EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (*2005*), *Number 3*

Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces

Markus Hadwiger Christian Sigg* Henning Scharsach Katja Bühler Markus Gross*

VRVis Research Center * ETH Zürich

Figure 1: *We render high-quality implicit surfaces on regular grids, e.g., distance fields or medical CT scans, in real-time without pre-computing additional per-voxel information. Gradients with C^1 continuity, second-order derivatives, and surface curvature are computed exactly for each output pixel using tri-cubic filtering. Applications include surface interrogation and visualizing levelset computations by color mapping curvature measures (center), and ridge and valley lines (left and right).*

Abstract

This paper presents a real-time rendering pipeline for implicit surfaces defined by a regular volumetric grid of samples. We use a ray-casting approach on current graphics hardware to perform a direct rendering of the isosurface. A two-level hierarchical representation of the regular grid is employed to allow object-order and image-order empty space skipping and circumvent memory limitations of graphics hardware. Adaptive sampling and iterative refinement lead to high-quality ray/surface intersections. All shading operations are deferred to image space, making their computational effort independent of the size of the input data. A continuous third-order reconstruction filter allows on-the-fly evaluation of smooth normals and extrinsic curvatures at any point on the surface without interpolating data computed at grid points. With these local shape descriptors, it is possible to perform advanced shading using high-quality lighting and non-photorealistic effects in real-time.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism Color, shading, shadowing, and texture

1. Introduction

Rendering isosurfaces represented implicitly by a volume of function samples is an important task in visualization, for example in medical applications, where volume data are naturally acquired directly, e.g., through CT or MRI scans, as well as a wide spectrum of other graphics disciplines including modeling and animation [MBWB02], and levelset simulation [LKHW03]. More general, implicit models are often specified and modified on volumetric grids such as regularly sampled distance fields, e.g., in levelset methods. Implicit representations naturally represent shapes of complex and changing topology. However, a major limitation of implicits is that the isosurface has to be extracted from the underlying volumetric representation for display. High-quality rendering at interactive speeds is a major bottleneck, particularly when the isosurface changes over time. When an implicit is represented by a discrete set of samples, rendering involves reconstruction of the data and the reconstruction filter is of crucial importance for image quality, especially for gradient reconstruction [MMK*98].

We present a real-time rendering pipeline for isosurfaces of dense volumetric grids of function samples that achieves

both high rendering quality and performance on current consumer graphics hardware (GPUs). Our algorithms are generally independent of specific hardware but we assume support for volumetric textures, render-to-texture and looping in fragment programs (e.g., ShaderModel 3.0). We address several shortcomings in existing GPU isosurface rendering approaches, particularly the lack and inefficiency of advanced shading, and texture memory usage. Modern GPUs are able to perform standard ray-casting of small regularly sampled data sets [KW03]. However, advanced shading, e.g., curvature-based transfer functions [HKG00, KWTM03], is still the domain of off-line rendering. The amount of texture memory limits data sizes significantly. This problem is aggravated by the demand of high-quality rendering for voxel data of 16-bit precision or more and lossless compression.

As a central part of our rendering pipeline, we support tri-cubic filtering throughout. Cubic filters allow for precise evaluations of differential properties of the isosurface, such as the normal and curvature, which both play a vital role in visualization, modeling, and simulation. These shape descriptors can be used for various advanced shading effects such as accessibility shading [Mil94], visualizing implicit surface curvature [KWTM03], and flow along curvature directions [vW03]. See Figure 1 for examples. In contrast to direct volume rendering, for isosurfaces only one sample position contributes to the color of a single pixel. Therefore, our method employs a ray-casting pass only for determining ray/surface intersections, and defers the computation of surface shape descriptors and shading to image space, where they are evaluated once per visible surface sample only. On-demand caching techniques are employed to dynamically download bricks of data only when they contain parts of the isosurface. Because only a small fraction of the grid samples contributes to the definition of an isosurface, this leads to significant reduction of texture memory usage without the need for lossy compression. See Figure 2 for an example. In summary, the combination of real-time performance and high quality yields a general-purpose rendering front-end for many powerful applications of implicit surfaces. The major contribution is a system that integrates the following:

- Tri-cubic filtering and high-quality shading with non-photorealistic effects using on-the-fly computation of smooth second-order geometric surface properties.
- Object space culling and empty space skipping without any per-sample cost during ray-casting.
- Precise ray/surface intersections without global oversampling, by combining adaptive resampling and iterative refinement of intersections with image order complexity.
- A very simple 3D brick cache alleviates GPU memory limitations significantly.
- Principal surface curvatures are computed in a simpler way than in previous approaches [KWTM03].

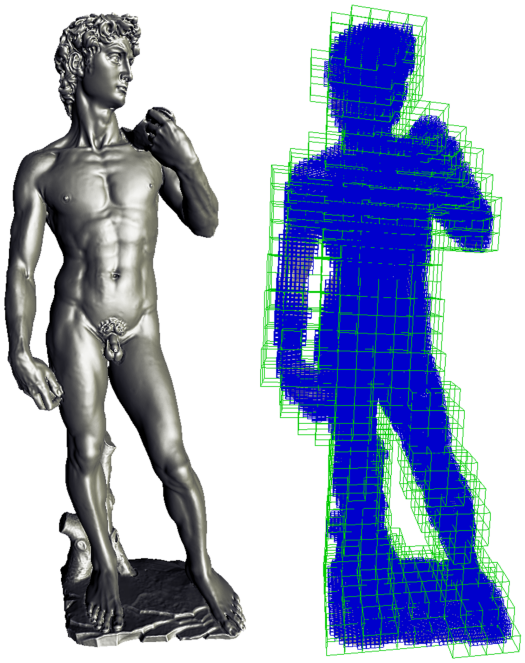

Figure 2: *Michelangelo's David extracted and shaded with tri-cubic filtering as isosurface of a 576x352x1536 16-bit distance field at 10 fps. The distance field is subdivided into two levels: a fine level for empty space skipping during ray-casting (blue) and a coarse level for texture caching (green).*

