level 6 for cubic reconstructions and to level 7 for linear ones, yielding a peak foot-print of roughly 140 MB for each cube pyramid. To exploit the memory budget more closely, cubes are merged to larger blocks – e.g., a cubic pyramid for a block of four cubes at level 6 and a linear pyramid for a block of 4 cubes at level 7 will both have a peak foot-print of approximately 0.55 GB.

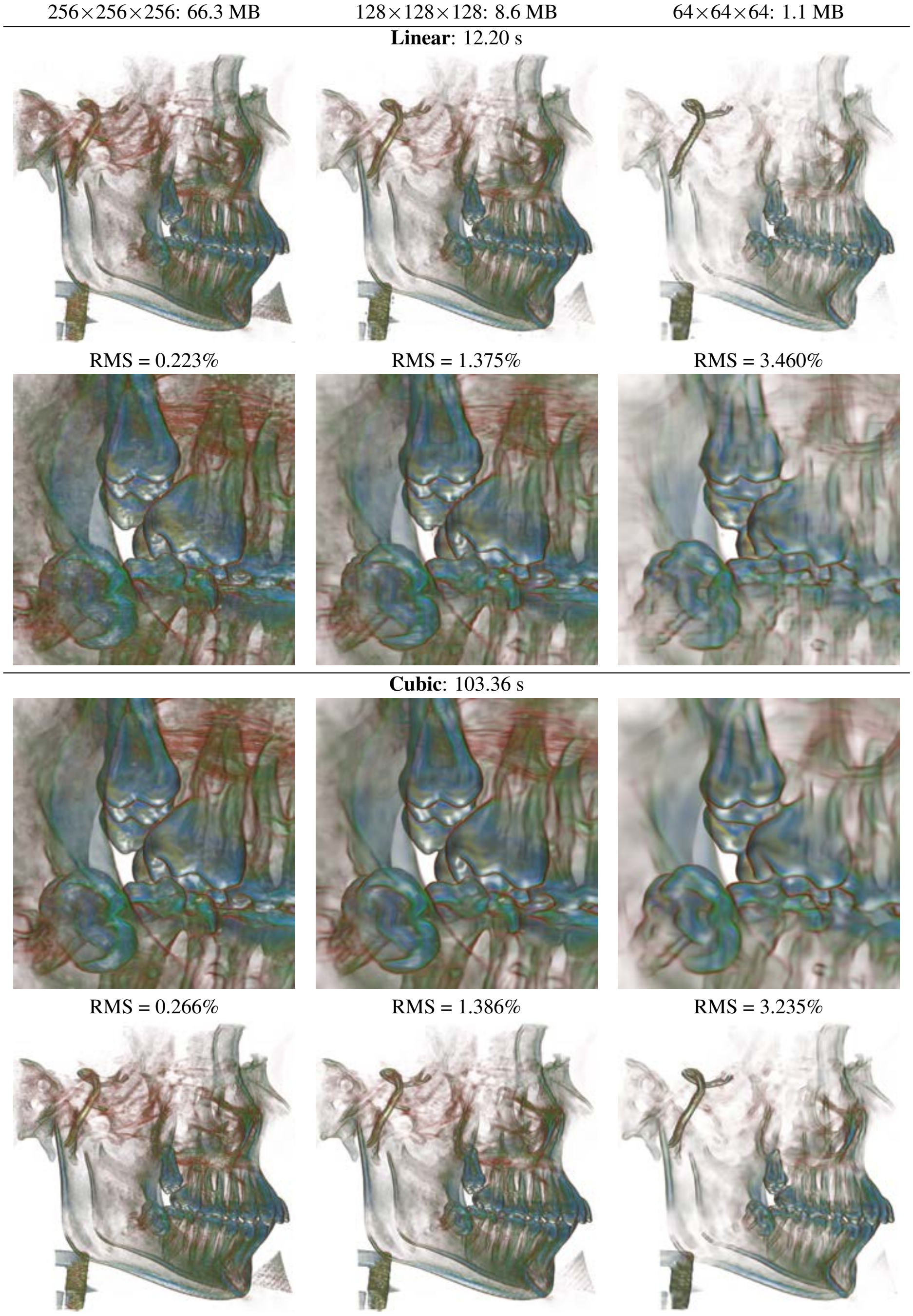
So by using appropriate subdivisions, we are able to reconstruct data sets at arbitrary resolutions (note that s, the number of cubes along the volume's longest axis, need not be an integer). This allows an evaluation of our approach for larger data sets, using higher resolutions which could not be reconstructed globally due to insufficient RAM. For all results given in Tables 9.5 to 9.9, we use an inner block padding of 4 B-spline intervals. The peak gradient magnitudes decrease for coarser scales due to lower resolutions prohibiting rapid changes. The gradient magnitude may not only affect shading, but also a ray sample's opacity (if we want to emphasize regions with stronger gradients). This is the reason for the higher translucency of coarser scales in Table 9.6.

While the linear reconstruction at the highest resolution often yields comparable results to cubic reconstruction at a fraction of the required run-time (as long as not zooming in a lot), we see that cubic B-splines are much better suited to represent the signal at coarser resolutions, where the C^2 continuity pays off.

Resolution	
768×91×191	**Linear**, RMS = 0.397%: 
	Cubic, RMS = 0.215%: 
384×46×96	**Linear**, RMS = 1.303%: 
	Cubic, RMS = 0.973%: 
192×23×48	**Linear**, RMS = 2.645%: 
	Cubic, RMS = 2.425%: 

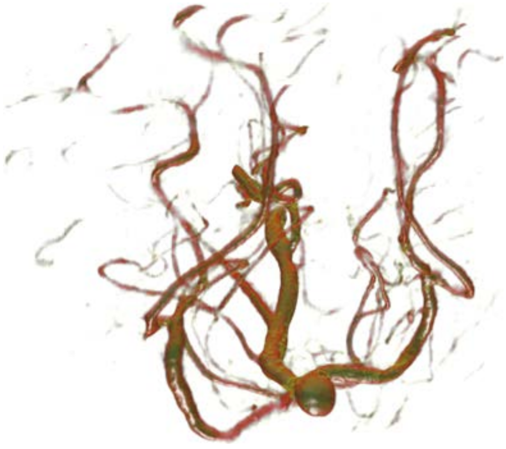
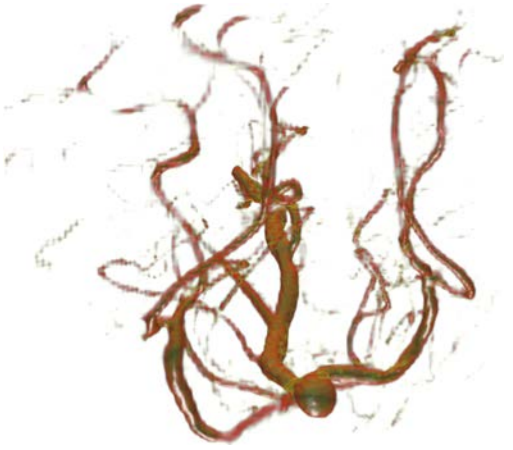
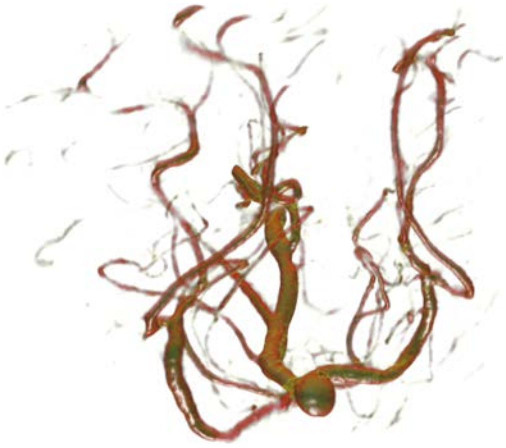
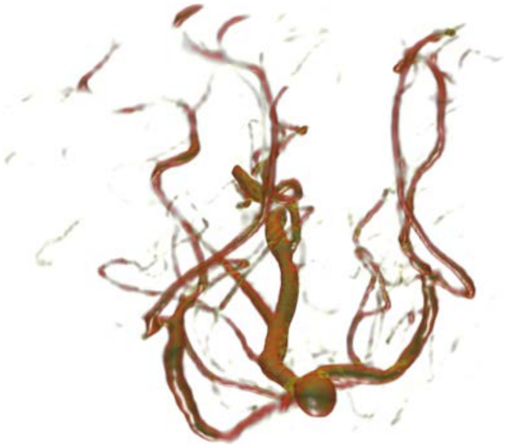
	Cubes	Blocks	Block pyramids	λ_p^n	f_λ	**Time**
Linear	3×1×1 = 3	3	levels 6-8	0.1	2	**6.81 s**
Cubic	6×1×2 = 12	12	levels 5-7	$5 \cdot 10^{-6}$	10	**60.81 s**

Table 9.5: **Bypass**, 7,929,856 points: comparison of linear and cubic pyramids.



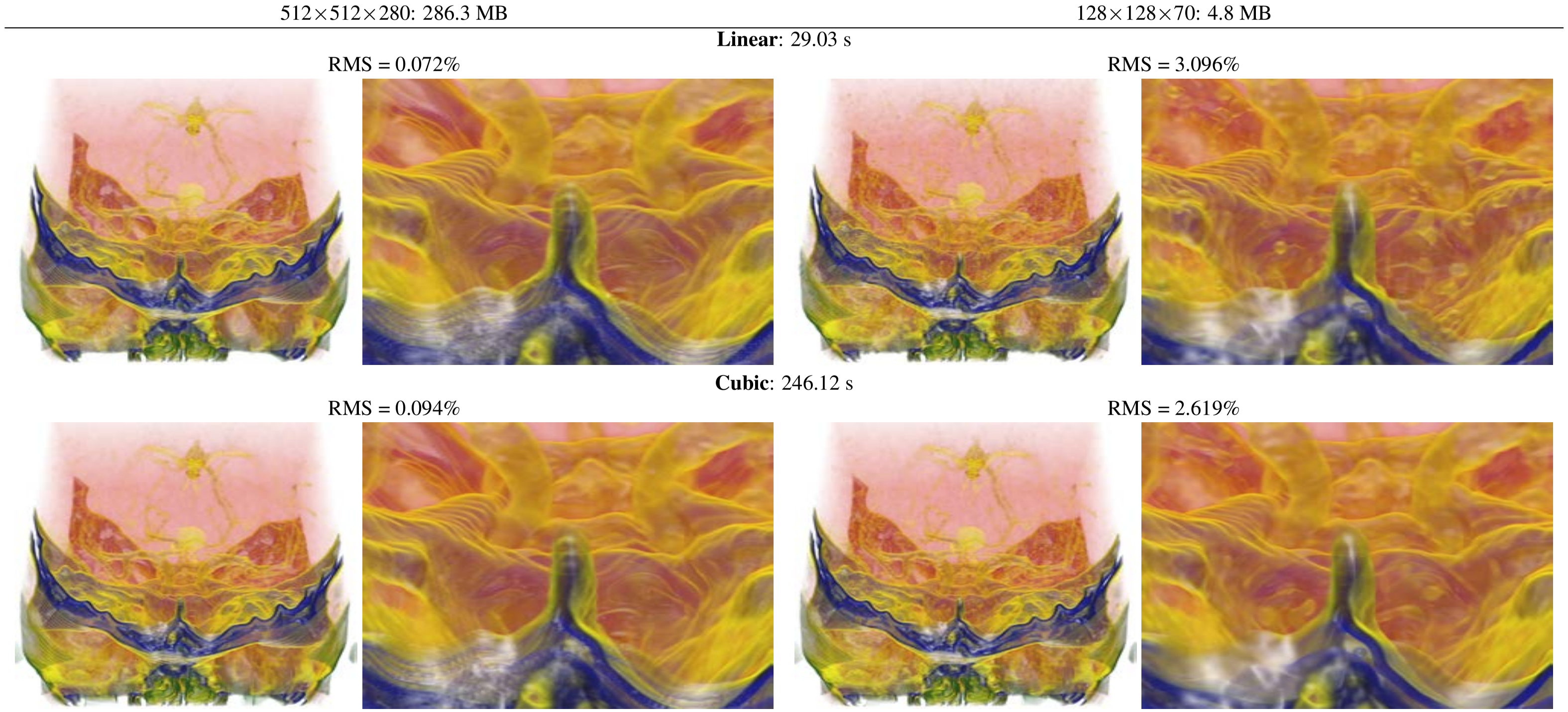
	Cubes	Blocks	Block pyramids	λ_p^n	f_λ
Linear	2×2×2 = 8	4	levels 5-7	0.5	2
Cubic	4×4×4 = 64	16	levels 4-6	$5 \cdot 10^{-6}$	10

Table 9.6: **Skull**, 3,355,444 points (20%) extracted from 256×256×256 grid: comparison of linear and cubic pyramids.

