High-Quality Rendering of Varying Isosurfaces with Cubic Trivariate C^1-continuous Splines

Thomas Kalbe, Thomas Koch, and Michael Goesele

GRIS, TU Darmstadt

Abstract. Smooth trivariate splines on uniform tetrahedral partitions are well suited for high-quality visualization of isosurfaces from scalar volumetric data. We propose a novel rendering approach based on spline patches with low total degree, for which ray-isosurface intersections are computed using efficient root finding algorithms. Smoothly varying surface normals are directly extracted from the underlying spline representation. Our approach is using a combined CUDA and graphics pipeline and yields two key advantages over previous work. First, we can interactively vary the isovalues since all required processing steps are performed on the GPU. Second, we employ instancing in order to reduce shader complexity and to minimize overall memory usage. In particular, this allows to compute the spline coefficients on-the-fly in real-time on the GPU.

1 Introduction

The visualization of discrete data on volumetric grids is a common task in various applications, e.g., medical imaging, scientific visualization, or reverse engineering. The construction of adequate non-discrete models which fit our needs in terms of visual quality as well as computational costs for the display and preprocessing is an interesting challenge. The most common approach is trilinear interpolation [1, 2], where the tensor-product extension of univariate linear splines interpolating at the grid points results in piecewise cubic polynomials. A sufficiently smooth function is approximated with order two, but in general, reconstructions are not smooth and visual artifacts, like stair-casing or imperfect silhouettes, arise. However, the simplicity of this model has motivated its widespread use. Tri-quadratic or tri-cubic tensor-product splines can be used to construct smooth models. These splines lead to piecewise polynomials of higher total degree, namely six and nine, which are thus more expensive to evaluate.

In order to alleviate these problems, we use cubic trivariate C^1-splines [3] for interactive visualizations of isosurfaces from volumetric data with ray casting. The low total degree of the spline pieces allows for efficient and stable ray-patch intersection tests. The resulting spline pieces are directly available in Bernstein-Bézier–form (BB-form) from the volumetric data by simple, efficient and local averaging formulae using a symmetric and isotropic data stencil from the 27-neighborhood of the nearest data value. The BB-form of the spline pieces has several advantages: well-known techniques from CAGD, like de Casteljau's algorithm and *blossoming*, can be employed for efficient and stable evaluation of the splines. The derivatives needed for direct illumination are immediately available as a by-product. The convex hull property of the BB-form allows to quickly decide if a given spline patch contributes to the final surface.

A first GPU implementation for visualization with cubic trivariate splines has been given by [4]. Data streams of a fixed isolevel were prepared on the CPU and then sent to the GPU for visualization. This preprocess, which takes a couple of seconds for medium-sized data sets ($\geq 256^3$ data values), has to be repeated for each change of isolevel. However, in many applications it is essential to vary the isosurface interactively in order to gain deeper understanding of the data (see Fig. 1). In this work, we significantly accelerate the preprocess using NVIDIA's *CUDA* framework and achieve reconstruction times which are even below the rendering times of a single frame. In addition, current innovations of the graphics pipeline in Shader Model 4.0, like *instancing*, allow us to compute all necessary spline coefficients on-the-fly directly in the vertex shader. Therefore, we do not need to inflate the data prior to visualization, but merely store the volume data as a texture on the GPU. In addition, geometry encoding is simplified and memory overhead is reduced. Combining these contributions, we significantly improved the usability of high-quality trivariate splines in real-world applications.

2 Related work

Techniques for visualizations of gridded scalar data can be categorized into two general classes. *Full volume rendering*, where the equations of physical light transport are integrated throughout the volume, commonly along viewing rays, and the somewhat less complex *isosurfacing*. In the latter case, we are interested in the zero contour of a continuous implicit function which approximates or interpolates the discrete values given at the grid points. We can classify isosurfacing further into methods that obtain discrete representations of the surfaces, e.g., triangle meshes. A standard approach in this area is *marching cubes* [5]. Alternatively, we are only interested in visualizations of the isosurfaces, which is often done by ray casting, where the first intersection of each viewing ray with the surface is determined for later illumination. The recent development of graphics processors has been a massive impulse for interactive volume graphics on consumer hardware (see [6] for a survey). Interactive techniques exist for full volume rendering, isosurface visualization and reconstruction, e.g, [7–10]. Still, most of these approaches are based on tri-linear interpolation and therefore trade visual quality in favor of rendering speed. Gradients can be pre-computed at the grid points, at the cost of increased memory and bandwidth consumption. Alternatively, the gradients are computed on-the-fly using central differences, which is an expensive operation. Either way, the obtained gradients are not smooth, and visual artifacts arise. To circumvent these problems, higher-order filter kernels, e.g., smooth tri-cubic or tri-quartic box splines, have been proposed [2, 11]. One of the few successful implementations of interactive isosurface visualization with higher order filtering has been given by [12]. These splines lead to polynomials of total degree nine, for which no exact root finding algorithms exist. Furthermore,

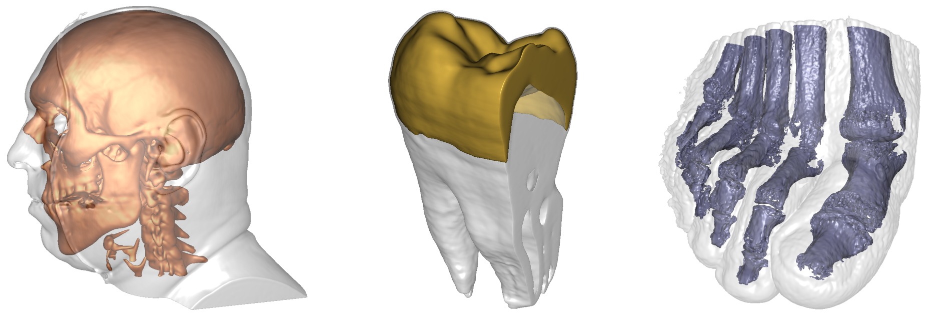

Fig. 1. Blending of different isosurfaces from real-world data sets. *From left to right*: *VisMale* (256^3 voxels), *Tooth* ($256^2 \times 161$ voxels), and *Foot* (256^3 voxels). VisMale and Tooth are smoothed with a Gaussian filter on the GPU. Choosing the desired isosurfaces and smoothing to an appropriate degree is an interactive process.

data stencils are large (usually the 64-neighborhood), and important features might be lost resulting from the large support of the filter kernels.