Previous Work

Our work is related to a large amount of previous research on volume rendering and rendering isosurfaces of volumetric data such as CT or MRI scans, as well as the area of implicit surfaces in general, especially when an implicit is represented by a grid of function samples, e.g., in levelset methods [MBWB02]. Although isosurfaces are often converted to triangle meshes for rendering [LC87], this produces very complex models and interactive changes of the isovalue or the volume itself are difficult to deal with. The two major approaches for rendering isosurfaces directly are ray-casting [Bar86, Lev88], and sampling ray-surface intersections on graphics hardware via slicing [WE98]. Although implicits are well-suited for finding guaranteed ray-surface intersections [KB89], precise computations and high-quality reconstruction are expensive. Hence interactive rates with high-quality or analytic ray-surface intersections and gradients have only been achieved by implementations using multiple CPUs [PSL*98, PPL*99] or clusters [DPH*03]. Different trade-offs have been presented [NMHW02, MKW*04]. The parallel architecture of GPUs has also been used extensively for interactive volume rendering, usually via slicing [WE98, EKE01]. In addition to hybrid CPU/GPU ray-casting [WS01], ray-casting on GPUs has been shown for small data sets [KW03, Gre04]. Adaptive sampling rates can be achieved by using pre-computed importance volumes [RGW*03]. Aliasing artifacts due to undersampling during slicing can be reduced by pre-integration [EKE01],

which also yields sharp isosurface boundaries, but assumes piecewise linear data variation along all viewing rays instead of tri-linear or higher-order reconstruction. All other previous interactive approaches for rendering isosurfaces of volume data are restricted to tri-linear data interpolation, and usually interpolate gradients pre-computed at grid points. If the volume data have not been scanned directly, signed distance fields are a natural choice as input for our rendering pipeline [WSE99]. Levelset methods change the distance fields dynamically and have many powerful applications such as surface editing and processing operators [MBWB02], and surface deformations [TO99]. Implicits are also well-suited for CSG modeling. In addition to reconstructing an isosurface, we are computing implicit surface curvature [KWTM03]. The space of principal curvature magnitudes is intuitive for shape depiction [HKG00], and can be used for non-photorealistic volume rendering [RE01] such as ridge and valley lines [IFP95]. Curvature directions can be visualized effectively by advecting dense noise textures [vW03], which we do entirely in image space [LJH03] on a per-pixel basis. The texture memory limitation for large volumes has been tackled by various means of lossy compression [GWGS02, SW03], which are not well suited for high-quality rendering. Texture packing has been used for static lossless compression [KE02], improved rendering performance [LMK03], and sparse levelset computations [LKHW03]. Octrees have also been used [LHJ99, WWH*00]. Our texture caching approach combines adaptive texture look-ups during rendering [KE02] with dynamically updated packed data [LKHW03].

2. Pipeline Overview

This section gives a high-level overview of our rendering pipeline, which is illustrated in Figure 3. The basic input is a regularly sampled scalar volume. The first stage (top row of Figure 3) performs ray-casting through the volume in order to obtain a floating point image of ray/isosurface intersection positions in volume coordinates, which drives the following stages in image space (lower two rows of Figure 3). The ray-casting stage (Section 3) is the only part of the pipeline that has object space complexity. All other computations (such as computing derivatives; Section 4.1) and shading operations (such as color-coding curvature; Section 4.2) are deferred to image space and thus have image space complexity [ST90].

The volume is subdivided into two regular grid levels: a fine level to facilitate empty space skipping (Section 3.1), and a coarse level to circumvent memory limitations of graphics hardware (Section 3.4). We call the elements of the fine subdivision level *blocks*, and those of the coarse level *bricks*. For each block we track min-max values of a set of voxels. Rays are started on block bounding faces and cast into the volume using adaptive sampling (Section 3.2). The last operation of the ray-casting stage iteratively refines isosurface hit-points (Section 3.3). This is done with a constant number of steps of image space complexity and is thus

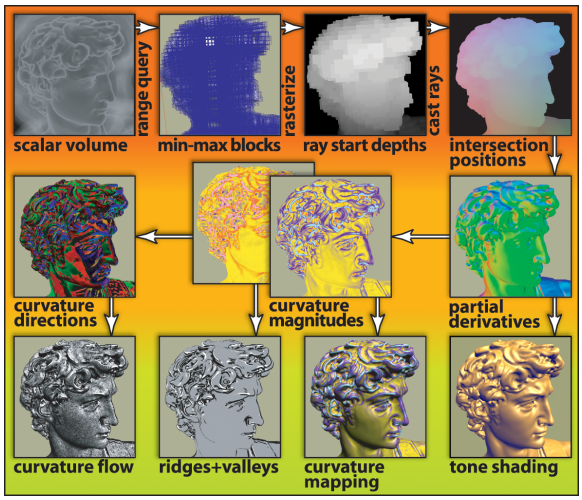

Figure 3: *Overview of our rendering pipeline. The top row operates with object space complexity until the refinement of ray/isosurface intersection positions. The middle row stages compute differential surface properties with image space complexity, and the bottom row stages perform deferred shading in image space.*

the transition from object to image space. The result of hit-point refinement is an image of high-quality ray/isosurface intersection positions. If the whole volume does not fit into graphics memory, rays are cast through a dynamic cache texture storing active bricks (Section 3.4). The cache is updated on-the-fly according to the current isovalue. An additional low-resolution texture references the positions of bricks of the volume in the cache.