256×256×256: 66.3 MB	128×128×128: 8.6 MB
Linear: 23.43 s	
RMS = 0.267%	RMS = 1.043%
	
Cubic: 316.77 s	
RMS = 0.323%	RMS = 1.066%
	

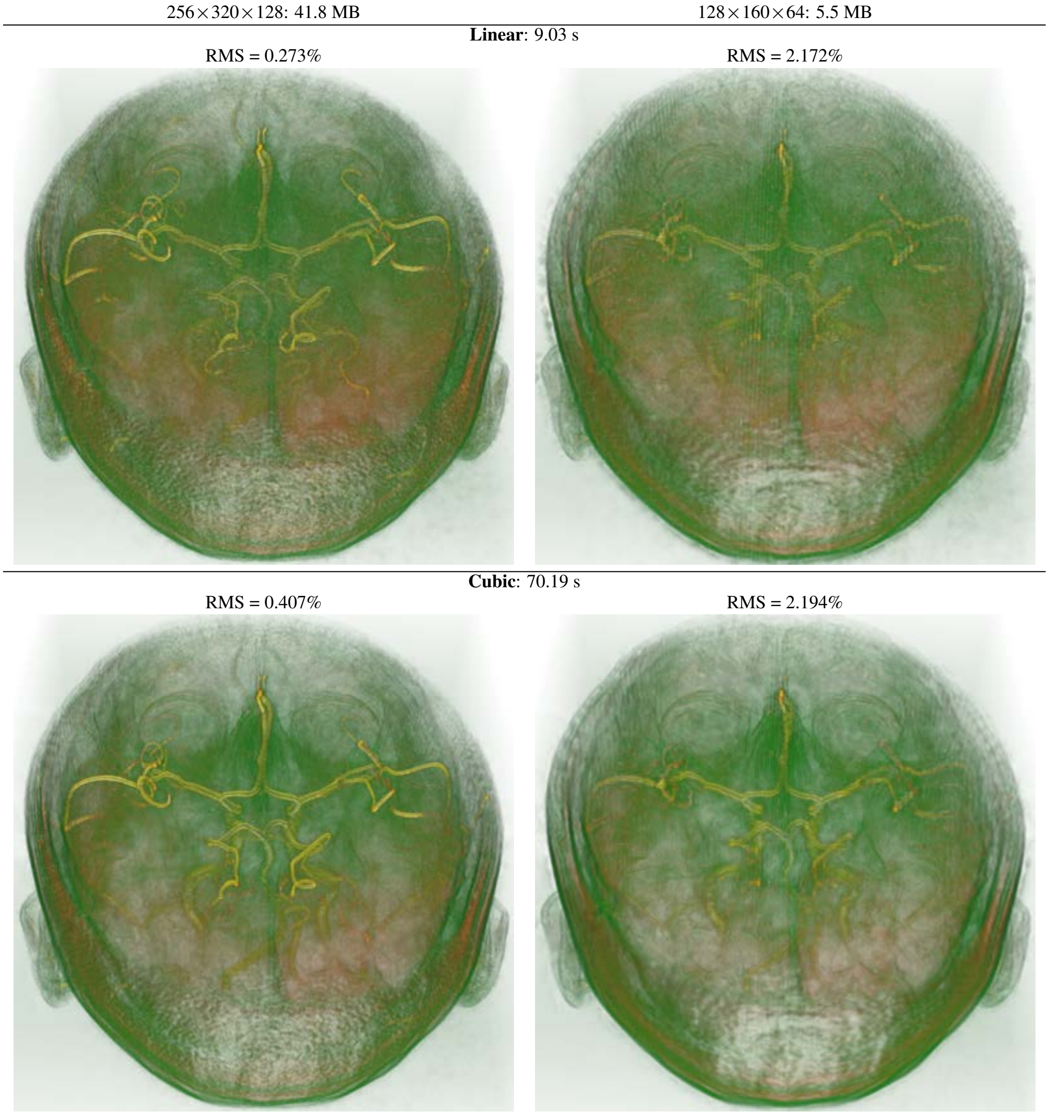
	Cubes	Blocks	Block pyramids	λ_p^n	f_λ
Linear	2×2×2 = 8	4	levels 5-7	0.5	2
Cubic	4×4×4 = 64	16	levels 4-6	$5 \cdot 10^{-5}$	10

Table 9.7: **Vertebra**, 26,843,546 points (20%) extracted from 512×512×512 grid: comparison of linear and cubic pyramids.



	Cubes	Blocks	Block pyramids	λ_p^n	f_λ
Linear	4×4×3 = 48	8	levels 5-7	0.5	2
Cubic	8×8×5 = 320	80	levels 4-6	10^{-6}	10

Table 9.8: **Brain**, 6,291,456 points (20%) extracted from anisotropic 512×512×120 grid: comparison of linear and cubic pyramids.



	Cubes	Blocks	Block pyramids	λ_p^{n}	f_λ
Linear	4×5×2 = 40	4	levels 4-6	0.5	2
Cubic	4×5×2 = 40	10	levels 4-6	$5 \cdot 10^{-6}$	10

Table 9.9: **Angio**, 2,097,152 points (20%) extracted from 256×320×128 grid: comparison of linear and cubic pyramids. The renderings of the coarser scales have been tweaked so as to compensate for higher translucency due to weaker gradients.

9.3 Limitations

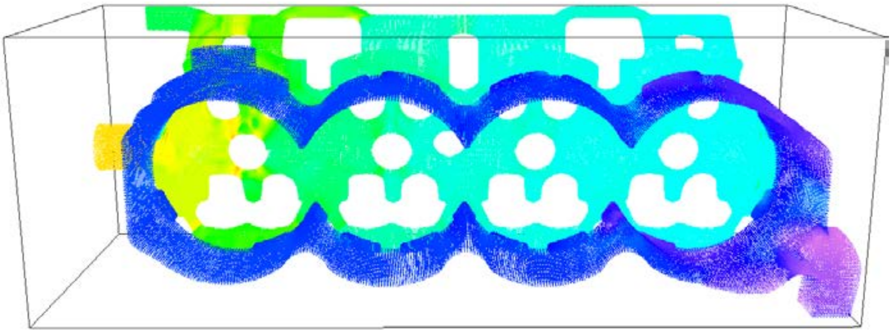
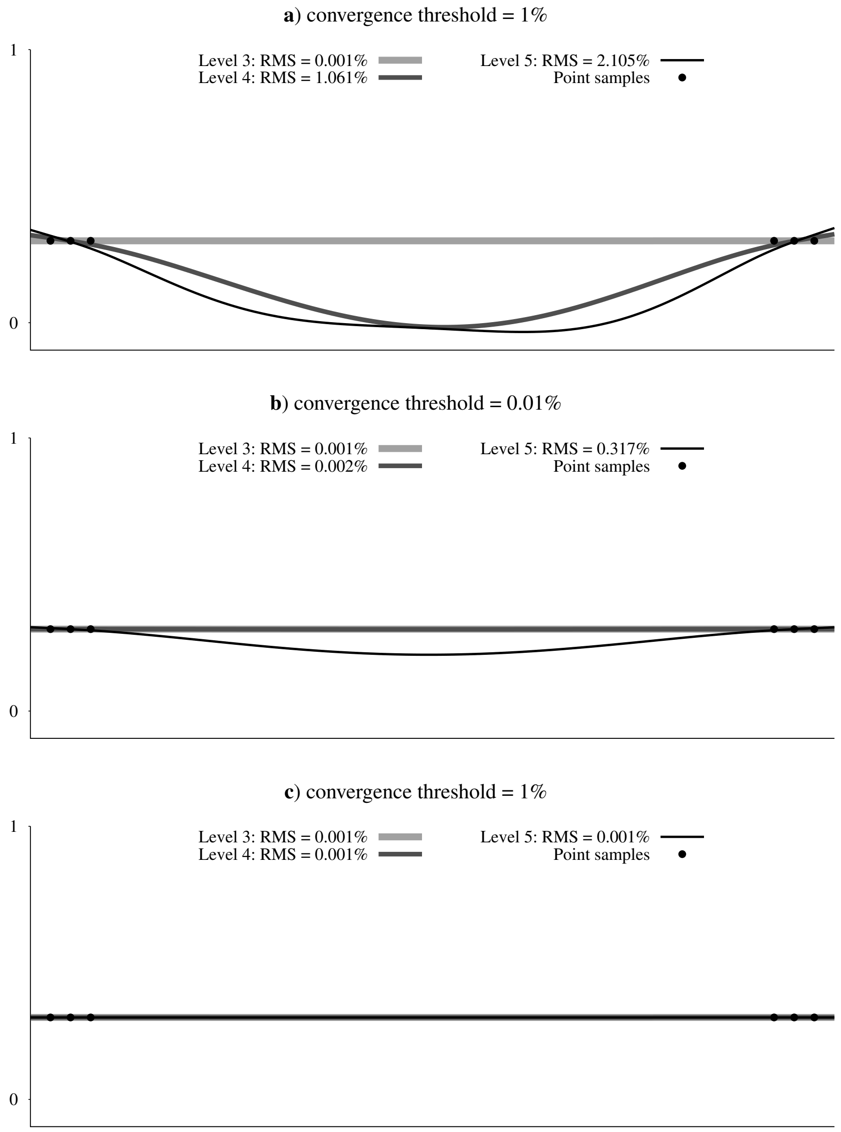


Figure 9.2: The Cooling Jacket (pressure) data set (1,537,898 points).

To illustrate the main limitation of our approach, we use a particularly challenging non-uniform data set: the Cooling Jacket (see Figure 9.2). Here, the point samples are only given in the interior of the object, and there is not a single point outside the object. This is a logical human conclusion after rendering the points. For our problem formulation on the other hand, this means that the signal is well-defined in the interior of the object, while there is not a single point indicating empty space (via null-samples with $f_i = 0$) and thereby defining the object's boundaries. So for all empty regions, variational reconstruction aims at minimizing the semi-norm, leading to wide extrapolation and potentially huge exceedances of the $[0, 1]$ range (recall Table 8.2 on page 61).

To illustrate the problem more clearly, we created a trivial 1D example (see Table 9.10). The analytical solution is clearly a straight line ($y = 0.3$), yielding a perfect interpolation of the 6 point samples as well as 0 for Duchon's semi-norms of order $p = 1$ and $p = 2$. If the absence of point samples inbetween the extremities actually meant that the signal should fall off to 0 and then rise to 0.3 again, we would generally need to include some corresponding point samples to make that clear. As can be seen from Table 9.10, we can also exploit a shortcoming of the iterative SOR solver and terminate the refinement early. This only works if the signal is initialized with zeros and the resolution is rather high, leading to a slow convergence in sparsely sampled regions.