We use smooth trivariate splines defined w.r.t. uniform tetrahedral partitions. Here, the filter kernels are small and isotropic. Since the total degree of the polynomial pieces does not exceed three, we can choose suitable starting values for an iterative root finding algorithm, such that precise intersections with the isosurface are obtained in a stable and efficient way. No further refinements, e.g., near the surface's silhouette, are needed. For an example see Fig. 7, right. An approach for interactive visualization using trivariate splines, has been given by [4]. While this work was the first to allow for real-time rendering of up to millions of smoothly connected spline patches simultaneously, it is based on the common principles described by, e.g, [13–15]. These methods project the bounding geometry of the polynomials in screen space and perform intersection tests and illumination during fragment processing. [4] rely on a CPU preprocessing of the data for each change of isovalue, which can be done only off-line. Furthermore, memory requirements for the storage of spline coefficients are substantial. In this paper, we shift the preprocessing to the GPU in order to allow for an interactive change of isosurface. To do that, we use *parallel prefix scans* as described in [16, 17]. In addition, we show how to reduce memory overhead to a minimum using current innovations in the graphics pipeline.

3 Trivariate splines and the BB-form

In this section, we give a brief outline of the basic terminology and mathematical background of trivariate splines in BB-form on tetrahedral partitions, along with a description of the calculation of the spline coefficients. These coefficients can be directly obtained from the volume data by simple averaging formulae.

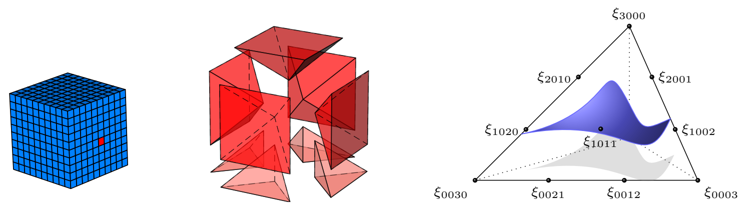

Fig. 2. *Left*: the cube partition $\diamondsuit$ of the domain Ω. *Middle*: the type-6 partition is obtained by first subdividing each cube into six pyramids, which are then further split into four congruent tetrahedra each. *Right*: the zero contour of a single cubic trivariate spline patch $s|_T$ within it's bounding tetrahedron T. The domain points $\xi_{ijk\ell}$ associated with the BB-coefficients $b_{ijk\ell}$ of the front-most triangle are shown.

3.1 Preliminaries and basic notation

For $n \in \mathbb{N}$ let $\mathcal{V} := \{\mathbf{v}_{ijk} = (ih, jh, kh) : i, j, k = 0, \ldots, n\}$ be the cubic grid of $(n+1)^3$ points with grid size $h = 1/n \in \mathcal{R}$. We define a cube partition $\diamondsuit = \{Q : Q = Q_{ijk}\}$ of the domain Ω, where each $Q_{ijk} \in \diamondsuit$ is centered at $\mathbf{v}_{ijk}$ and the vertices of Q_{ijk} are $(2i \pm 1, 2j \pm 1, 2k \pm 1)^t \cdot h/2$, see Fig. 2, left.

We consider trivariate splines on the *type-6 tetrahedral partition* Δ^6, where each Q is subdivided into 24 congruent tetrahedra. This is done by connecting the vertices of Q_{ijk} with the center $\mathbf{v}_{ijk}$. Each of the resulting six pyramids is then further split into four tetrahedra, see Fig. 2, right. The space of cubic trivariate C^1 splines on Δ^6 is defined by $\mathcal{S}_3^1(\Delta^6) = \{s \in C^1(\Omega) : s|_T \in \mathcal{P}_3, \text{ for all } T \in \Delta^6\}$, where $C^1(\Omega)$ is the set of continuously differentiable functions on Ω, $\mathcal{P}_3 := \operatorname{span}\{x^\nu y^\mu z^\kappa : 0 \leq \nu + \mu + \kappa \leq 3\}$ is the 20-dimensional space of trivariate polynomials of total degree three, and T is a tetrahedron in Δ^6. We use the BB-form of the polynomial pieces, i.e.

$$s|_T = \sum_{i+j+k+\ell} b_{ijk\ell} B_{ijk\ell}, \; i + j + k + \ell = 3,$$

where the $B_{ijk\ell} = \frac{3!}{i!j!k!\ell!} \phi_0^i \phi_1^j \phi_2^k \phi_3^\ell \in \mathcal{P}_3$ are the cubic *Bernstein polynomials* w.r.t a tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \in \Delta^6$. For each T, we set $\mathbf{v}_0$ to the center of it's cube Q, $\mathbf{v}_1$ to the centroid of one of the faces of Q and $\mathbf{v}_2, \mathbf{v}_3$ to the vertices of Q sharing a common edge. The *barycentric coordinates* $\boldsymbol{\phi}(\mathbf{x}) = (\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \phi_3(\mathbf{x}))^t$ of a point $\mathbf{x} = (x, y, z, 1)^t$ w.r.t. a non-degenerate T are the linear trivariate polynomials determined by $\phi_\nu(\mathbf{v}_\mu) = \delta_{\nu,\mu}$, $\nu, \mu = 0, \ldots, 3$, where $\delta_{\nu,\mu}$ is Kronecker's symbol. They are given by the linear system of equations