The image space pipeline stages generate a series of images of differential isosurface properties, which are then used in a final shading pass to generate an output image using a variety of shading styles. Surface properties are computed at the exact positions of ray/isosurface intersections specified by the intersection image. Computing the first and second partial derivatives of the scalar volume yields floating point images for the components of the gradient and the Hessian matrix (Section 4.1). These derivatives are then used to compute curvature measures, which are likewise written into floating point images. The output image is generated in a final image space shading pass with a variety of effects that build on the shape descriptors computed before. The gradient image can be used for all shading models that require a surface normal, such as standard Blinn-Phong or tone shading. Curvature measures can be mapped to colors via 1D or 2D transfer functions, which is well-suited for shape depiction. For example drawing ridge and valley lines without generating actual line primitives. Pixels that correspond to ridge or valley areas are identified on a per-pixel basis via a curvature transfer function. Curvature directions are also effective shape cues, and we illustrate the curvature field on the isosurface with image space flow advection.

3. Ray-Casting

The basic idea of GPU-based ray-casting is to store the entire volume in a single 3D texture, and drive a fragment program that casts rays into the volume. Each pixel corresponds to a single ray $\mathbf{p}(t,x,y) = \mathbf{c} + t\ \mathbf{d}(x,y)$ in volume coordinates. Here, the normalized direction vector $\mathbf{d}(x,y)$ can be computed from the camera position $\mathbf{c}$ and the screen space coordinates (x,y) of the pixel. The range of depths $[t_{start}(x,y), t_{exit}(x,y)]$ which has to be searched for an isosurface intersection is computed per frame during initialization. In the simplest case, t_{start} is obtained by rasterizing the front faces of the volume bounding box with the corresponding distance to the camera. Rendering the back faces of the bounding box yields the depths t_{exit} of each ray exiting the volume.

In contrast to earlier approaches, we are using a single rendering pass and looping in the fragment shader for casting through the volume in front-to-back order instead of multiple passes [KW03], and employ object-order in addition to image-order empty space skipping. Most importantly, we overcome the following limitations:

- Empty space skipping overhead is reduced by using a two-level approach. Most empty space is skipped with no cost using modified ray segments $[t_{start}(x,y), t_{exit}(x,y)]$. Only for a small number of samples empty space has to be skipped on a sample-by-sample basis, which is accelerated via an adaptive sampling strategy.
- The quality of ray/isosurface intersection positions is refined by an iterative bisection procedure, which yields quality identical to much higher constant sampling rates [KW03] except at silhouette edges. A simple adaptive approach improves the quality of silhouette edges, without significant book-keeping overhead [RGW*03].
- The entire volume is not required to fit in GPU memory. Instead of casting through the original volume, we sample a brick cache texture storing only bricks intersected by the isosurface. Fast culling and LRU cache brick replacement allow changing the isovalue in real-time.

3.1. Empty Space Skipping

In order to facilitate object-order empty space skipping without per-sample overhead, we maintain min-max values of a regular subdivision of the volume into small blocks, e.g., with 4^3 or 8^3 voxels per block. These blocks do not actually re-arrange the volume. For each block, a min-max value is simply stored in an additional structure for culling. If the whole volume does not fit in GPU memory, however, a second level of coarser bricks is maintained, which is described in Section 3.4. Whenever the isovalue changes, blocks are culled against it using their min-max information and a range query [CSS98], which determines their active status. See Figure 4. The view-independent geometry of active block bounding faces that are adjacent to inactive blocks is kept in GPU memory for fast rendering.

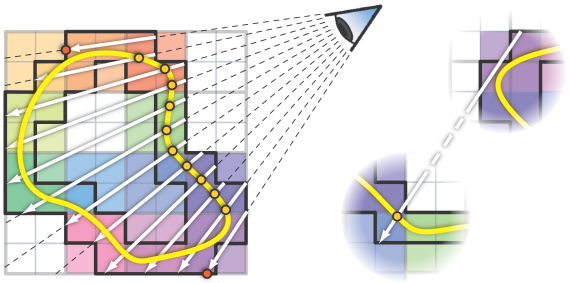

Figure 4: *Ray-casting with object-order empty space skipping. The bounding geometry (black) between active and inactive blocks that determines start and exit depths for the intersection search along rays (white) encloses the isosurface (yellow). Colored bricks of 2x2 blocks reference bricks in the cache texture (Figure 6). White bricks are not in the cache. Actual ray termination points are shown in yellow and red, respectively.*

In order to obtain ray start depths $t_{start}(x,y)$, the front faces of the block bounding geometry are rendered with their corresponding distance to the camera. The front-most points of ray intersections are retained by enabling a corresponding depth test (e.g., `GL_LESS`). For obtaining ray exit depths $t_{exit}(x,y)$ we rasterize the back faces with an inverted depth test that keeps only the farthest points (e.g., `GL_GREATER`). Figure 4 shows that this approach does not exclude inactive blocks from the search range if they are enclosed by active blocks with respect to the current viewing direction. The corresponding samples are skipped on a per-sample basis early in the ray-casting loop. However, most rays hit the isosurface soon after being started and are terminated quickly (yellow points in Figure 4, left). Only a small number of rays on the outer side of the isosurface silhouette are traced for a larger distance until they hit the exit position of the block bounding geometry (red points in Figure 4, left). The right side of Figure 4 illustrates the worst case scenario, where rays are started close to the view point, miss the corresponding part of the isosurface, and sample inactive blocks with image-order empty space skipping until they enter another part of the isosurface bounding geometry and are terminated or exit without any intersection. In order to minimize the performance impact when the distance from ray start to exit or termination is large, we use an adaptive strategy for adjusting the distance between successive samples along a ray.