The 3D results in Table 9.11 clearly show that the extrapolation decreases for finer resolutions (using a convergence threshold of 10%) if the signal is initialized with zeros. The pyramid's finer scale (figure **d**) is very similar to the coarser scale (analogous to figure **b**), its initialization. While the RMS error is smaller than the corresponding single-scale reconstruction (figure **c**), the latter is most likely more desirable due to sharper object boundaries (less extrapolation). Hence initializing finer scales with coarser results may not be desirable if extrapolation is to be limited. Zero-initialized multi-scale pyramids may diverge strongly across scales and thus may not be suited for our LoD scheme.

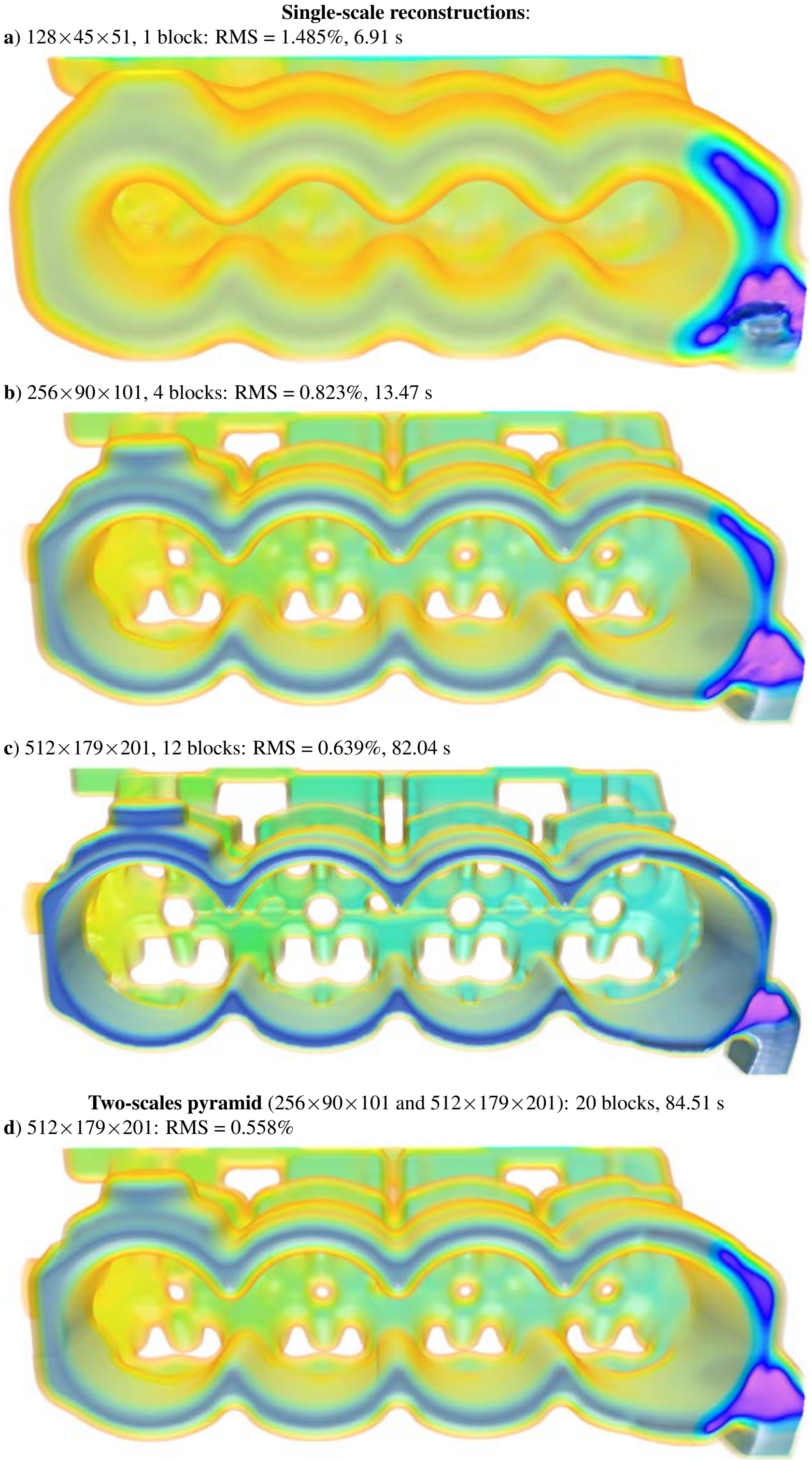


a) convergence threshold = 1%

b) convergence threshold = 0.01%

c) convergence threshold = 1%

Table 9.10: Impact of resolution, convergence threshold and signal initialization for a trivial 1D data set using levels 3-5 pyramids and $\lambda_2^{n} = 10^{-4}$ ($f_\lambda = 1$). All scales of figures **a** and **b** are initialized with zeros; for figure **c**, finer scales are initialized with coarser ones.



Single-scale reconstructions:

a) 128×45×51, 1 block: RMS = 1.485%, 6.91 s

b) 256×90×101, 4 blocks: RMS = 0.823%, 13.47 s

c) 512×179×201, 12 blocks: RMS = 0.639%, 82.04 s

Two-scales pyramid (256×90×101 and 512×179×201): 20 blocks, 84.51 s

d) 512×179×201: RMS = 0.558%

Table 9.11: Impact of resolution and signal initialization for the challenging Cooling Jacket (pressure) data set (1,537,898 points) using $\lambda_2^{\mathrm{n}} = 5 \cdot 10^{-5}$ (for all scales).

9.4 Comparison to Previous Work

Let us now compare our approach to previous work by Vuçini et al. [18] and Morozov et al. [14] which we consider state-of-the-art. All approaches are based on a regularized uniform B-spline approximation of a non-uniform point cloud. Vuçini et al. [18] use Lagrange's semi-norm of order $p = 2$ for regularization, Morozov et al. [14] use Duchon's semi-norm of order $p = 2$ as do we. Vuçini et al. [18] chose a constant $\lambda_2 = 0.3^3$ for the finest scale (and an implicit $f_\lambda = 1$), two coarser scales and 10 SOR iterations for each scale; to be able to reconstruct high resolutions, they also use a block-based approach. Morozov et al. [14] use a quite different implementation based on tensors and a conjugate-gradient solver and also exploit parallelizability.

Table 9.12 compares our results to those published by Vuçini et al. [18] for non-uniform data sets (not tested by Morozov et al. [14]). As can be seen, we yield significantly lower RMS errors (the only minimal exception being the Blunt-Fin data set). Our optimized implementation performs very well in terms of required run-time as well compared to the single-threaded implementation by Vuçini et al. [18], although the timings cannot be compared directly due to different test machines: they employ a not further specified Intel dual-core CPU @ 2.7 GHz and 6 GB RAM.

In Table 9.13, we compare the results for Laplacian data sets. The timings of Morozov et al. [14] have been obtained with a comparable Intel Core 2 Quad Q9400 CPU @ 2.67 GHz and 2 GB DDR2-800 memory.

	Points	Resolution	Our approach			Vuçini et al. [18]	
			λ_2^n	RMS error	Time	RMS error	Time
Oil	29,094	37×40×37	$5 \cdot 10^{-6}$	0.004%	0.80 s	0.190%	4.20 s
Blunt-Fin	40,960	93×35×25	$5 \cdot 10^{-5}$	1.144%	0.49 s	1.140%	7.20 s
Natural Convection	68,921	61×61×61	10^{-6}	0.001%	1.58 s	0.630%	4.20 s
Synthetic Chirp	75,000	64×64×64	10^{-6}	0.005%	2.47 s	1.120%	4.80 s
Cooling Jacket (pressure)	1,537,898	256×90×101	$5 \cdot 10^{-5}$	0.823%	13.47 s	0.920%	138.00 s
Bypass	7,929,856	768×91×191	$5 \cdot 10^{-7}$	0.175%	59.36 s	0.610%	384.00 s

Table 9.12: Comparison of our approach to Vuçini et al. [18] for non-uniform data sets. We use 2 coarser scales (exceptions: Blunt-Fin (3 coarser scales) and Cooling Jacket (no coarser scale)) and $f_\lambda = 10$. The first 4 data sets are reconstructed as single blocks while we use 4 blocks for the Cooling Jacket and 16 blocks for the Bypass data set; the block padding is fixed at 4 intervals at each side.

	Resolution	Our approach					[14]		[18]	
		λ_2^n	Blocks	RMS	RMS_g	Time	RMS_g	Time	RMS_g	Time
Engine (20%)	256×256×128	$5.0 \cdot 10^{-6}$	4	0.22%	0.54%	45.9 s	0.58%	67.2 s	0.94%	76.8 s
Tooth (20%)	256×256×161	$2.5 \cdot 10^{-5}$	7	0.45%	0.41%	74.0 s	0.29%	78.6 s	0.57%	112.8 s
CT-Chest (20%)	384×384×240	$2.0 \cdot 10^{-5}$	28	0.49%	0.39%	192.9 s	0.32%	262.8 s	0.60%	304.8 s
Carp (20%)	256×256×512	$5.0 \cdot 10^{-6}$	16	0.20%	0.42%	219.1 s	0.30%	288.6 s	0.36%	343.8 s

Table 9.13: Comparison of our approach to Morozov et al. [14] as well as Vuçini et al. [18] for Laplacian data sets. We use a single scale for better RMS_g results.

CHAPTER 10

Mixed-Resolution LoDs

By transitioning from a coarser resolution level to the finer one, the number of coefficients is approximately increased by a factor of $2^d = 8$. This implies large steps memory-wise. To fit a memory budget more closely, we introduced the LoD scheme in chapter 5, thereby selecting a varying resolution level for each cube while trying to minimize the global RMS error. The cubes are fused and then merged to blocks for the LoD BIH tree.

Tables 10.1 to 10.4 compare linear and cubic LoDs. We have already seen that cubic B-splines are better suited to represent signals at coarser scales. As expected, the C^2 continuity at block boundaries also pays off for mixed-resolution LoDs and significantly reduces visual interscale discontinuities (see Figure 10.1 and Table 10.5 for close-ups). These visual discontinuities occur if the coarse and fine blocks diverge significantly at the boundary. Figure 10.2 and Table 10.6 present some LoD examples for non-uniform data sets.

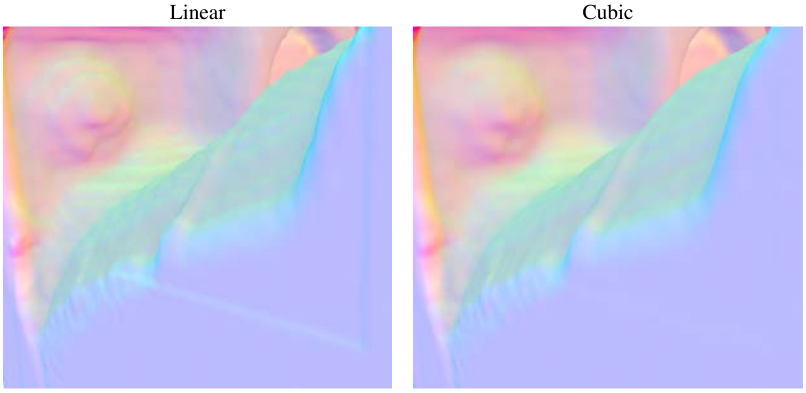


Figure 10.1: Direct comparison between linear and cubic mixed-resolution LoDs (the ≈ 6.7 MB LoDs for the Engine (20%) data set from Tables 10.1 and 10.2) for a zoomed interscale boundary. The renderings represent color-coded gradients of an iso-surface.