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \\ 1 & 1 & 1 & 1 \end{pmatrix}^{-1} \cdot \mathbf{x}. \tag{1}$$

The *BB-coefficients* $b_{ijk\ell} \in \mathcal{R}$ are associated with the 20 domain points $\xi_{ijk\ell} = (i\mathbf{v}_0 + j\mathbf{v}_1 + k\mathbf{v}_2 + \ell\mathbf{v}_3)/3$, see Fig. 2, right, and we let $\mathcal{D}(\Delta^6)$ be the union of the sets of domain points associated with the tetrahedra of Δ^6. As

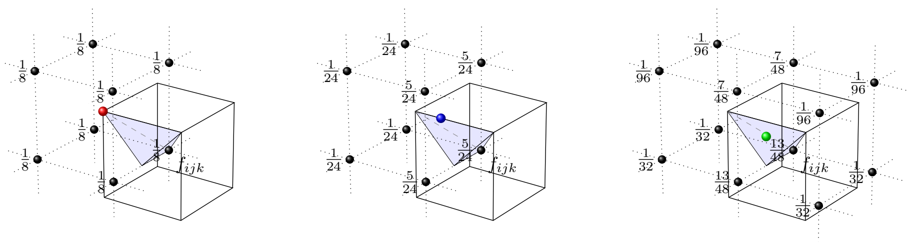

Fig. 3. The masks for the coefficients associated with the domain points ξ_{0003} (left), ξ_{0021} (middle) and ξ_{0111} (right) for the shaded tetrahedron in Q_{ijk}. The remaining coefficients of $\Gamma_{Q_{ijk}}$ follow from symmetry and rotations. Black dots denote data values.

pointed out by e.g. [18], the BB-form is especially useful for defining smoothness conditions between neighboring polynomial pieces. Let $T, \tilde{T}$ be two neighboring tetrahedra sharing a common face $F = T \cap \tilde{T} = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2]$, then a cubic spline s on $T \cup \tilde{T}$ is *continuous* ($s \in \mathcal{S}_3^0$) if $s|_T(\mathbf{x}) = s|_{\tilde{T}}(\mathbf{x}),\ \mathbf{x} \in F$. Using the BB-form, we have $s \in \mathcal{S}_3^0$ if $b_{ijk0} = \tilde{b}_{ijk0}$ and a continuous spline s is uniquely defined by the coefficients $\{b_\xi : \xi \in \mathcal{D}(\Delta^6)\}$. Furthermore, s is C^1-continuous across F iff

$$\tilde{b}_{ijk1} = b_{i+1,j,k,0}\, \phi_0(\tilde{\mathbf{v}}_3) + b_{i,j+1,k,0}\, \phi_1(\tilde{\mathbf{v}}_3) + b_{i,j,k+1,0}\, \phi_2(\tilde{\mathbf{v}}_3) + b_{i,j,k,1}\, \phi_3(\tilde{\mathbf{v}}_3), \qquad (2)$$

where $i + j + k = 2$ and $\tilde{\mathbf{v}}_3$ is the vertex of $\tilde{T}$ opposite to F. Smoothness of the splines on Δ^6 is thus easily described when considering only two neighboring polynomial pieces. The complexity of the spline spaces arises from the fact that smoothness conditions have to be fulfilled not only between two neighboring patches, but across all the interior faces of Δ^6 simultaneously.

We can evaluate a spline patch $s|_T(\mathbf{x})$ with de Casteljau's algorithm. Set $b^{[0]}_{ijk\ell} = b_{ijk\ell}$, a de Casteljau step computes $b^{[\eta]}_{ijk\ell}$, $i+j+k+\ell = 3-\eta$, as the inner product of $(\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \phi_3(\mathbf{x}))^t$ and $(b^{[\eta-1]}_{i+1,j,k,\ell}, b^{[\eta-1]}_{i,j+1,k,\ell}, b^{[\eta-1]}_{i,j,k+1,\ell}, b^{[\eta-1]}_{i,j,k,\ell+1})^t$ with $s|_T(\mathbf{x}) = b^{[3]}_{0000}$. In addition, the $(3-1)$th step provides the four independent directional derivatives

$$\frac{\partial s|_T(\mathbf{x})}{\partial \phi_\nu(\mathbf{x})} = b^{[2]}_{\boldsymbol{\nu}},\ \ \nu = 0, \ldots, 3, \qquad (3)$$

where $\boldsymbol{\nu} \in \mathbb{N}_0^4$ is the vector with a 1 at position ν and 0 everywhere else. With $s|_T = p$ and using the chain rule, we have

$$\frac{\partial p(\mathbf{x})}{\partial x_\iota} = \sum_\nu \frac{\partial s|_T(\mathbf{x})}{\partial \phi_\nu(\mathbf{x})} \cdot \frac{\partial \phi_\nu(\mathbf{x})}{\partial x_\iota},\ \ \iota \in 1, 2, 3, \qquad (4)$$

with the gradient $\nabla s|_T(\mathbf{x}) = (\partial p/\partial x_1, \partial p/\partial x_2, \partial p/\partial x_3)^t$. Since each ϕ_ν is a linear polynomial, the $\partial \phi_\nu / \partial x_\iota$ are scalar constants characterized by the barycentric coordinates of the ιth Cartesian unit vector $\mathbf{e}_\iota$ w.r.t. T.