3.2. Adaptive Sampling

In order to find the position of intersection for each ray, the scalar function is reconstructed at discrete sampling positions $\mathbf{p}_i(x,y) = \mathbf{c} + t_i\mathbf{d}(x,y)$ for increasing values of t_i in $[t_{start}, t_{exit}]$. The intersection is detected when the first sample lies behind the isosurface, e.g., when the sample value is smaller than the isovalue. Note that in general the exact intersection occurs somewhere between two successive samples. Due to this discrete sampling, it is possible that an intersec-

tion is missed entirely when the segment between two successive samples crosses the isosurface twice. This is mainly a problem for rays near the silhouette. Guaranteed intersections even for thin sheets are possible if the gradient length is bounded by some value L [KB89]. Note that for distance fields, L is equal to 1. For some sample value f, it is known that the intersection at isovalue ρ cannot occur for any point closer than $h = |f - \rho|/L$. Yet, h can become arbitrarily small near the isosurface, which would lead to an infinite number of samples for guaranteed intersections.

We use adaptive sampling to improve intersection detection. The actual intersection position of an intersection that has been detected is then further refined using the approach described in Section 3.3. We have found that completely adaptive sampling rates are not well suited for implementations on graphics hardware. These architectures use multiple pipelines where small tiles of neighboring pixels are scan-converted in parallel using the same texture cache. With completely adaptive sampling rate, the sampling positions of neighboring pixels diverge during parallel execution, leading to under-utilization of the cache. Therefore, we use only two different discrete sampling rates. The *base sampling rate* r_0 is specified directly by the user where 1.0 corresponds to a single voxel. It is the main tradeoff between speed and minimal sheet thickness with guaranteed intersections. In order to improve the quality of silhouettes (see Figure 5), we use a second *maximum sampling rate* r_1 as a constant multiple of r_0: $r_1 = nr_0$. We are currently using $n = 8$ in our system. However, we are not detecting silhouettes explicitly at this stage, because it would be too costly. Instead, we automatically increase the sampling rate from r_0 to r_1 when the current sample's value is closer to the isovalue ρ by a small threshold δ. In our current implementation, δ is set by the user as a quality parameter, which is especially easy for distance fields where the gradient magnitude is 1.0 everywhere. In this case, a constant δ can be used for all data sets, whereas for CT scans it has to be set according to the data.

3.3. Intersection Refinement

Once a ray segment containing an intersection has been detected, the next stage determines an accurate intersection position using an iterative bisection procedure. In one iteration, we first compute an approximate intersection position assuming a linear field within the segment. Given the sample values f at positions $\mathbf{x}$ for the near and far ends of the segment, the new sample position is

$$\mathbf{x}_{new} = (\mathbf{x}_{far} - \mathbf{x}_{near}) \frac{\rho - f_{near}}{f_{far} - f_{near}} + \mathbf{x}_{near} \qquad (1)$$

Then the value f_{new} is fetched at this point and compared to the isovalue ρ. Depending on the result, we update the ray segment with either the front or the back sub-segment. If the new point lies in front of the isosurface (e.g. $f_{new} > \rho$), we set $\mathbf{x}_{near}$ to $\mathbf{x}_{new}$, otherwise we set $\mathbf{x}_{far}$ to $\mathbf{x}_{new}$ and repeat. We have found empirically that a fixed number of four iteration steps is enough for high-quality intersection positions.

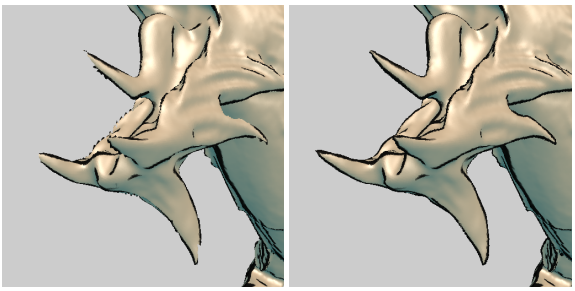

Figure 5: *The left image illustrates a small detail of the asian dragon model with a sampling rate of 0.5. On the right, adaptive sampling increases the sampling rate to 4.0 close to the isosurface. Note that except at the silhouettes there is no visible difference due to iterative refinement of intersections.*

3.4. Brick Caching

For any possible isovalue, many of the blocks described in Section 3.1 do not contain any part of the isosurface. In addition to improving rendering performance by skipping empty blocks, this fact can also be used for reducing the effective memory footprint of relevant parts of the volume significantly. Whenever the isovalue changes, the corresponding range query also determines the active status of bricks of coarser resolution, e.g., 32^3 voxels. The colored squares in Figure 4 depict these bricks with a size of 2*x*2 blocks per brick for illustration purposes. In contrast to blocks, bricks re-arrange the volume and include neighbor samples to allow filtering without complicated look-ups at the boundaries, i.e., a brick of resolution n^3 is stored with size $(n+1)^3$ [KE02]. This overhead is inversely proportional to the brick size, which is the reason for using two levels of subdivision. Small blocks fit the isosurface tightly for empty space skipping and larger bricks avoid excessive storage overhead for memory management.