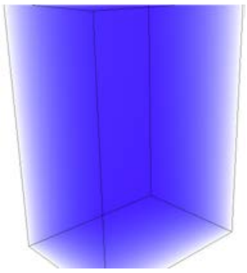
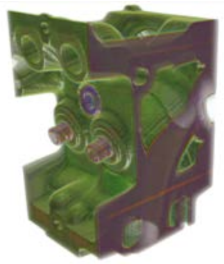
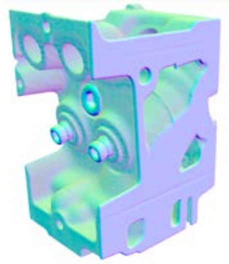
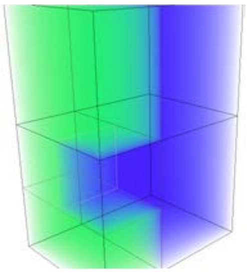
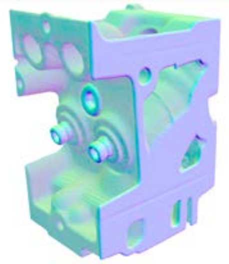
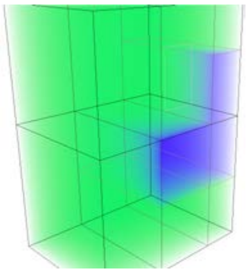
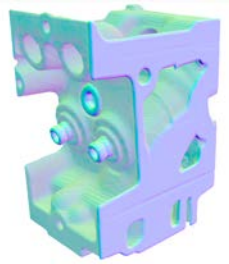
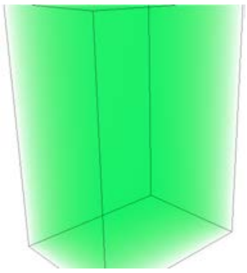
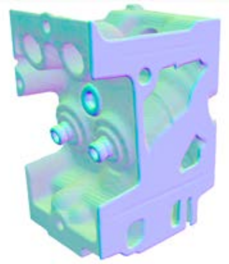
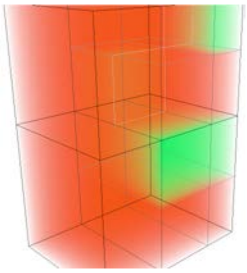
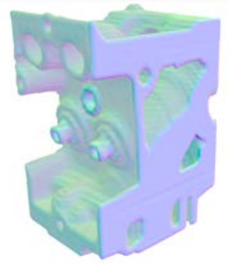
LoD size	LoD blocks	DVR	Iso-surface
20.20 MB	1	RMS = 0.100%	
			
14.89 MB	6	RMS = 1.421% (+2.00%)	
			
6.72 MB	10	RMS = 2.335% (+1.03%)	
			
2.67 MB	1	RMS = 2.805%	
			
1.11 MB	12	RMS = 5.291% (+2.51%)	
			

Table 10.1: Comparison of different **linear** LoDs for the Engine (20%) data set, reconstructed using 3×4×2 = 24 cubes, 6 blocks and levels 4-6 pyramids. Red blocks represent resolution level 4, green ones level 5 and blue ones level 6. The percentage in parenthesis represents the *relative* increase of the RMS error due to interscale cubes fusion.

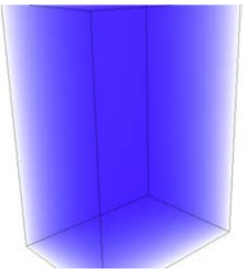
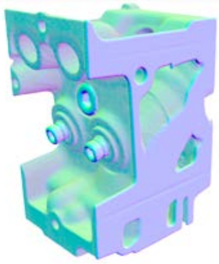
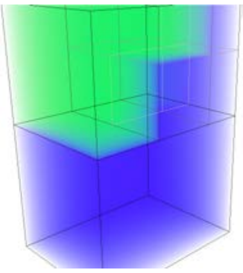
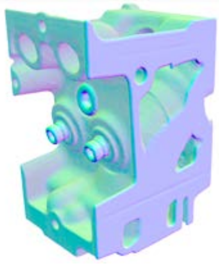
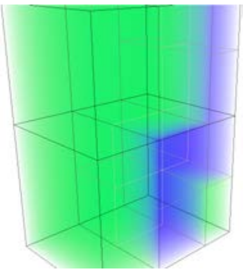
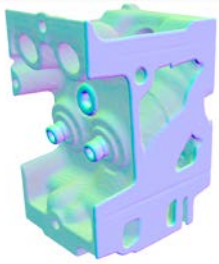
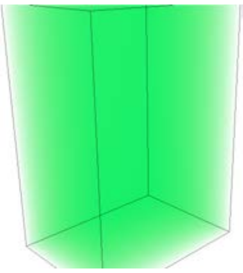
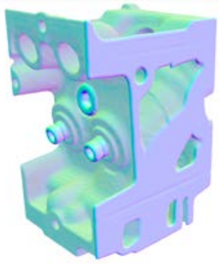
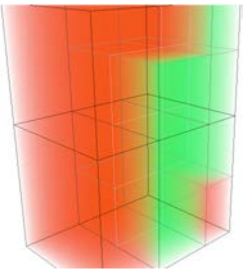
LoD size	LoD blocks	DVR	Iso-surface
20.20 MB	1	RMS = 0.135%	
			
14.45 MB	9	RMS = 0.880% (+2.19%)	
			
6.60 MB	14	RMS = 1.379% (+0.68%)	
			
2.67 MB	1	RMS = 1.560%	
			
1.13 MB	16	RMS = 4.874% (+2.16%)	
			

Table 10.2: Comparison of different **cubic** LoDs for the Engine (20%) data set, reconstructed using 3×4×2 = 24 cubes, 6 blocks and levels 4-6 pyramids. Red blocks represent resolution level 4, green ones level 5 and blue ones level 6. The percentage in parenthesis represents the *relative* increase of the RMS error due to interscale cubes fusion.

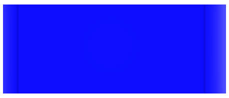
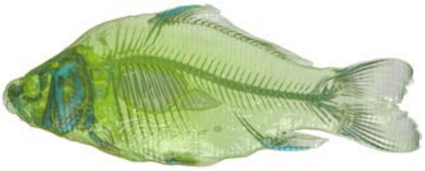
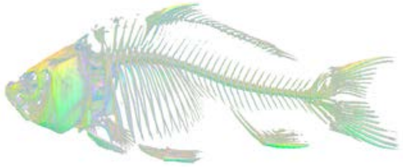
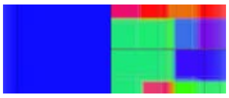
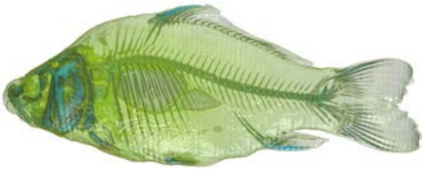
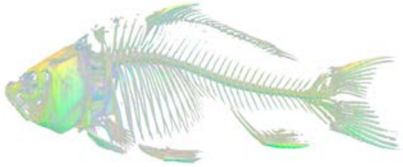
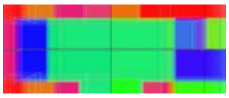
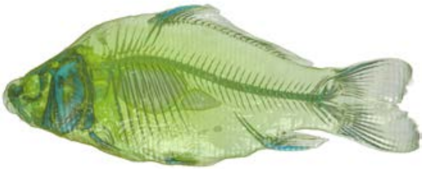
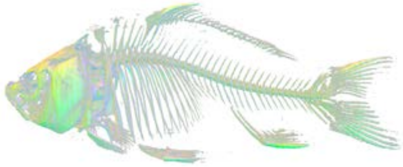
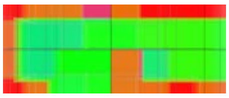
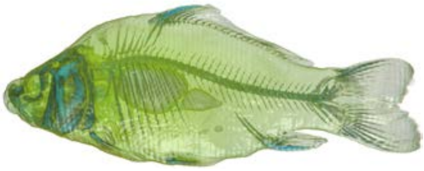
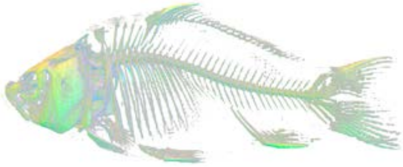
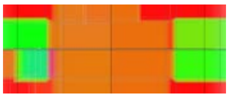
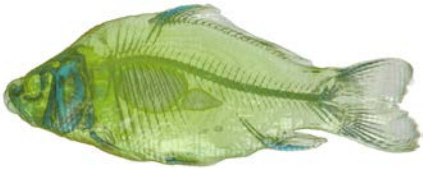
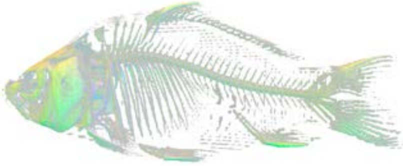
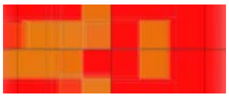
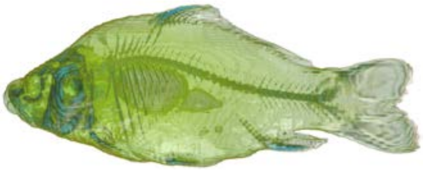
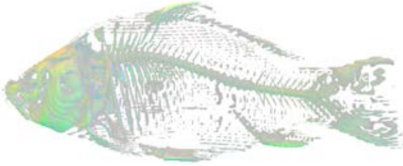
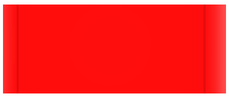
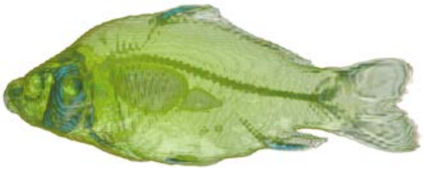
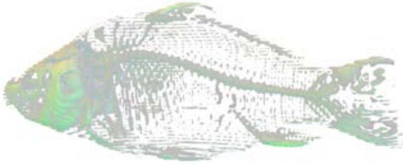
LoD	DVR	Iso-surface
131.8 MB, 1 block	RMS = 0.338%, RMS$_g$ = 2.193%	
		
89.2 MB, 32 blocks	RMS = 0.388% (+2.51%), RMS$_g$ = 2.251%	
		
52.2 MB, 66 blocks	RMS = 0.517% (+5.77%), RMS$_g$ = 2.520%	
		
26.3 MB, 67 blocks	RMS = 0.880% (+7.14%), RMS$_g$ = 3.151%	
		
12.0 MB, 46 blocks	RMS = 1.372% (+7.02%), RMS$_g$ = 3.800%	
		
5.6 MB, 41 blocks	RMS = 2.059% (+5.30%), RMS$_g$ = 4.576%	
		
2.2 MB, 1 block	RMS = 2.922%, RMS$_g$ = 5.427%	
		

Table 10.3: Comparison of different **linear** LoDs for the Carp (20%) data set, reconstructed in 37.6 seconds using 4×4×8 = 128 cubes, 4 blocks and levels 4-6 pyramids. Red blocks represent resolution level 4, green ones level 5 and blue ones level 6. The percentage in parenthesis represents the *relative* increase of the RMS error due to interscale cubes fusion.