Trivariate *blossoming* [19] is a generalization of de Casteljau's algorithm, where the arguments may vary on the different levels. For any $\mathbf{x}_\eta$, $\eta \in 1, 2, 3$

with $\boldsymbol{\phi}_\eta = \boldsymbol{\phi}(\mathbf{x}_\eta)$, we denote the blossom of $s|_T$ as $\mathrm{bl}[\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \boldsymbol{\phi}_3]$, meaning that the first step of de Casteljau's algorithm is carried out with $\boldsymbol{\phi}_1$, the second step with $\boldsymbol{\phi}_2$ and the third with $\boldsymbol{\phi}_3$. It is easy to see that $\mathrm{bl}[\boldsymbol{\phi}_\eta, \boldsymbol{\phi}_\eta, \boldsymbol{\phi}_\eta] = s|_T(\mathbf{x}_\eta)$. In addition, the blossom is *multi-affine*

$$\mathrm{bl}[\ldots, \alpha \cdot \boldsymbol{\phi}_\eta + (1-\alpha) \cdot \bar{\boldsymbol{\phi}}_\eta, \ldots] = \alpha \cdot \mathrm{bl}[\ldots, \boldsymbol{\phi}_\eta, \ldots] + (1-\alpha) \cdot \mathrm{bl}[\ldots, \bar{\boldsymbol{\phi}}_\eta, \ldots], \qquad (5)$$

for $\alpha \in \mathbb{R}$, and symmetric, i.e., for a permutation $\sigma = (\sigma(1), \sigma(2), \sigma(3))$ we have

$$\mathrm{bl}[\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \boldsymbol{\phi}_3] = \mathrm{bl}[\boldsymbol{\phi}_{\sigma(1)}, \boldsymbol{\phi}_{\sigma(2)}, \boldsymbol{\phi}_{\sigma(3)}].$$

These properties of the blossom enable us to find intersections of rays with a spline patch in an efficient way, see Sect. 4.4.

3.2 The approximating scheme

For smooth approximations of the given data values $f(\mathbf{v}_{ijk})$, associated with the grid points $\mathbf{v}_{ijk}$, we use quasi-interpolating cubic splines as described in [3], which approximate sufficiently smooth functions with order two. The BB-coefficients b_ξ for each tetrahedron are directly available from appropriate weightings of the data values in a symmetric 27-neighborhood of the centering data value $f(\mathbf{v}_{ijk})$,

$$b_\xi = \sum_{i_0,j_0,k_0} \omega_{i_0 j_0 k_0} f(\mathbf{v}_{i+i_0, j+j_0, k+k_0}),\ i_0, j_0, k_0 \in \{-1, 0, 1\},$$

where the $\omega_{i_0 j_0 k_0} \in \mathcal{R}$ are constant and positive weights. A *determining set* $\Gamma \subseteq \mathcal{D}(\Delta^6)$ is a subset of the domain points with associated BB-coefficients, from which the remaining coefficients for each $s|_T$ can be uniquely identified from the smoothness conditions. We use a symmetric determining set Γ_Q for each $Q \in \Diamond$, formed by the coefficients associated with the domain points $\xi_{00k\ell} \bigcup \xi_{0111}$, where $k + \ell = 3$. For a tetrahedron, ξ_{0030} and ξ_{0003} are vertices of Q, ξ_{0021} and ξ_{0012} are on the outer edges of Q, and ξ_{0111} corresponds to the centroid of the face $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$. We show the weights for the coefficients of the determining set in Fig. 3. The weights for the remaining coefficients follow from the smoothness conditions (see Eq. 2) and can also be found in [3, 4].

4 Trivariate splines – GPU visualization

In this section, we give an overview of our GPU-algorithm for efficient visualization of varying isosurfaces, see also Fig. 4, left. In the first part, we use a set of CUDA kernels, which are invoked for each change of isolevel or data set. The kernels determine all the tetrahedra which can contribute to the final surface and prepare the appropriate data structures. The second part uses vertex and fragment programs for the visualization of the surface in a combined rasterization / ray casting approach. For a 2-dimensional example of the visualization principle see Fig. 4, right top. For each *active* tetrahedron, i.e., tetrahedra contributing to the surface, the bounding geometry is processed in the OpenGL pipeline. The vertex programs initialize various parameters, such as the viewing rays, the BB-coefficients and appropriate barycentric coordinates. The fragment programs then perform the actual ray-patch intersection tests.

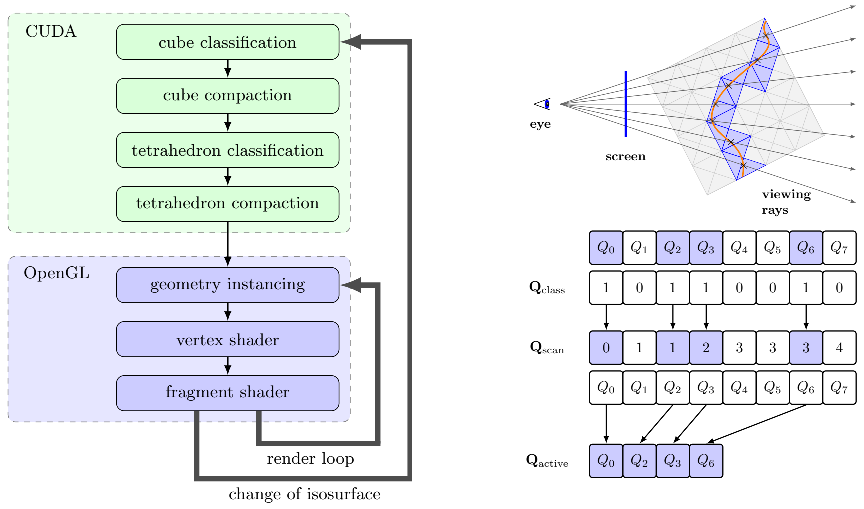
CUDA
cube classification
cube compaction
tetrahedron classification
tetrahedron compaction
OpenGL
geometry instancing
vertex shader
fragment shader
render loop
change of isosurface
eye
screen
viewing rays
Q_0 Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7
$\mathbf{Q}_{\text{class}}$ 1 0 1 1 0 0 1 0
$\mathbf{Q}_{\text{scan}}$ 0 1 1 2 3 3 3 4
Q_0 Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7
$\mathbf{Q}_{\text{active}}$ Q_0 Q_2 Q_3 Q_6