In order to decouple the volume size from restrictions imposed by GPUs on volume resolution (e.g., 512^3 on NVIDIA GeForce 6) and available video memory (e.g., 256MB), we can perform ray-casting directly on a re-

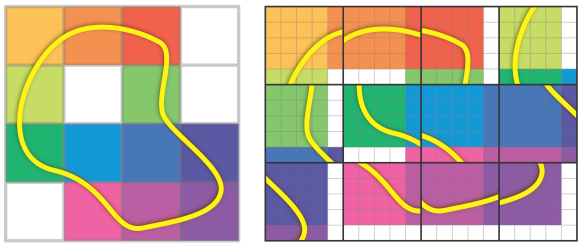

Figure 6: *A low-resolution brick reference texture (left) stores references from volume coordinates to texture cache bricks (right). The reference texture is sampled in the fragment shader to transform volume coordinates into brick cache texture coordinates. White bricks denote* null *references for bricks that are not resident in the cache.*

arranged brick structure. Similar to the idea of adaptive texture maps [KE02], we maintain an additional low-resolution floating point reference texture (e.g., 16^3 for a 512^3 volume with 32^3 bricks) storing texture coordinate offsets of bricks in a single brick cache texture that is always resident in GPU memory (e.g., a 512x512x256 texture). However, both the reference and the brick cache texture are maintained dynamically and not generated in a pre-process [KE02]. Figure 6 illustrates the use of the reference and brick cache textures. Note that since no gradient reconstruction or shading is performed during ray-casting, no complicated neighbor lookups are required at this stage. When the isovalue changes, bricks that potentially contain a part of the isosurface are downloaded into the brick cache texture. Inactive bricks are removed with a simple LRU (least recently used) strategy when their storage space is required for active bricks. Bricks that are currently not resident in the cache texture are specially marked at the corresponding position in the reference texture (shown as white squares in Figure 6). During ray-casting, samples in such bricks are simply skipped.

4. Deferred Shading

While the last section showed how to compute accurate ray/surface intersections for each pixel, this section describes how to turn the position image into a high quality rendering using deferred shading. All algorithms described here have image space complexity, meaning that they are independent of the size of the grid data. Each pass of the deferred shading stage writes a different property of the intersection position to an off-screen pixel buffer [ST90]. The result of one pass can serve as input for successive passes by mapping the pixel buffer as a texture. The final shading pass uses the property images to render the shaded isosurface to the viewport.

Figure 7: *Color mapping of maximum principal curvature magnitude using a 1D color look-up table (dragon data set with 512x512x256 samples).*

operation	#passes	inputs	outputs
Ray-Casting	3 [3]	*volume*	pos
Gradient	3 [6]	pos, *volume*	**g**
Hessian	6 [12]	pos, *volume*	**H**
Curvature	1 [13]	**g**, **H**	$\kappa_{1,2}$, $\mathbf{e}_{1,2}$
Shading	1 [14]	pos, **g**, $\kappa_{1,2}$, $\mathbf{e}_{1,2}$	image

Table 1: *Number of image space rendering passes and required input images for differential properties and deferred shading. Pass counts in brackets denote total number of passes after the intersection position computation.*

4.1. Differential Surface Properties

The appendix describes briefly how we quickly evaluate cubic reconstruction filters and their partial derivatives. See [SH05] for more details. This section shows that these basic capabilities can be exploited to calculate differential properties of isosurfaces from the scalar volume. In our implementation on a NVIDIA GeForce 6800, each property is calculated in one to six rendering passes, where each of these passes renders only a single screen-aligned quad in order to invoke the fragment shader for every output pixel. An overview of the number and types of rendering passes is given in Table 1.

Partial derivatives. The first differential property of the scalar volume that we need to reconstruct is its gradient $\mathbf{g} = \nabla f$, which we use as implicit surface normal and for curvature computations. The surface normal is the normalized gradient of the volume, or its negative, depending on the notion of being inside/outside the object: $\mathbf{n} = \pm\mathbf{g}/|\mathbf{g}|$. We compute **g** in three rendering passes, each of which evaluates a tri-cubic B-spline convolution sum in order to compute one of the three first-order partial derivatives via eight texture fetches from the 3D volume texture, plus three fetches from 1D filter weight textures [SH05]. The calculated gradient is stored in a single RGB floating point image, see Figure 3(derivatives). The Hessian $\mathbf{H} = \nabla\mathbf{g}$, comprised of all second partial derivatives of the volume, is calculated analogously. Due to symmetry, only six unique components need to be calculated, which is done in six rendering passes using either eleven or fourteen texture fetches each. The six calculated coefficients of **H** are stored in two RGB floating point images.

Extrinsic curvature. The first and second principal curvature magnitudes (κ_1, κ_2) of the isosurface can be estimated directly from the gradient **g** and the Hessian **H** [KWTM03], whereby tri-cubic filtering in general yields high-quality results. We do this in a single rendering pass, which uses the three partial derivative RGB floating point images generated by previous pipeline stages as input textures. The principal curvature magnitudes amount to two eigenvalues of the shape operator **S**, defined as the tangent space projection of the normalized Hessian:

$$\mathbf{S} = \mathbf{P}^T \frac{\mathbf{H}}{|\mathbf{g}|}\mathbf{P}, \qquad \mathbf{P} = \mathbf{I} - \frac{\mathbf{g}\mathbf{g}^T}{|\mathbf{g}|^2} \tag{2}$$

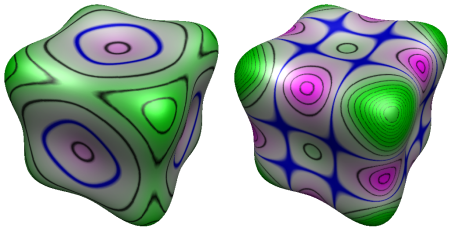

Figure 8: *Curvature Mapping of a* 64^3 *synthetic data set. Mean curvature* $(\kappa_1+\kappa_2)/2$ *(left), and Gaussian curvature* $\kappa_1\kappa_2$ *(right). Our renderer is capable to reproduce images from [KWTM03] at interactive rates. Data set and color mapping function are courtesy of Gordon Kindlmann.*

where **I** denotes the identity matrix. The eigenvalue corresponding to the eigenvector **g** vanishes, and the other two eigenvalues are the principal curvature magnitudes. Because one eigenvector is known, it is possible to solve for the remaining two eigenvectors in the two-dimensional tangent space without ever computing **S** explicitly. This results in reduced amount of operations and improved accuracy compared to the approach given in [KWTM03]. The transformation of the shape operator **S** to some orthogonal basis $(\mathbf{u},\mathbf{v})$ of the tangent space is given by