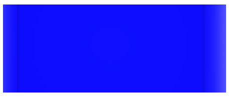
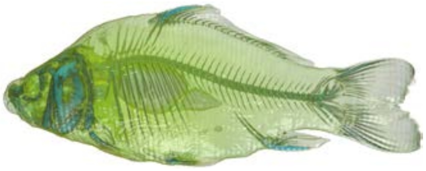
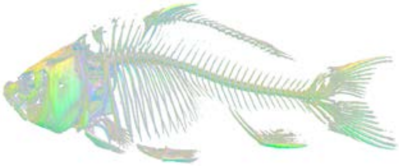
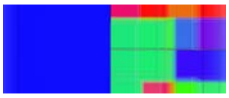
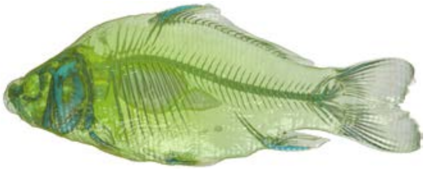
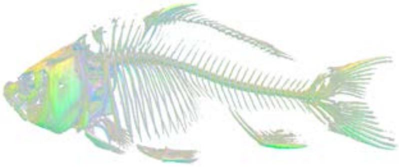
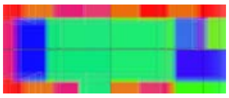
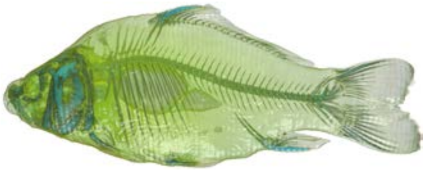
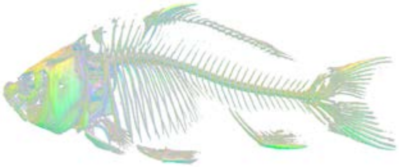
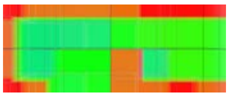
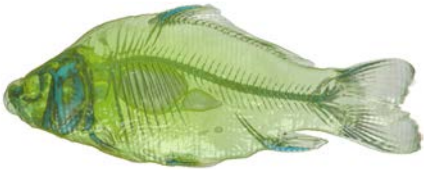
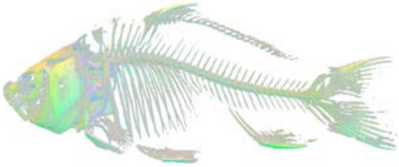
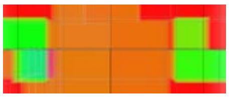
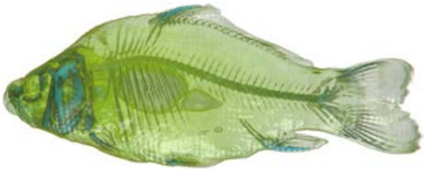
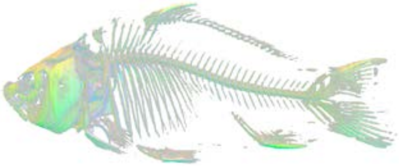
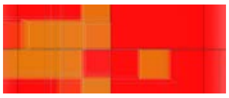
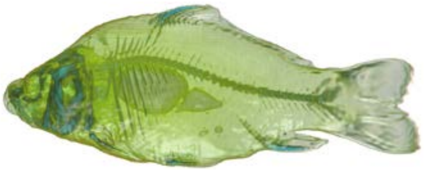
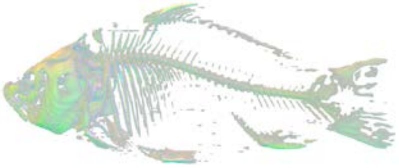
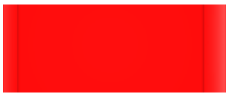
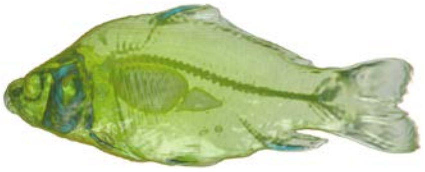
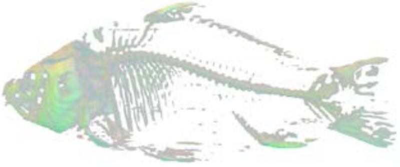
LoD	DVR	Iso-surface
131.8 MB, 1 block	RMS = 0.196%, $\mathrm{RMS_g}$ = 0.377%	
		
88.2 MB, 33 blocks	RMS = 0.261% (+5.41%), $\mathrm{RMS_g}$ = 0.402%	
		
52.2 MB, 67 blocks	RMS = 0.394% (+9.95%), $\mathrm{RMS_g}$ = 0.450%	
		
26.2 MB, 65 blocks	RMS = 0.745% (+9.61%), $\mathrm{RMS_g}$ = 0.548%	
		
12.0 MB, 49 blocks	RMS = 1.221% (+8.42%), $\mathrm{RMS_g}$ = 0.725%	
		
5.6 MB, 38 blocks	RMS = 1.855% (+5.97%), $\mathrm{RMS_g}$ = 0.989%	
		
2.2 MB, 1 block	RMS = 2.690%, $\mathrm{RMS_g}$ = 1.343%	
		

Table 10.4: Comparison of different **cubic** LoDs for the Carp (20%) data set, reconstructed in 251.1 seconds using 4×4×8 = 128 cubes, 32 blocks and levels 4-6 pyramids. Red blocks represent resolution level 4, green ones level 5 and blue ones level 6. The percentage in parenthesis represents the *relative* increase of the RMS error due to interscale cubes fusion.

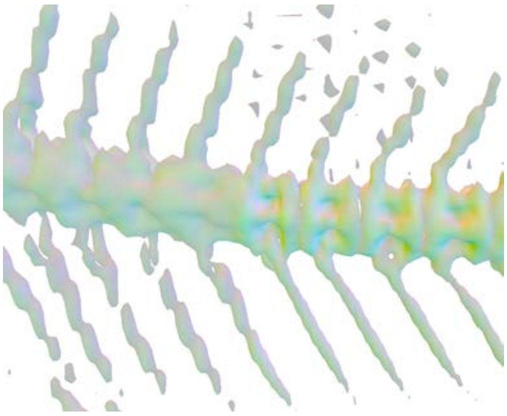
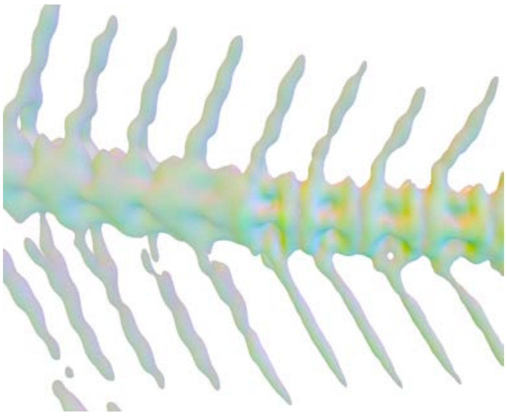
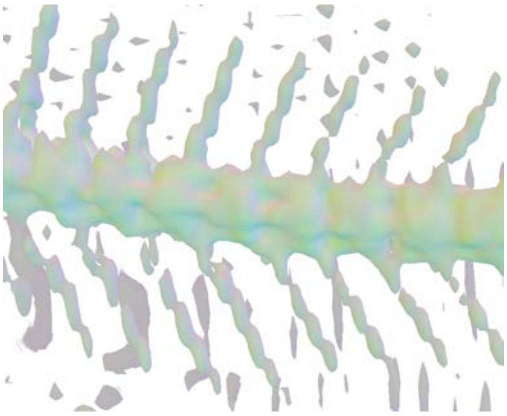
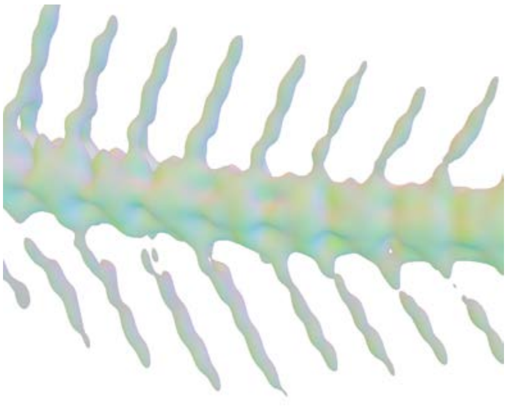
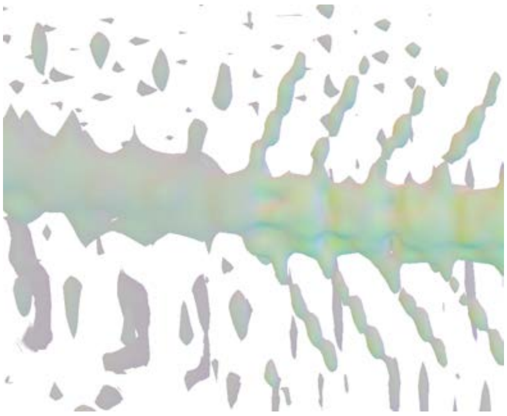
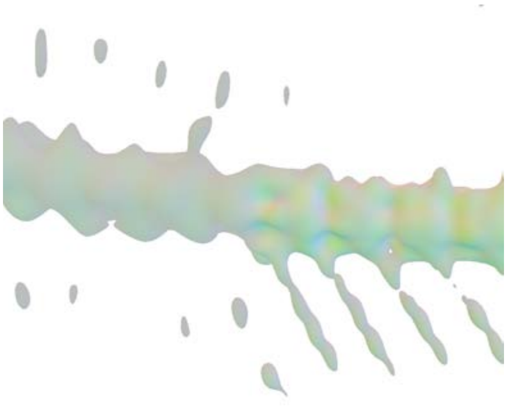
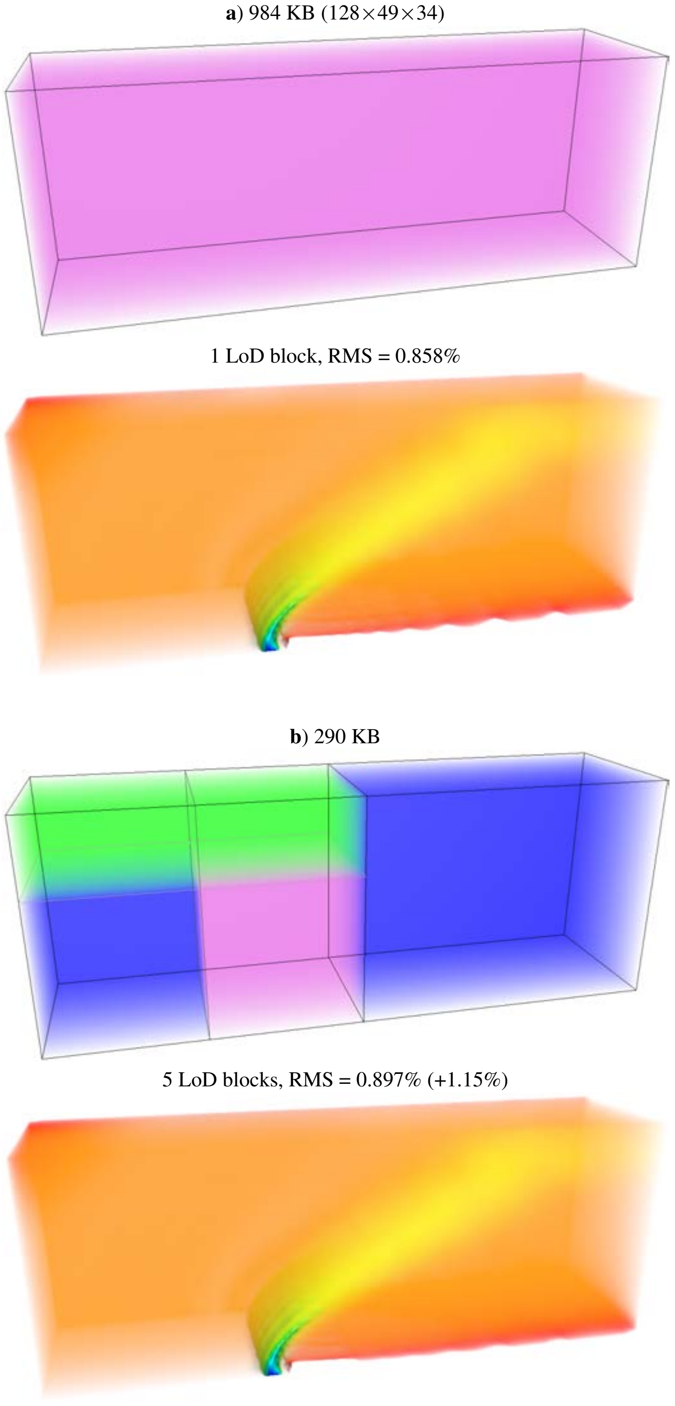
Linear	Cubic
LoD size: $\approx$ 26 MB	
	