Fig. 4. *Left*: Overview of our our GPU algorithm. The CUDA part specifies the relevant cubes and tetrahedra, respectively, when the isovalue or data set is modified. The visualization is using the rendering pipeline. *Right top*: 2-dimensional illustration of the ray casting principle. On each triangle satisfying the convex hull property (dark shaded), the corresponding BB-curve is intersected with the viewing rays. *Right bottom*: parallel stream compaction with prefix scans. For each entry with a 1 in $\mathbf{Q}_{\text{class}}$, $\mathbf{Q}_{\text{scan}}$ contains an unique address into the compressed array $\mathbf{Q}_{\text{active}}$. The sum of the last entries of $\mathbf{Q}_{\text{class}}$ and $\mathbf{Q}_{\text{scan}}$ gives the size of $\mathbf{Q}_{\text{active}}$. Illustration based on [17].

4.1 The CUDA kernels

For each $Q \in \lozenge$, we first start a kernel thread that computes the coefficients for the determining set $\mathit{\Gamma}_Q$, from which the remaining coefficients can be found quickly using simple averaging. The *cube classification* tests if the coefficients of $\mathit{\Gamma}_Q$ are either below or above the isolevel. In this case, it follows from the convex hull property of the BB-form that the patches in Q cannot contribute to the surface and we can exclude Q from further examination. Otherwise, Q contains at least one tetrahedron with a visible patch. The result of the classification is written in the corresponding entry of a linear integer array $\mathbf{Q}_{\text{class}}$ of size $(n+1)^3$: we write a 1 in $\mathbf{Q}_{\text{class}}$ if Q passes the classification test and a 0 otherwise. From this unsorted array $\mathbf{Q}_{\text{class}}$, we construct a second array $\mathbf{Q}_{\text{scan}}$ of the same size using the parallel prefix scan from the *CUDA data parallel primitives* (CUDPP) library. For each *active* cube, i.e., cubes with a 1 in $\mathbf{Q}_{\text{class}}$, $\mathbf{Q}_{\text{scan}}$ then contains an unique index corresponding to the memory position in a compacted array $\mathbf{Q}_{\text{active}}$. Since we use an exclusive prefix scan, the sum of the last entries of $\mathbf{Q}_{\text{class}}$ and $\mathbf{Q}_{\text{scan}}$ gives the number of active cubes a for the surface. In the *compaction* step, we reserve memory for the array $\mathbf{Q}_{\text{active}}$ of size a. For each active Q_{ijk}, we write the index $i + j \cdot (n+1) + k \cdot (n+1)^2$ in $\mathbf{Q}_{\text{active}}$ at the position given by the corresponding entry of $\mathbf{Q}_{\text{scan}}$, see Fig. 4, right bottom.

Similarly, we perform a compaction for the active tetrahedra. Since we use instancing for later rendering of the tetrahedra, we reserve 24 arrays $\mathbf{T}_{\text{class},i}$, where each $\mathbf{T}_{\text{class},i}$ has size a and corresponds to one of the 24 different orientations of tetrahedra in Δ^6. For each active cube Q, a kernel thread performs the classification for $T_0, T_1, \ldots, T_{23} \in Q$, now using the convex hull property of the BB-form on the tetrahedra. Note that the BB-coefficients for Q have to be calculated only once and are then assigned to the corresponding domain points on the tetrahedra using a constant lookup table. The following stream compaction works exactly as described above, except that here we use 24 distinct arrays $\mathbf{T}_{\text{scan},i}$, and the compaction is performed by a set of 24 kernels (one for each type of tetrahedron), which write their results into the arrays $\mathbf{T}_{\text{active},i}$. These arrays are interpreted as pixel buffer objects, which can be directly used to render the bounding geometry of the spline patches.

4.2 Geometry instancing setup

The 24 arrays $\mathbf{T}_{\text{active},i}$ give us all the information needed to visualize the surface. In order to encode the necessary bounding geometries, i.e., the active tetrahedra, in the most efficient way, we construct a triangle strip for each *generic* tetrahedron of Δ^6 in the unit cube $[-0.5, 0.5]^3$. These 24 triangle strips are then stored as separate vertex buffer objects (VBOs). Each VBO is used to draw all the active tetrahedra of it's type with a call to `glDrawArraysInstanced`, where the number of tetrahedra is given by the size of $\mathbf{T}_{\text{active},i}$.