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = (\mathbf{u},\mathbf{v})^T \frac{\mathbf{H}}{|\mathbf{g}|}(\mathbf{u},\mathbf{v}) \qquad (3)$$

Eigenvalues of **A** can now be computed using the direct formulas for 2x2 matrices. The two eigenvectors of the shape operator **S** corresponding to the principal curvature directions are computed by transforming the eigenvectors of **A** back to three-dimensional object space.

$$\kappa_{1,2} = \frac{1}{2}\left(\mathrm{trace}(\mathbf{A}) \pm \sqrt{\mathrm{trace}(\mathbf{A})^2 - 4\det(\mathbf{A})}\right) \qquad (4)$$

$$\mathbf{e}_i = \kappa_i\mathbf{u} + (\kappa_i + a_{22} - a_{11})\mathbf{v} \qquad (5)$$

This amounts to a moderate number of vector and matrix multiplications, solving a quadratic polynomial, and three texture instructions. The curvature magnitudes and directions are rendered to two floating point targets.

4.2. Shading Effects

After the computation of differential surface properties, the resulting floating point images can be used for deferred shading in image space. Hence, all shading is decoupled from the volume and only calculated for actually visible pixels. This section outlines some of the example shading modes that we have implemented. This is only a small selection of possible rendering modes that can be used in our pipeline.

Shading from gradient image. The simplest shading equations depend on the normal vector of the isosurface. We have implemented standard Blinn-Phong shading and tone shading.

Figure 9: *Asian dragon data set (512x256x256). Left: tone shading. Right: tone shading blended with accessibility shading, allowing better depiction of local surface details.*

Curvature color mapping. The extrinsic curvature can be visualized on the isosurface by mapping curvature measures to colors via lookup textures. First and second principal curvatures, mean curvature $(\kappa_1+\kappa_2)/2$ and Gaussian curvature $\kappa_1\kappa_2$ can be visualized using a 1D lookup texture (see Figures 7 and 8) and give a good understanding of the local shape of the isosurface. Using a two-dimensional lookup texture for the (κ_1,κ_2) domain allows to highlight different structures on the surface. Figure 9 shows approximated accessibility shading [Mil94]. In this case, we have used a simple 1D curvature transfer function to darken areas with large negative maximum curvature. A 2D curvature function could also be employed for this purpose, giving finer control over the appearance.

Curvature-aligned flow advection. Direct mappings of principal curvature direction vectors to RGB colors are hard to interpret, see Figure 3(curvature directions). Instead of showing curvature directions directly, we visualize them with an approach based on image-based flow visualiza-

Figure 10: *Dense flow advected in the direction of maximum principal curvature (head of the David data set with* 512^3 *samples).*

tion [vW03]. In particular, we are advecting flow on the surface entirely in image space [LJH03] by computing advection on a per-pixel basis according to the underlying vector field of principal curvature directions. Image-based flow advection methods can be used on surfaces without parametrization by projecting a 3D flow field to the 2D image plane and advecting entirely in the image [vW03]. We do this by simply projecting each 3D curvature direction vector stored in the corresponding floating point image to the image plane immediately before performing advection for a given pixel. Image-based flow advection easily attains real-time rates, which complements the capability of our pipeline to generate the underlying, potentially unsteady, flow field in real-time. See Figure 10 for an example. A problem with advecting flow along curvature directions is that their orientation is not uniquely defined and thus seams in the flow cannot be entirely avoided [vW03]. Although these seams are visible when looking closely, we have found them to be not very disturbing in practice. Even though the flow field we are computing from curvature directions contains clearly visible patches (Figure 3: curvature directions), the resulting flow has much higher quality (Figure 10).

Non-photorealistic effects. Curvature information can be used for a variety of non-photorealistic rendering modes. We have implemented silhouette outlining taking curvature into account in order to control thickness, and depicting ridge and valley lines specified via colors in the (κ_1, κ_2) domain [KWTM03]. See Figures 11 and 8. In our pipeline, rendering modes such as these are simple operations that can be carried out in a single final shading pass, usually in combination with other parts of a larger shading equation, e.g., tone shading or solid texturing. We find the combination of curvature magnitude color maps and curvature-directed flow especially powerful for visualizing surface shape, e.g., as guidance during modeling.

5. Results

Volume rendering. Since the input to our rendering pipeline is an arbitrary scalar volume, it is naturally applicable to the rendering of isosurfaces such as the CT scan shown in Figure 11. We have integrated our renderer into an existing volume rendering framework as high-quality isosurface rendering front-end. In particular, real-time curvature estimation can be used to guide volume exploration, e.g., visualizing isosurface uncertainty, as has been proposed previously for off-line volume rendering [KWTM03].

Rendering from distance fields. For surface editing using a levelset approach, an initial implicit representation of the surface is usually generated by computing the signed distance to a triangle mesh. We used a variation of radially weighted linear fields [Nie04] to compute high resolution distance fields from triangle meshes, see Figures 7 and 9 for examples. Our rendering pipeline could easily be extended to include on the fly evaluation of CSG operations between multiple distance fields using min/max operations.