LoD size: 12 MB	
	
LoD size: 5.6 MB	
	

Table 10.5: Comparison between linear and cubic mixed-resolution LoDs (from Tables 10.3 and 10.4) for zoomed interscale boundaries. For the 12 MB LoDs, all visible front regions are given at resolution level 5 (green blocks). For the $\approx$ 26 MB LoDs, the front bottom right region is enhanced to resolution level 6 (blue). For the 5.6 MB LoDs, the front left region is reduced to level 4 (red); for the cubic LoD, the front top right region is reduced to level 4 as well.



a) 984 KB (128×49×34)

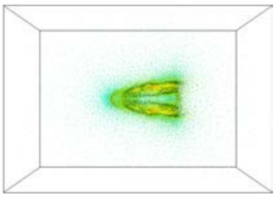
1 LoD block, RMS = 0.858%

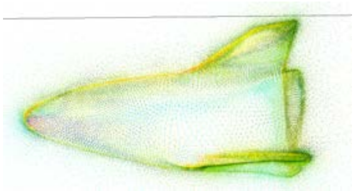
b) 290 KB

5 LoD blocks, RMS = 0.897% (+1.15%)

Figure 10.2: Comparison between two cubic LoDs for the Blunt-Fin data set (compare to Table 8.18 on page 77). Green blocks represent resolution level 3, blue ones level 4 and violet ones level 5.

X38: 331,163 points





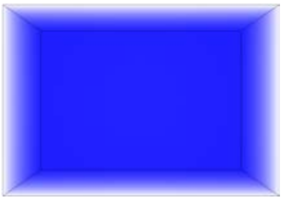
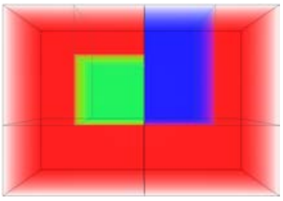
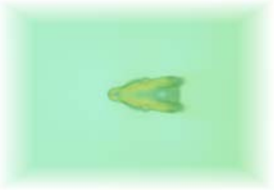
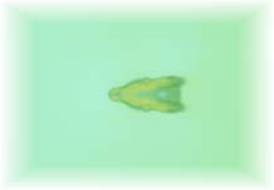
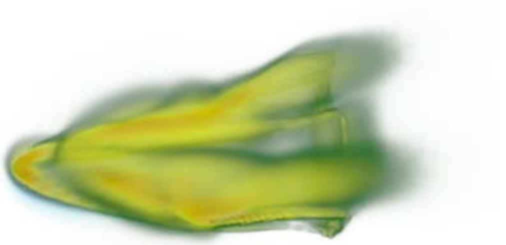
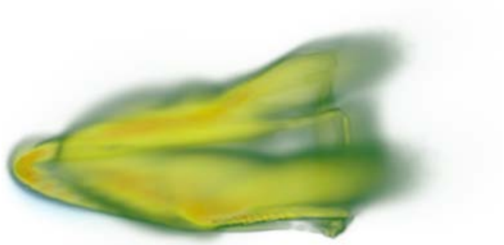
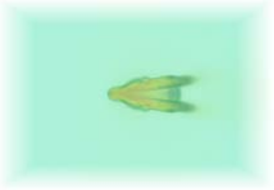
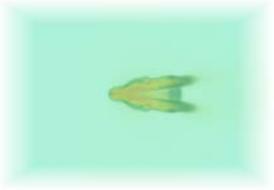
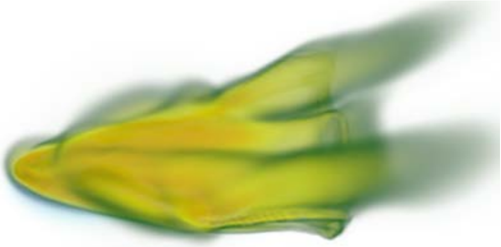
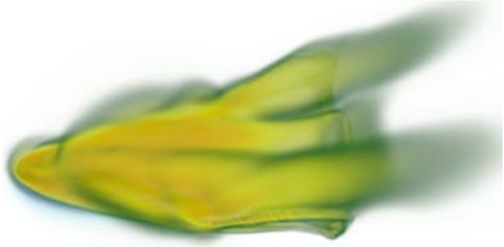
LoD size: 21.7 MB (250×173×125)	LoD size: 5.3 MB
1 LoD block	8 LoD blocks
	
Linear ($\lambda_1^{\mathfrak{n}} = 10^2, f_\lambda = 2$): 6.84 s	
	
RMS = 2.202%	RMS = 2.243% (+0.58%)
	
Cubic ($\lambda_2^{\mathfrak{n}} = 5 \cdot 10^{-4}, f_\lambda = 4$): 20.46 s	
	
RMS = 2.032%	RMS = 2.062% (+0.60%)
	

Table 10.6: Comparison between two linear and two cubic LoDs for the challenging X38 data set subdivided into 4×3×2 cubes merged to a single block and reconstructed via a levels 1-6 pyramid. Red LoD blocks represent resolution level 4, green ones level 5 and blue ones level 6.

CHAPTER 11

GPU Ray-Casting Performance

Direct Volume Rendering is a computationally intensive task. Suppose we have a viewport of 1,000 × 1,000 pixels (1 MPixels) and the volume covers all pixels. We then need to cast 1 million rays for each rendered frame. For each ray, we need to traverse the BIH to identify blocks hit by the ray and then sample the ray segments. Suppose we choose a sampling step length so that each ray is sampled at an average of 1,000 steps. The total number of ray samples for a rendering will then accumulate to 1 billion. To evaluate a linear B-spline at a sample location, we need a single trilinear lookup and additionally 6 trilinear lookups to compute a continuous gradient by central differencing. For a cubic B-spline, we need 8 trilinear lookups plus some lerping for a sample and additionally $3 \cdot 8$ trilinear look-ups (+ lerping) for the gradient. Each sample is mapped to color and opacity via a TF lookup; if $\alpha > 0$, the sample is shaded (requiring the normalization of the gradient), its opacity corrected (implying a pow() invocation) etc. – i.e., current GPUs are easily brought to their limits, especially for cubic B-splines.

Crucial factors influencing the ray-casting performance are:

- The ray sampling step length. Scaling it by two cuts the number of ray samples in half, so that the frame-rate scales pretty well with the sampling step length.

- Viewport resolution or, more precisely, the number of pixels covered by the volume to be rendered. This number is equal to the number of rays to be cast, so that the frame-rate is approximately inversely proportional. Higher resolutions favor the coherence between neighboring pixels processed in parallel (GPU thread wave-fronts), so in practice the frame-rate is scaled by a little more than $\frac{1}{4}$ when doubling width and height.

- The size of the BIH tree. Finer subdivisions increase the traversal overhead and lead to less coherence of GPU thread wave-fronts. Additionally, the size of the array for the stack needs to be at least equal to the maximum BIH depth, therefore deeper trees require more of the precious General Purpose Registers (GPRs). A higher GPR usage restricts the number of total GPU threads, which in turn decreases the GPU's ability to hide memory latency.

- The opacity of the transfer function. More opaque TFs favor early ray termination which can significantly decrease the number of required ray samples.

Table 11.1 illustrates the impact of the ray sampling step length (and the number of pixels covered by the volume). For all of our renderings, we use a viewport resolution of about 1,355 × 825 pixels ($\approx$ 1.12 MPixels). Equidistant sampling results in ringing artifacts if the step length is too large for the given signal/TF combination because we use a central (perspective) projection. These undersampling artifacts caused by an insufficiently accurate approximation of the rendering integral are more notable in zoomed renderings.

The frame-rate usually does not drop much when rendering finer resolutions. For example, the frame-rates for the Engine at $64 \times 64 \times 32$ ($\approx \frac{1}{64}$ of the coefficients compared to Table 11.1) are almost identical. There are exceptions though, although the impact is not huge. Table 11.2 represents such an exception and also illustrates the impact of the BIH tree on the rendering performance. In Tables 11.3 and 11.4, we analyze the impact of trees and resolutions for different LoDs and specify frame-rates for iso-surface renderings, which benefit greatly by adaptive sampling. The latter tables also show that the algorithm for fitting the blocks into a single 3D texture works pretty well and usually achieves a packing efficiency (defined as the sum of the block volumes divided by the texture volume) between 75% and 90%. The size of the 3D texture is significantly smaller than the LoD size (exactly 50% for a perfect packing efficiency) due to the usage of the half-precision format (2 bytes per coefficient vs. 4 bytes for the single-precision format).

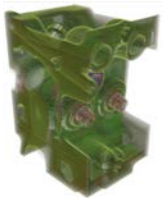
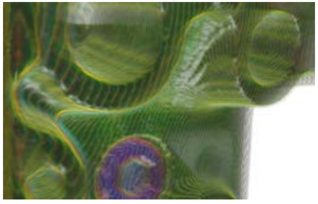
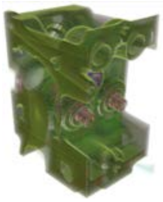
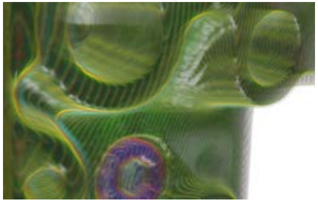
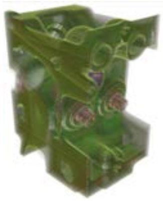
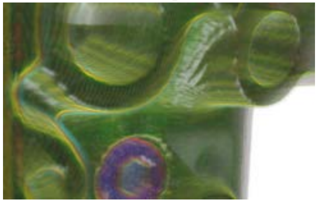
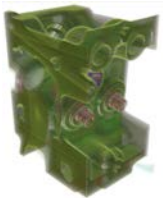
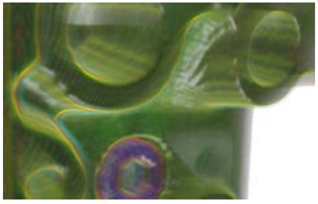
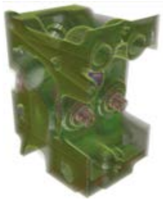
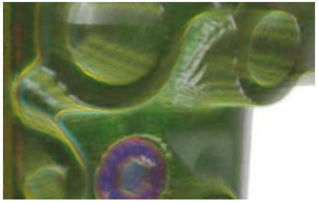
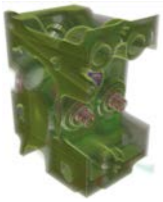
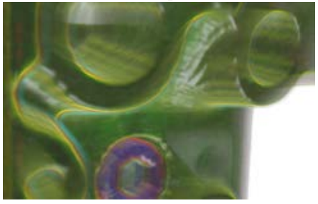
Samples	Linear		Cubic	
256	66.8 fps 	38.7 fps 	18.9 fps 	10.2 fps 
512	35.5 fps 	20.0 fps 	9.7 fps 	5.1 fps 
1,024	18.2 fps 	10.1 fps 	4.9 fps 	2.5 fps 

Table 11.1: Impact of the ray sampling step length (determined by the given number of samples along the volume's longest axis) in terms of visual artifacts and rendering performance. We used 20% of the Engine data set and rendered it at a resolution of 256×256×128. The zoomed renderings cause all pixels to be covered by the volume, leading to about twice as many rays to be cast with respect to the narrower full rendering. This results in the frame-rate being approximately halved.