4.3 Vertex shader computations

We use 24 different shader sets, one for each of the different types of tetrahedra. Since each tetrahedron has it's dedicated vertex and fragment programs, we can avoid conditional branches. For every vertex $\mathbf{v}_\mu, \mu = 0, 1, 2, 3$, of a tetrahedron T, the vertex program first determines T's displacement $\mathbf{v}_Q$ from a texture reference into $\mathbf{T}_{\text{active},i}$. For later computation of the ray-patch intersection (see Sect. 4.4), we find the barycentric coordinates $\phi_{\nu,\mu}, \nu = 0, 1, 2, 3$ as $\phi_{\nu,\mu}(\mathbf{v}_\mu) = \delta_{\nu,\mu}$. In addition, a second set of barycentric coordinates $\bar{\phi}_{\nu,\mu}$, corresponding to the unit length extension of the vector defined by $\mathbf{v}_\mu + \mathbf{v}_Q$ and the eye point $\mathbf{e}$ is computed as $\bar{\phi}_{\nu,\mu}(\bar{\mathbf{v}}) = \bar{\phi}_{\nu,\mu}(\mathbf{v}_\mu + (\mathbf{v}_\mu - (\mathbf{e} - \mathbf{v}_Q))/||\mathbf{v}_\mu - (\mathbf{e} - \mathbf{v}_Q)||)$. To do this, we use Eq. 1, where the matrices are pre-computed once for each generic tetrahedron. The barycentric coordinates $\phi_{\nu,\mu}, \bar{\phi}_{\nu,\mu}$ are interpolated across T's triangular faces during rasterization. Finally, we have to determine the 20 BB-coefficients of $s|_T$. This can be done in the following way: first, we read the data values $f(\mathbf{v}_Q)$ and it's neighbors from the volume texture. For one patch, only 23 of the 27 values have to be fetched, corresponding to 23 texture accesses. Then, we can directly obtain the $b_{ijk\ell}$ according to Sect. 3.2. Alternatively, we can pre-compute the determining set for each Q in a CUDA kernel and store them as textures. Then, only the remaining coefficients for $s|_T$ need to be computed. This method is less memory efficient, but leads to a slightly improved rendering performance. For a comparative analysis, see Sect. 5.

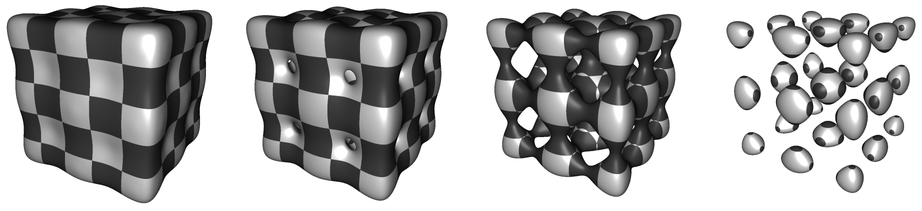

Fig. 5. Varying isosurfaces for a synthetic function of *Chmutov* type, $f(x, y, z) = x^{16} + y^{16} + z^{16} - \cos(7x) - \cos(7y) - \cos(7z)$, sampled from a sparse grid (64^3 data points) with real-time reconstruction and rendering times.

4.4 Fragment shader computations

For every fragment of a front-facing triangle, the fragment program performs the actual ray-patch intersection. To do this, we need an univariate representation of $s|_T$ restricted along the viewing ray. Using trivariate blossoming, we obtain an univariate cubic BB-curve which can be easily intersected. In addition, we can re-use intermediate results from the blossoms for quick gradient calculation and do not need to determine the exit point of the ray w.r.t. T. Using the interpolated barycentric coordinates $\boldsymbol{\phi} = (\phi_0, \phi_1, \phi_2, \phi_3)$ and $\bar{\boldsymbol{\phi}} = (\bar{\phi}_0, \bar{\phi}_1, \bar{\phi}_2, \bar{\phi}_3)$ obtained from rasterization, we can read off the univariate BB-coefficients directly from the blossoms, setting $b_{30} = \mathrm{bl}[\boldsymbol{\phi}, \boldsymbol{\phi}, \boldsymbol{\phi}], b_{21} = \mathrm{bl}[\boldsymbol{\phi}, \boldsymbol{\phi}, \bar{\boldsymbol{\phi}}], b_{12} = \mathrm{bl}[\bar{\boldsymbol{\phi}}, \bar{\boldsymbol{\phi}}, \boldsymbol{\phi}]$, and $b_{03} = \mathrm{bl}[\bar{\boldsymbol{\phi}}, \bar{\boldsymbol{\phi}}, \bar{\boldsymbol{\phi}}]$, see Sect. 3. Since intermediate results can be re-used, the first step of de Casteljau's algorithm, which accounts for ten inner products each, has to be performed for $\boldsymbol{\phi}$ and $\bar{\boldsymbol{\phi}}$ only once. We proceed in the same way for the second de Casteljau step with $b^{[1]}_{ijk\ell}(\boldsymbol{\phi})$ (resulting from the first step with $\boldsymbol{\phi}$) and $\boldsymbol{\phi}$, as well as $b^{[1]}_{ijk\ell}(\bar{\boldsymbol{\phi}})$ and $\bar{\boldsymbol{\phi}}$, where $i + j + k + \ell = 2$, using in total eight inner products. Finally, the blossoms are completed with four additional inner products using $b^{[2]}_{ijk\ell}(\boldsymbol{\phi})$ and $b^{[2]}_{ijk\ell}(\bar{\boldsymbol{\phi}})$, $i + j + k + \ell = 1$, which correspond to the remaining de Casteljau steps on the last level.