5.1. Rendering Performance

Table 2 gives performance numbers of our rendering pipeline corresponding to the figures shown in this paper. Except for very small volumes, the overall performance is dominated by the initial volume sampling step that computes approximate intersection positions. This fact is illustrated in Table 3. Although differential surface properties are expensive to compute in general, the fact that all of these computations have image space complexity combined with fast filtering decrease their impact on overall frame rate significantly. Even more important, the time spent in these computations is constant with respect to sampling rate and volume resolution. The same is true for intersection optimization via bisection. Table 4 illustrates the performance impact of different sampling rates. With respect to adaptive sampling, we compare constant sampling rates with the same rates for the maximum sampling rate r_1 that is used close to the isosurface (Section 3.2). We observe that although the overhead introduced by bricking is significant, it can be reduced via adaptive sampling so that overall performance is about 80-85% of rendering without bricking and without adaptive sampling.

data set	grid size	figure	fps
asian dragon	512x256x256	1	20.3
asian dragon	512x256x256	9	24.0
david head	512x512x512	1	15.3
david head	512x512x512	10	14.9
david	576x352x1536	2	10.3
cube	64x64x64	8	29.6
dragon	512x512x256	7	11.7

Table 2: *Performance of the renderings shown in the figures. Frame rates are given in frames per second for a 512x512 viewport. Four bisection steps have always been used, since they do not influence overall performance significantly.*

bounding geometry	ray-cast	differential properties	shading
1.7%	66.1%	31.0%	1.1%

Table 3: *Relative performance of the different stages of the pipeline for asian dragon rendering of Figure 1. Rendering performance is dominated by the surface intersection time.*

adaptive sampling	brick size	sampling rate (adaptive: r_1) 0.25	0.5	1	2	4	8
no	**none**	33.2	29.0	22.7	16.9	12.4	
no	**32**	23.8	19.5	16.1	11.7	7.2	
$r_1 = 8r_0$	**none**			34.6	27.4	20.3	15.2
$r_1 = 8r_0$	**32**			19.2	13.8	10.2	6.9

Table 4: *Rendering performance in frames per second corresponding to different sampling rates for asian dragon rendering of Figure 1. Brick caching introduces an additional texture indirection per sample (Section 3.4). Adaptive sampling (Section 3.2; $n = 8$) with bricking reduces this overhead compared to constant sampling.*

5.2. Discussion and Limitations

This section discusses some limitations of our system. A problem that can be seen in Figure 11, is that even when cubic filters are used, the curvature computed on actual scanned data contains visible noise. However, the quality of cubic filters is almost indistinguishable from filters up to order seven [KWTM03]. In any case it is important to use full 32-bit floating point precision for all GPU computations.

A limitation of our bisection approach for intersection is that in comparison to an analytic root search [DPH*03] or isolation of exactly one intersection [MKW*04], our discrete sampling with fixed step size does not guarantee correct detection of segments with multiple intersections. Furthermore, our bisection search might not find the intersection closest to the camera in such configurations.

A disadvantage of all deferred shading pipelines in general is the memory consumption of the image buffers. We are maintaining up to six window-sized images consisting of four 32-bit floating point channels each, which consumes a significant amount of GPU memory for high rendering resolutions and thus decreases the maximum volume or brick cache size.

Another consideration is whether to use an interpolating filter, such as tri-linear interpolation or Catmull-Rom cubic splines, or a smoothing filter such as the cubic B-spline for reconstruction purposes. A very good combination seems to be using an interpolating filter for value reconstruction, and a smoothing filter for reconstructing derivatives.

6. Conclusions

We have presented a rendering pipeline for real-time rendering of isosurfaces defined implicitly by regularly sampled scalar volumes. Using empty space skipping techniques and brick caching, we are able to render volumes of large sizes that would not fit into GPU texture memory at once at interactive rates. In comparison to volume rendering algorithms which perform color integration along the viewing ray, our method is optimized for rendering of isosurfaces. Because only one sample position contributes to the color of each pixel, differential surface properties can be computed on-the-fly in image space as part of the deferred shading stage. Due to its general nature, our pipeline is applicable to many practical problems involving implicit surfaces, such as volume rendering of scientific or medical data, modeling, morphing, and surface investigation using non-photorealistic techniques.

We would like to thank Gordon Kindlmann, Bob Laramee, Jiří Hladuvka, and Christof Rezk-Salama for their help and valuable contributions. The VRVis research center is funded in part by the Austrian Kplus project. The second author has been supported by Schlumberger Cambridge Research. The medical data sets are courtesy of Tiani MedGraph. The David model is courtesy of the Digital Michelangelo Project.

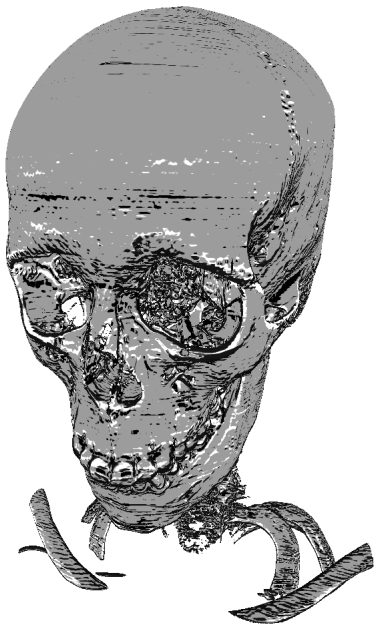

Figure 11: *Contours modulated with curvature in view direction, and ridges and valleys on an isosurface of a 512x512x333 CT scan of a human head.*

References

[Bar86] BARR A. H.: Ray tracing deformed surfaces. In *Proc. of SIGGRAPH '86* (1986), pp. 287 – 296.

[CSS98] CHIANG Y.-J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *Proc. of IEEE Visualization '98* (1998), pp. 167–174.

[DPH*03] DEMARLE D., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed interactive ray tracing for large volume visualization. In *Proc. of IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 87–94.

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. of Graphics Hardware 2001* (2001), pp. 9–16.

[Gre04] GREEN S.: Procedural volumetric fireball effect. In *NVIDIA SDK samples* (2004).

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. In *Proc. of IEEE Visualization 2002* (2002), pp. 53–60.