Resolution	BIH leafs (depth)	Linear			Cubic		
		256 samples	512 samples	1,024 samples	256 samples	512 samples	1,024 samples
512×512×280	1 (0)	45.3 fps	24.2 fps	12.6 fps	11.1 fps	5.8 fps	3.0 fps
(286.3 MB)	320 (9)	18.8 fps	10.4 fps	5.6 fps	6.3 fps	3.3 fps	1.7 fps
128×128×70	1 (0)	53.6 fps	28.0 fps	14.3 fps	13.0 fps	6.6 fps	3.3 fps
(4.8 MB)	320 (9)	21.7 fps	11.7 fps	6.1 fps	7.2 fps	3.7 fps	1.9 fps

Table 11.2: Impact of the LoD BIH tree on the rendering performance for the Brain (20%) data set (see Table 9.8 on page 89). For the BIH featuring a single leaf, we merged all cubes to a single LoD block. For the other BIH, we kept all 320 cubes separated to represent an example for more complex BIH trees. We see that cubic renderings incur a performance hit of slightly less than 50% while linear ones suffer from even higher frame-rate drops.

LoD size	BIH leafs (depth)	3D texture			DVR	Iso-surface	
		Resolution	Size	Efficiency	2,048 samples	474 samples	2x adaptive
Linear:							
20.20 MB	1 (0)	171×240×129	10.1 MB	100%	13.9 fps	229.0 fps	229.8 fps
14.89 MB	6 (4)	214×179×129	9.4 MB	79%	9.4 fps	167.9 fps	206.9 fps
6.72 MB	10 (5)	116×134×132	3.9 MB	86%	9.6 fps	176.0 fps	270.7 fps
2.67 MB	1 (0)	87×122×66	1.3 MB	100%	13.9 fps	254.1 fps	428.7 fps
1.11 MB	12 (5)	77×70×70	0.7 MB	77%	9.4 fps	175.0 fps	389.1 fps
Cubic:							
20.20 MB	1 (0)	171×240×129	10.1 MB	100%	3.4 fps	41.1 fps	41.3 fps
14.45 MB	9 (5)	171×131×195	8.3 MB	87%	2.9 fps	36.6 fps	45.9 fps
6.60 MB	14 (5)	133×134×131	4.5 MB	74%	2.9 fps	36.1 fps	58.7 fps
2.67 MB	1 (0)	87×122×66	1.3 MB	100%	3.4 fps	42.3 fps	77.3 fps
1.13 MB	16 (5)	71×73×73	0.7 MB	78%	2.8 fps	36.3 fps	95.6 fps

Table 11.3: Impact of the LoD tree and resolutions on the rendering performance, based on the mixed-resolution LoDs in Tables 10.1 and 10.2 for the Engine (20%) data set. Adaptive 2x oversampling corresponds to 474 equidistant samples for the highest resolution (the ≈20 MB LoDs). The two iso-surface renderings per LoD are virtually identical, so adaptive sampling pays off here.

LoD size	BIH leafs (depth)	3D texture			DVR	Iso-surface	
		Resolution	Size	Efficiency	2,048 samples	2,048 samples	4x adaptive
Linear:							
131.79 MB	1 (0)	259×259×515	65.89 MB	100%	8.6 fps	24.9 fps	24.9 fps
89.15 MB	32 (7)	326×268×326	54.33 MB	82%	4.8 fps	14.9 fps	20.3 fps
52.20 MB	66 (7)	268×236×262	31.61 MB	83%	4.3 fps	13.6 fps	23.5 fps
26.30 MB	67 (7)	204×201×201	15.72 MB	84%	4.2 fps	13.7 fps	28.5 fps
11.97 MB	46 (7)	153×172×140	7.03 MB	85%	4.4 fps	14.5 fps	37.3 fps
5.61 MB	41 (7)	124×124×105	3.08 MB	91%	4.5 fps	15.2 fps	45.7 fps
2.24 MB	1 (0)	67×67×131	1.12 MB	100%	9.2 fps	28.6 fps	104.1 fps
Cubic:							
131.79 MB	1 (0)	259×259×515	65.89 MB	100%	2.3 fps	4.0 fps	4.0 fps
88.22 MB	33 (7)	326×268×326	54.33 MB	81%	1.7 fps	3.3 fps	4.5 fps
52.20 MB	67 (7)	268×236×262	31.61 MB	83%	1.6 fps	3.1 fps	5.6 fps
26.23 MB	65 (7)	188×204×204	14.92 MB	88%	1.6 fps	3.1 fps	7.2 fps
11.99 MB	49 (7)	153×169×140	6.90 MB	87%	1.6 fps	3.2 fps	8.8 fps
5.56 MB	38 (7)	114×114×137	3.40 MB	82%	1.7 fps	3.4 fps	11.0 fps
2.24 MB	1 (0)	67×67×131	1.12 MB	100%	2.4 fps	4.2 fps	16.1 fps

Table 11.4: Impact of the LoD tree and resolutions on the rendering performance, based on the mixed-resolution LoDs in Tables 10.3 and 10.4 for the Carp (20%) data set. Adaptive 4x oversampling corresponds to 2,048 equidistant samples for the highest resolution (the ≈132 MB LoDs).

Part III

End

CHAPTER 12

Conclusions

12.1 Summary

We presented a framework to reconstruct and render a scalar d-dimensional field based on non-uniform point samples by using uniform B-spline pyramids, focusing on $d = 3$. Uniform B-splines, due to their optimal support for a given degree of differentiability, perfectly suit both key requirements of the reconstructed field:

- providing a highly efficient, scalable and numerically stable way to compute an accurate discretized representation of the field based on the point samples, and
- allowing for efficient evaluation based on its discretized representation to favor ray-casting performance. This also means that we are able to render the reconstructed field directly and do not need to approximate the field via interpolation.

Additionally, B-splines of odd degree allow to exploit their interscale relation to derive computationally very cheap and accurate coarser representations to be used for LoDs/mip-maps by reconstructing a multi-scale B-spline pyramid. At the same time, a multi-scale pyramid significantly increases the convergence rate of the iterative SOR solver, especially for high resolutions.

We developed a fusing scheme capable of guaranteeing global C^{n-1} continuity for a subdivision into blocks, even across scales. For the presented scheme to work, the block boundaries need to be aligned on shared knots, and the resolutions may only differ by powers of two. These constraints are handled by subdividing the volume into a Cartesian lattice of cubes and using powers of two as resolution levels for each cube. These cubes represent atomic primitives and may be adaptively merged to larger cuboid blocks by employing a binary BIH tree built on top of the cubes. A subdivision into blocks allows to reconstruct large volumes requiring huge amounts of RAM and improves parallelizability to better exploit multiple CPU cores. At the same time, it allows to adapt a block's highest resolution level and to skip empty blocks altogether. We have seen that redundancy in form of temporary block paddings during reconstruction is crucial to obtain almost equivalent results to global reconstruction. Fusing blocks in a shared scale will then work very well. Our minimalistic interscale fusing scheme requires the blocks of differing resolution levels not to diverge significantly at the boundaries, otherwise notable artifacts result, especially for linear B-splines. The C^2 continuity of cubic B-splines effectively reduces these artifacts and generally yields visually more pleasing renderings, especially for coarser resolutions and zoomed renderings. These advantages compared to linear B-splines imply a way higher run-time for reconstruction and lower ray-casting performance.

By selecting a suited scale for each cube and employing a BIH tree to merge cubes to blocks, we may derive different volume LoDs. The blocks represent autonomous parts of the global B-spline at potentially varying resolution levels. Redundant outer coefficients allow to directly sample the global B-spline by sampling the corresponding block's B-spline, thereby favoring ray-casting performance and scalability. We presented a heuristic to fit the block coefficient grids into a single 3D texture as work-around for current GPU limitations and exploit the GPU's native support of the half-precision floating-point format to reduce memory requirements. The GPU is then used to cast a ray for each pixel and to sample the volume along each ray. The evaluation of linear and cubic B-splines (and derivatives thereof) is tailored for GPUs, allowing to exploit dedicated hardware for linear filtering. We have seen that current high-end GPUs are able to deliver nice direct volume renderings at interactive frame-rates and iso-surface renderings at real-time frame-rates.

12.2 Our Contributions

Let us summarize our main contributions to the approach introduced by Arigovindan et al. [3] and Vuçini et al. [18]. Reconstruction:

- Instead of deriving coarser linear systems only to improve the convergence of the SOR solver for the reconstruction at a specific resolution, we use the solutions of these coarser linear systems as computationally very cheap and accurate coarser approximations for LoDs. We have shown that derived linear systems are virtually equivalent to dedicatedly constructed ones, but that the required run-time is way smaller and only dependent upon the resolution. The resulting coarser solutions are way more accurate than the ones obtained by coarsening the finest solution (used by Vuçini et al. [18]; coarsening the solution even introduces a small additional computational overhead). For this scheme to work out properly, details such as boundary conditions, pyramid alignment and the relation between λ and resolution (leading to our normalized λ_p^{n}) need to be carefully analyzed.

- A multi-scale reconstruction via a B-spline pyramid significantly increases the convergence rate of the iterative SOR solver, especially for high resolutions. We introduced the λ_p^{n} interscale factor f_λ to obtain smoother results for coarser scales. We have also seen that the analytical solution is sometimes not desirable if the given point cloud lacks important point samples, e.g., for the Cooling Jacket data set or sparse Laplacian data sets. For these cases, one can exploit the nature of the SOR solver by initializing the signal with zeros and terminating the refinement early.

- Our optimized implementation of the volume reconstruction exploits parallelizability to take full advantage of current CPU architectures. The presented reconstruction timings show what nowadays regular desktop PCs are capable of. Note that for many cases, we could have relaxed the convergence threshold for similar results (for finer scales of multi-scale pyramids), leading to smaller run-times.

- While Arigovindan et al. [3] present 2D results based on linear and cubic B-splines, Vuçini et al. [18] as well as Morozov et al. [14] only focus on cubic B-splines for their 3D data sets. We evaluated both linear and cubic B-splines and allow for a direct comparison between the two.

- We gave explicit formulas for all entities required for the linear system as well as hints on how to efficiently construct and solve the linear system.

Subdivision:

- Our subdivision scheme is primarily motivated by the aim to allow for good ray-casting performance. The key to this is to enable efficient sampling of the volume's B-spline. The approach presented by Vuçini et al. [18] does not meet this requirement. Firstly, their neighboring blocks overlap as each block side is extended by two intervals. So for instance, sampling the volume near an inner block corner requires evaluating and summing up the B-splines of 8 blocks. Secondly, they aim at reducing memory requirements by storing a complete coarse coefficients grid and 20% of each finer scale's error volume (run-length encoded). A single block's signal is therefore a sum of multiple B-splines as well, and additionally, the coefficients of the error volumes need to be decoded.