Next, the monomial form of the BB-curve, $\sum_{i=0}^{3} x_i \cdot t^i$, is solved for the ray parameter t. There exist several ways to find the zeros of a cubic equation. [14] choose an analytic approach, whereas [15] apply a recursive BB-hull subdivision algorithm. Since the first method involves trigonometric functions and the second does not converge very quickly, we opt for an iterative Newton approach. As starting values we choose $t_1^{(0)} = -x_0/x_1, t_1^{(1)} = (\frac{1}{4}(x_3 + x_2) - x_0)/(\frac{3}{4} \cdot x_3 + x_2 + x_1)$ and $t_1^{(2)} = (2 \cdot x_3 + x_2 - x_0)/(3 \cdot x_3 + 2 \cdot x_2 + x_1)$. Note that this corresponds to the first Newton iteration starting with $0, 1/2$, and 1, respectively. Four additional iterations with $t_{j+1}^{(\mu)} = ((t_j^{(\mu)})^2 (x_2 + 2 \cdot t_j^{(\mu)} x_3) - x_0)/(t_j^{(\mu)}(2 \cdot x_2 + 3 \cdot t_j^{(\mu)} x_3) + x_1)$, $\mu = 0, 1, 2$, suffice to find precise intersections without notable artifacts. For each solution $t \in t_5^{(\mu)}$, the associated barycentrics $\boldsymbol{\phi}(\mathbf{x}(t))$ are found by a simple linear interpolation with $\boldsymbol{\phi}$ and $\bar{\boldsymbol{\phi}}$. We take the first valid zero t, where all the components of $\boldsymbol{\phi}(\mathbf{x}(t))$ are positive, if it exists, and discard the fragment otherwise.

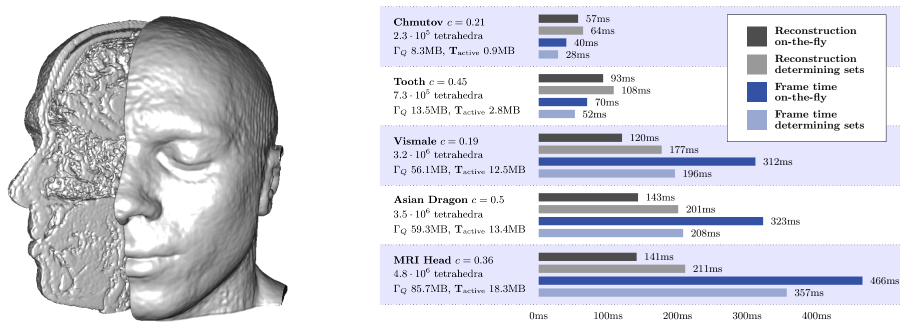

Fig. 6. *Left*: MRI scan ($192^2 \times 126$ voxels) volume clipped with the plane $x = 0$. *Right*: timings of the reconstruction process and the visualization (using precomputed determining sets as well as on-the-fly spline coefficient computations) for selected data sets and isovalues c. For each data set, the number of active tetrahedra, the size of the determining set Γ_Q and for the geometry encoding T_{active}, respectively, are given.

From the multi-affine property of the blossom (see Eq. 5), it follows that the directional derivatives $b^{[2]}_{\boldsymbol{\nu}}(\boldsymbol{\phi}(\mathbf{x}(t)))$ (see Eq. 3) are obtained by a linear interpolation of $b^{[1]}_{\boldsymbol{\nu}}(\boldsymbol{\phi})$ and $b^{[1]}_{\boldsymbol{\nu}}(\bar{\boldsymbol{\phi}})$, with the ray parameter t, followed by a de Casteljau step on the second level using $\boldsymbol{\phi}(\mathbf{x}(t))$ Finally, we calculate the gradient for later illumination according to Eq. 4 with three additional scalar products. Here, the $\partial\phi_\nu/\partial x_\iota$ are pre-computed for each of the 24 different generic tetrahedra.

5 Results and Discussion

We demonstrate our results with a series of data sets: *Tooth*, *VisMale* and *Foot* (see Fig. 1) are publicly available from the Universities of Tübingen and Erlangen, and the US Library of Medicine. Fig. 5 is an example of synthetic data obtained from a sparsely sampled smooth function. Fig. 6, left, shows a MRI scan of the head of one of the authors. Finally, the *Asian Dragon* (Fig. 7) is generated from a signed distance function on the original triangle mesh. All results demonstrate the high visual quality and smooth shading of our method.

Note that the low total degree of the spline patches allows us to obtain precise intersections, even for the objects' silhouettes, without resorting to interval refinements or similar approaches (see Fig. 7, right). Precise intersections are also needed for procedural texturing (see Fig. 5 and 7), and for volume clipping with arbitrary planes and surfaces (see Fig. 6, left). Furthermore, the obtained intersections are exact w.r.t. z-buffer resolution, which allows us to combine ray-casted isosurfaces with standard object representations, i.e., triangle meshes.

The table in Fig. 6, right, summarizes the performance for the chosen data sets and lists typical isovalues c, the number of active tetrahedra, as well as the size of the determining set Γ_Q and the geometry encoding $\mathbf{T}_{\text{active}}$. Timings were recorded on a GeForce GTX 285 and CUDA 2.2. For each data set, the first

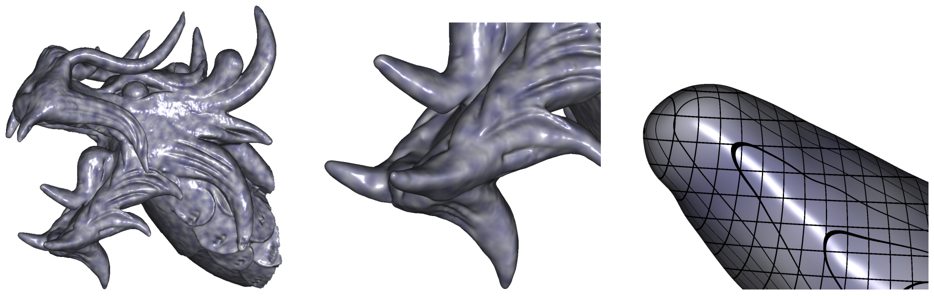

Fig. 7. Ray casted isosurface of the *Asian Dragon* head (256^3 voxels) with noise-based procedural texturing. *Right*: close-up into the Dragons' mouth where the C^1-continuous boundary curves on the outside of each cube of $\Diamond$ are shown in black.