[HKG00] HLADUVKA J., KÖNIG A., GRÖLLER E.: Curvature-based transfer functions for direct volume rendering. In *Proc. of SCCG 2000* (2000), pp. 58–65.

[IFP95] INTERRANTE V., FUCHS H., PIZER S.: Enhancing transparent skin surfaces with ridge and valley lines. In *Proc. of IEEE Visualization '95* (1995), pp. 52–59.

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *Proc. of SIGGRAPH '89* (1989), pp. 297 – 306.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. of Graphics Hardware 2002* (2002), pp. 7–15.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 287–292.

[KWTM03] KINDLMANN G., WHITAKER R., TASDIZEN T., MÖLLER T.: Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. of IEEE Visualization 2003* (2003), pp. 513–520.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. of SIGGRAPH '87* (1987), pp. 163–169.

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37.

[LHJ99] LAMAR E., HAMANN B., JOY K.: Multiresolution techniques for interactive texture-based volume visualization. In *Proc. of IEEE Visualization '99* (1999), pp. 355–361.

[LJH03] LARAMEE B., JOBARD B., HAUSER H.: Image space based visualization of unsteady flow on surfaces. In *Proc. of IEEE Visualization 2003* (2003), pp. 131–138.

[LKHW03] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proc. of IEEE Visualization 2003* (2003), pp. 75–82.

[LMK03] LI W., MUELLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 317–324.

[MBWB02] MUSETH K., BREEN D. E., WHITAKER R. T., BARR A. H.: Level set surface editing operators. In *Proc. of SIGGRAPH 2002* (2002), pp. 330–338.

[Mil94] MILLER G.: Efficient algorithms for local and global accessibility shading. In *Proc. of SIGGRAPH '94* (1994), pp. 319–326.

[MKW*04] MARMITT G., KLEER A., WALD I., FRIEDRICH H., SLUSALLEK P.: Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proc. of Vision, Modeling, and Visualization* (2004), pp. 429–435.

[MMK*98] MÖLLER T., MÜLLER K., KURZION Y., MACHIRAJU R., YAGEL R.: Design of accurate and smooth filters for function and derivative reconstruction. In *Proc. of IEEE VolVis '98* (1998), pp. 143–151.

[Nie04] NIELSON G.: Radial hermite operators for scattered point cloud data with normal vectors and applications to implicitizing polygon mesh surfaces for generalized CSG operations and smoothing. In *Proc. of IEEE Vis. 2004* (2004), pp. 203–210.

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based first-hit ray casting. In *Proc. of VisSym 2002* (2002), pp. 77–86.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999), 238–250.

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Proc. of IEEE Visualization '98* (1998), pp. 233–238.

[RE01] RHEINGANS P., EBERT D.: Volume illustration: Nonphotorealistic rendering of volume models. In *Proc. of IEEE Visualization 2001* (2001), pp. 253–264.

[RGW*03] RÖTTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *Proc. of VisSym 2003* (2003), pp. 231–238.

[SH05] SIGG C., HADWIGER M.: Fast third-order texture filtering. In *GPU Gems 2, Matt Pharr (ed.)* (2005), Addison-Wesley, pp. 313–329.

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3D shapes. In *Proc. of SIGGRAPH '90* (1990), pp. 197–206.

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 293–300.

[TO99] TURK G., O'BRIEN J. F.: Shape transformation using variational implicit functions. In *Proc. of SIGGRAPH '99* (1999), pp. 335–342.

[vW03] VAN WIJK J.: Image based flow visualization for curved surfaces. In *Proc. of IEEE Visualization 2003* (2003), pp. 745 – 754.

[WE98] WESTERMANN R., ERTL T.: Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH '98* (1998), pp. 169–177.

[WS01] WESTERMANN R., SEVENICH B.: Accelerated volume ray-casting using texture mapping. In *Proc. of IEEE Visualization 2001* (2001), pp. 271–278.

[WSE99] WESTERMANN R., SOMMER O., ERTL T.: Decoupling polygon rendering from geometry using rasterization hardware. In *Proc. of Eurographics Workshop on Rendering* (1999), pp. 45–56.

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMAN K., ERTL T.: Level-of-detail volume rendering via 3D textures. In *Proc. of IEEE VolVis 2000* (2000), pp. 7–13.

Appendix: Fast tri-cubic interpolation

To reconstruct a texture with a cubic B-spline filter at texture coordinate x, the convolution sum

$$f(x) = w_0(x)\, f_{i-1} + w_1(x)\, f_i + w_2(x)\, f_{i+1} + w_3(x)\, f_{i+2} \qquad (6)$$

of four weighted neighboring texels f_i has to be evaluated. Note that the weights are periodic in the sample positions of the input texture. The number of texture fetches is reduced by employing the linear filtering capability of GPU texture units. Instead of fetching all four neighbors independently, we fetch two consecutive samples at the same time using linear interpolation and perform a single weighted sum.

$$f(x) = g_0(x)\, f_{\lfloor x \rfloor - h_0(x)} + g_1(x)\, f_{\lfloor x \rfloor + h_1(x)} \qquad (7)$$

The weight functions g_i and offset functions h_i are pre-computed and stored in a lookup texture.

$$g_0(x) = w_0(x) + w_1(x), \qquad h_0(x) = 1 - \frac{w_1(x)}{w_0(x) + w_1(x)} \qquad (8)$$

$$g_1(x) = w_2(x) + w_3(x), \qquad h_1(x) = 1 + \frac{w_3(x)}{w_2(x) + w_3(x)} \qquad (9)$$

The extension to three dimensional textures is straight-forward due to separability of tensor-product B-splines, and it is possible to evaluate a tri-cubic filter with 64 summands using just eight tri-linear texture fetches. In order to compute partial derivatives, the functions g_i and h_i are computed using the appropriate derivatives of w_i. More details can be found in [SH05].