- We therefore opted for autonomous blocks, meaning that the inner B-spline signal of a block represents the global B-spline inside this block. Sampling the volume is hence reduced to evaluating the B-spline of the corresponding block. We also wanted to be able to use different resolutions for the blocks. To ensure global C^{n-1} continuity, we presented our simple fusing scheme, which imposes two restrictions: the resolutions may only differ by powers of two, and the block boundaries need to be aligned at shared knots. These restrictions lead to our proposed bottom-up subdivision, using a Cartesian lattice of arbitrarily sized cubes as building block of our blocks and powers of two as resolution levels for the cubes.

- We emphasized the need for some temporary block padding during reconstruction so as to increase the accuracy of block borders and to minimize visual block discontinuities.

- We have shown how to derive different volume LoDs by selecting an appropriate scale from each cube's multi-scale pyramid, based on a simple heuristic aimed at minimizing the RMS error.

Rendering:

- Vuçini et al. [18] use a GPU ray-caster for volumes consisting of a single resolution level only. To be able to render blocks at different resolutions, we need a data structure as spatial blocks index which is suited for ray-casting. We employ a binary BIH tree and have shown how to construct and traverse the tree.

- As work-around for current GPU limitations, we presented a heuristic to fit blocks of different size and resolutions into a single 3D texture.
- We analyzed the GPU ray-casting performance based on an optimized implementation making use of Sigg and Hadwiger's acceleration scheme [16] and adapting the sampling step length for iso-surface renderings of mixed-resolution LoDs.

12.3 Potential Improvements

- The proposed LoD scheme offers room for improvements. Firstly, our criterion for the scale selection, the minimization of the total RMS error, may be replaced by a more elaborate analysis of the local signal frequencies. Secondly, our minimalistic interscale fusing scheme may be extended to minimize visual interscale discontinuities more effectively. Thirdly, the BIH tree may be optimized by selecting more suited splitting planes instead of only pruning the tree when merging cubes to blocks.
- Data sets may be preprocessed in case they are not ideally suited for our approach – such as data sets where empty regions are indicated by the absence of point samples instead of including important null-samples to clearly indicate the object boundaries (the Cooling Jacket data set, for example). For these data sets, artificial null-samples could be inserted into empty regions.
- The SOR solver could be replaced by another iterative solver, e.g., one based on the conjugate-gradient method which is inherently parallelizable. This would allow to exploit the GPU for solving the linear systems, although currently GPUs are not equipped with as much memory as mainboards.

APPENDIX A

Proofs

A.1 Natural Boundary Conditions for Expanding and Coarsening

In chapter 3, we stated that the 1D expansion and coarsening matrices need to be related by $\mathbf{D}_j = s \cdot \mathbf{U}_{j+1}^T$ for an arbitrary scale factor $s \neq 0$, otherwise $\mathbf{A}_{j+1}$ would generally not be symmetric anymore. The element in row k and column l of the derived coarser matrix is explicitly defined by

$$\{\mathbf{A}_{j+1}\}_{k,l} = \sum_{i_1} \sum_{i_2} \{\mathbf{D}_j\}_{k,i_1} \{\mathbf{A}_j\}_{i_1,i_2} \{\mathbf{U}_{j+1}\}_{i_2,l}. \tag{A.1}$$

The symmetric element is defined by

$$\begin{aligned}\{\mathbf{A}_{j+1}\}_{l,k} &= \sum_{i_1} \sum_{i_2} \{\mathbf{D}_j\}_{l,i_1} \{\mathbf{A}_j\}_{i_1,i_2} \{\mathbf{U}_{j+1}\}_{i_2,k} \\ &= \sum_{i_2} \sum_{i_1} \{\mathbf{U}_{j+1}\}_{i_1,k} \{\mathbf{A}_j\}_{i_2,i_1} \{\mathbf{D}_j\}_{l,i_2}.\end{aligned} \tag{A.2}$$

For symmetricity, we set $\{\mathbf{A}_{j+1}\}_{k,l} = \{\mathbf{A}_{j+1}\}_{l,k}$ and yield

$$\sum_{i_1} \sum_{i_2} \{\mathbf{D}_j\}_{k,i_1} \{\mathbf{A}_j\}_{i_1,i_2} \{\mathbf{U}_{j+1}\}_{i_2,l} = \sum_{i_1} \sum_{i_2} \{\mathbf{U}_{j+1}\}_{i_1,k} \{\mathbf{A}_j\}_{i_2,i_1} \{\mathbf{D}_j\}_{l,i_2}. \tag{A.3}$$

The finer matrix $\mathbf{A}_j$ is symmetric, hence $\{\mathbf{A}_j\}_{i_1,i_2} = \{\mathbf{A}_j\}_{i_2,i_1} \ \forall\, i_1, i_2$. So for general symmetricity independent from symmetric $\mathbf{A}_j$, we finally obtain

$$\{\mathbf{D}_j\}_{k,i_1} \{\mathbf{U}_{j+1}\}_{i_2,l} = \{\mathbf{U}_{j+1}\}_{i_1,k} \{\mathbf{D}_j\}_{l,i_2} \quad \forall\, i_1, i_2. \tag{A.4}$$

This constraint is fulfilled by $\mathbf{D}_j = s \cdot \mathbf{U}_{j+1}^T$:

$$\begin{aligned} s\{\mathbf{U}_{j+1}^T\}_{k,i_1}\ \{\mathbf{U}_{j+1}\}_{i_2,l} &= \{\mathbf{U}_{j+1}\}_{i_1,k}\ s\{\mathbf{U}_{j+1}^T\}_{l,i_2} \\ s\{\mathbf{U}_{j+1}\}_{i_1,k}\ \{\mathbf{U}_{j+1}\}_{i_2,l} &= \{\mathbf{U}_{j+1}\}_{i_1,k}\ s\{\mathbf{U}_{j+1}\}_{i_2,l} \end{aligned} \tag{A.5}$$

which holds for arbitrary (k, l).

The matrices for a fine inner signal length of four intervals in scale j and a coarse one of two intervals in scale $(j+1)$ for the cubic 1D examples in Figures 2.3 and 2.4 are given by

$$\mathbf{U}_{j+1} = \frac{1}{8} \cdot \begin{pmatrix} 4 & 4 & & & \\ 1 & 6 & 1 & & \\ & 4 & 4 & & \\ & 1 & 6 & 1 & \\ & & 4 & 4 & \\ & & 1 & 6 & 1 \\ & & & 4 & 4 \end{pmatrix}$$

$$\mathbf{D}_j = \frac{1}{16} \cdot \begin{pmatrix} 4 & 1 & & & & & \\ 4 & 6 & 4 & 1 & & & \\ & 1 & 4 & 6 & 4 & 1 & \\ & & & 1 & 4 & 6 & 4 \\ & & & & & 1 & 4 \end{pmatrix} = \frac{1}{2} \cdot \mathbf{U}_{j+1}^T.$$

If we mirrored the coefficients at the inner signal boundaries to obtain a periodic signal (see Figure 2.5) and therefore excluded both outer coefficients, the matrices would be given by

$$\mathbf{U}_{j+1} = \frac{1}{8} \cdot \begin{pmatrix} 6 & 1+1 & \\ 4 & 4 & \\ 1 & 6 & 1 \\ & 4 & 4 \\ & 1+1 & 6 \end{pmatrix}$$

$$\mathbf{D}_j = \frac{1}{16} \cdot \begin{pmatrix} 6 & 4+4 & 1+1 & & \\ 1 & 4 & 6 & 4 & 1 \\ & & 1+1 & 4+4 & 6 \end{pmatrix}$$

and thus $\mathbf{D}_j$ would not be a scaled transpose of $\mathbf{U}_{j+1}$ anymore. It can be easily shown that the coarser matrix $\mathbf{A}_{j+1}$ would generally not be symmetric anymore, unlike the matrix obtained by constructing a dedicated linear system in the same scale $(j+1)$.

In practice, this means that periodic signals are to be avoided and that simple natural boundary conditions are to be used, i.e., missing elements in the convolution window are to be treated as zeros when coarsening matrices and when expanding/coarsening vectors.

Bibliography

[1] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point Set Surfaces. In *Proceedings of IEEE Visualization*, pages 21–28, 2001.

[2] N. Amenta and M. Bern. Surface Reconstruction by Voronoi Filtering. *Discrete and Computational Geometry*, 22:481–504, 1998.

[3] M. Arigovindan, M. Sühling, P. Hunziker, and M. Unser. Variational Image Reconstruction from Arbitrarily Spaced Samples: A Fast Multiresolution Spline Solution. *IEEE Transactions on Image Processing*, 14(4):450–460, 2005.

[4] J. Beyer, M. Hadwiger, T. Möller, and L. Fritz. Smooth Mixed-Resolution GPU Volume Rendering. In *Proceedings of IEEE International Symposium on Volume and Point-Based Graphics*, pages 163–170, 2008.

[5] P. Brigger, F. Müller, K. Illgner, and M. Unser. Centered Pyramids. *IEEE Transactions on Image Processing*, 8(9):1254–1264, 1999.

[6] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11:285–295, 2005.

[7] J. Duchon. Splines minimizing rotation-invariant semi-norms in Sobolev spaces. In *Constructive Theory of Functions of Several Variables*, pages 85–100, 1977.

[8] A. Entezari, D. Van De Ville, and T. Möller. Practical Box Splines for Reconstruction on the Body Centered Cubic Lattice. *IEEE Transactions on Visualization and Computer Graphics*, pages 313–328, 2008.

[9] C. P. Gribble, A. J. Stephens, J. E. Guilkey, and S. G. Parker. Visualizig Particle-Based Simulation Datasets on the Desktop. In *Workshop on Combining Visualization and Interaction to Facilitate Scientific Exploration and Discovery*, pages 111–118, 2006.

[10] Y. Jang, R. P. Botchen, A. Lauser, D. S. Ebert, K. P. Gaither, and T. Ertl. Enhancing the Interactive Visualization of Procedurally Encoded Multifield Data with Ellipsoidal Basis Functions. *Computer Graphics Forum*, 25(3):587–596, 2006.

[11] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl. Interactively Visualizing Procedurally Encoded Scalar Fields. In *Proceedings of Joint Eurographics-IEEE TCVG Symposium on Visualization*, pages 35–44, 2004.

[12] J. Kybic, T. Blu, and M. Unser. Generalized Sampling: A Variational Approach—Part I: Theory. *IEEE Transactions on Signal Processing*, 50(8):1965–1976, 2002.

[13] J. Kybic, T. Blu, and M. Unser. Generalized Sampling: A Variational Approach—Part II: Applications. *IEEE Transactions on Signal Processing*, 50(8):1977–1985, 2002.

[14] O. Morozov, M. Unser, and P. Hunziker. Reconstruction of Large, Irregularly Sampled Multidimensional Images. A Tensor-Based Approach. *IEEE Transactions on Medical Imaging*, 30(2):366–374, 2011.

[15] S. W. Park, L. Linsen, O. Kreylos, J. D. Owens, and B. Hamann. Discrete Sibson Interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):243–253, 2006.

[16] C. Sigg and M. Hadwiger. Fast Third-Order Texture Filtering. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 313–329, 2005.

[17] E. Vuçini. *On Visualization and Reconstruction from Non-uniform Point Sets*. PhD thesis, Vienna University of Technology, Austria, 2009.

[18] E. Vuçini, T. Möller, and M. E. Gröller. On Visualization and Reconstruction from Non-Uniform Point Sets using B-splines. *Computer Graphics Forum*, 28(3):1007–1014, 2009.

[19] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006.

[20] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization*, pages 333–340, 2003.