two bars show the timings of our GPU kernels (see Sect. 4.1), which are invoked when the surface needs to be reconstructed. The on-the-fly computation of coefficients is slightly faster than preparing the determining sets, since less data is written. The most expensive part in the reconstruction is the classification, i.e., the determination of the array $\mathbf{Q}_{\text{class}}$. This could be improved by using appropriate spatial data structures, e.g., min/max octrees, but the recursive nature of these data structures makes an efficient implementation challenging. In addition, the data structures have to be rebuilt if the data itself changes over time, which is not necessary in our simpler approach. Still, for our largest data sets the reconstruction times are in a range of a few hundred ms and in most cases even below the rendering times of a single frame. Compared to former optimized CPU reconstruction based on octrees [4], we achieve significant speed-ups of up to two orders of magnitude.

The per-frame rendering times in Fig. 6, right, are given for a 1280×1024 view port with the surface filling the entire screen. This corresponds to a worst-case scenario, where all active tetrahedra need to be processed and the number of tetrahedra is the limiting factor in both approaches (on-the-fly coefficients and using determining sets). The bottleneck is then determined by the vertex shader complexity. The fact that our on-the-fly vertex programs have about 1/3 more instructions than the version using determining sets is thus directly reflected in the rendering times. An analysis of our fragment programs with NVIDIA's tool *ShaderPerf* yields a peak performance of more than 430 Mio. fragments per second. Thus, fragment processing is already very efficient and further improvements should concentrate on the vertex programs, load balancing, and the reduction of processed geometry during rendering. E.g., for close inspections of the surface, significant speed ups can be achieved from hierarchical view frustum culling, where whole areas of the domain can be omitted and less tetrahedra are processed in the pipeline. This requires splitting up the arrays $\mathbf{T}_{\text{active}}$ based on a coarse spatial partition of $\Diamond$. As [4] have shown, in this case we can expect that frame-rates increase by an order of magnitude.

6 Conclusion

We have shown that interactive and high-quality visualization of volume data with varying isosurfaces can be efficiently performed on modern GPUs. Both, isosurface reconstruction and rendering, are hereby performed using a combined CUDA and graphics pipeline. Our approach benefits strongly from the mathematical properties of the splines. Memory requirements for geometry encoding are significantly reduced using instancing. The method scales well with the fast developing performance of modern graphic processors, and will directly benefit from increased numbers of multiprocessors and texture units. The proposed algorithm can be used for an interactive variation of isolevels, as well as for applications where the data itself varies over time, e.g., simulations and animations.

References

1. C.L. Bajaj. *Data Visualization Techniques.* John Wiley & Sons, 1999.
2. S. Marschner, R. Lobb. An evaluation of reconstruction filters for volume rendering. *IEEE Vis.,* 100–107, 1994.
3. T. Sorokina, F. Zeilfelder. Local Quasi-Interpolation by cubic C^1 splines on type-6 tetrahedral partitions. In *IMJ Numerical Analysis,* 27:74–101, 2007.
4. T. Kalbe, F. Zeilfelder. Hardware-Accelerated, High-Quality Rendering Based on Trivariate Splines Approximating Volume Data. *Eurographics,* 331–340, 2008.
5. W.E. Lorensen, H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH 87,* 21(5):79–86, 1987.
6. M. Hadwiger, P. Ljung, C. Rezk-Salama, T. Ropinski. GPU-based Ray Casting. In *Annex Eurographics,* 2009.
7. J. Krüger, R. Westermann. Acceleration techniques for GPU-based volume rendering. *IEEE Vis.,* 287–292, 2003.
8. J. Mensmann, T. Ropinski, K. Hinrichs. Accelerating Volume Raycasting using Occlusion Frustum. *IEEE/EG Symp. on Vol. & Point-Based Gr.,* 147–154, 2008.
9. NVIDIA CUDA Compute Unified Device Architecture. NVIDIA Corp., 2008.
10. N. Tatarchuk, J. Shopf, C. DeCoro. Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline. *SIGGRAPH 2007 courses,* 122–137, 2007.
11. L. Barthe, B. Mora, N. Dodgson, M.A. Sabin. Triquadratic reconstruction for interactive modelling of potential fields. In *Shape Modeling Int.,* 145–153, 2002
12. M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *CGF,* 24(3):303–312, 2005
13. G. Reis. Hardware based Bézier patch renderer. In *Proc. of IASTED Visualization, Imaging, and Image Processing (VIIP),* 622–627, 2005.
14. C. Loop, J. Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *ACM Trans. on Graphics,* 25(3):664–670, 2006.
15. J. Seland, T. Dokken. Real-Time Algebraic Surface Visualization. In *Geometric Modelling, Numerical Simulation, and Optimization*, Springer, 163–183, 2007.
16. G. Blelloch. Vector Models for Data-Parallel Computing. In *The MIT Press,Cambridge, MA,* 1990
17. M. Harris, S. Sengupta, J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3,* Addison-Wesley, 677–696, 2007.
18. M.-J. Lai, L.L. Schumaker. Spline functions on Triangulations. Cambridge University Press, 2007.
19. L. Ramshaw. Blossoming: A Connect-the-Dots Approach to Splines. 1987.