Illustrative Volume Visualization Using GPU-Based Particle Systems

Roy van Pelt, Anna Vilanova, *Member*, *IEEE Computer Society*, and Huub van de Wetering

Abstract—Illustrative techniques are generally applied to produce stylized renderings. Various illustrative styles have been applied to volumetric data sets, producing clearer images and effectively conveying visual information. We adopt particle systems to produce user-configurable stylized renderings from the volume data, imitating traditional pen-and-ink drawings. In the following, we present an interactive GPU-based illustrative volume rendering framework, called VolFliesGPU. In this framework, isosurfaces are sampled by evenly distributed particle sets, delineating surface shape by illustrative styles. The appearance of these styles is based on locally-measured surface properties. For instance, hatches convey surface shape by orientation and shape characteristics are enhanced by color, mapped using a curvature-based transfer function. Hidden-surfaces are generally removed to avoid visual clutter, after that a combination of styles is applied per isosurface. Multiple surfaces and styles can be explored interactively, exploiting parallelism in both graphics hardware and particle systems. We achieve real-time interaction and prompt parametrization of the illustrative styles, using an intuitive GPGPU paradigm that delivers the computational power to drive our particle system and visualization algorithms.

Index Terms—Volume visualization, illustrative rendering, particle systems, consumer graphics hardware, parallel processing.

1 Introduction

There are various volume-rendering techniques that produce images from three-dimensional volumetric data sets. Typical examples of three-dimensional volumetric data are medical data obtained by computed tomography (CT) or magnetic resonance imaging (MRI). Throughout the years, the prevailing objective within the volume visualization field has been to generate images that closely resemble reality. However, a new volume-rendering branch investigates ways to create illustrative images from three-dimensional scalar data. Techniques from traditional art and illustration are incorporated in the volume-rendering process. The goal is to gain clarity compared to photo-realism by emphasizing important features, improving data exploration. Less relevant details are omitted and important aspects are highlighted, resulting in more comprehensible images [1], [2].

Illustrative rendering applications typically include a substantial amount of user-configurable parameters. Fast and reliable interaction with these parameters is of great importance in order to produce the desired illustrative styles. Furthermore, rendering illustrative styles from large volumetric data sets at interactive speed requires a considerable amount of computational power. The desired power in modern consumer graphics hardware has been engaged to increase overall performance and interaction speed of both illustrative and volume-rendering applications [3], [4].

- *R. van Pelt and A. Vilanova are with the Department of Biomedical Engineering, Biomedical Image Analysis, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands. E-mail: {r.f.p.v.pelt, a.vilanova}@tue.nl.*
- *H. van de Wetering is with the Department of Mathematics and Computer Science, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands. E-mail: h.v.d.wetering@tue.nl.*

Manuscript received 19 Feb. 2009; revised 24 July 2009; accepted 14 Sept. 2009; published online 9 Feb. 2010.
Recommended for acceptance by H.-C. Hege, D.H. Laidlaw, and R. Machiraju. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2009-02-0038.
Digital Object Identifier no. 10.1109/TVCG.2010.32.

We have adopted the illustrative concepts of the software-rendered VolumeFlies framework presented by Busking et al. [5]. This framework offers a general basis to produce illustrative depictions from volumetric data sets, as exemplified in Fig. 1. A variety of illustrative styles can be directly applied based on particle systems that operate on the volume data. VolumeFlies encompasses several styles that imitate traditional pen-and-ink drawing techniques.

We have chosen the flexible particle-based approach of the VolumeFlies framework, aiming to achieve interactive parametrization and rendering. GPU-based particle systems are able to process and visualize hundreds of thousands of particles in real time [6], [7]. We have investigated the latest graphics hardware to accelerate particle systems for illustrative volume visualization. We present a real-time framework where the algorithms from VolumeFlies [5] have been transformed to optimally benefit GPU parallelism. Both our particle system and our visualization algorithms are based on a novel paradigm for general purpose computations on the GPU (GPGPU). This paradigm is based on the GPU pipeline, and incorporates recent extensions of the shader model. Summarizing, the main contributions of this paper are:

- A GPGPU paradigm, serving as a model for a wide range of algorithms, exploiting computational parallelism (Section 3). Algorithms vary from data parallel sorting and searching to image and volume processing.
- A GPU-based generic particle system, employing this paradigm. This system incorporates energy minimization for particle redistribution based on the work by Meyer et al. [8] (Section 4.1).
- An interactive illustrative volume-rendering framework, initiating particle systems to create stylized depictions (Section 4.2). Styles resembling pen-and-ink illustration techniques, known from VolumeFlies [5], can now be applied to multiple volume features interactively. Additionally, curvature-based transfer

functions are adopted to emphasize ridges and valleys as presented by Kindlmann et al [9]. Most algorithms were transformed to use GPU parallelism achieving fast interaction and parametrization.

- A performance analysis, considering the performance gain compared to VolumeFlies, together with a scalability study of the GPU-based framework. Performance characteristics, together with adequate clarification, are provided for most images.

2 Previous Work

A particular extensive field of research investigates illustrative visualization [10]. We strive for an interactive framework offering a variety of illustrative styles. We are mainly concerned with hardware-rendered approaches producing pen-and-ink-style renderings from volume data. Before considering illustrative frameworks, we address separate illustrative styles.

Pen-and-ink-style drawing techniques convey object shape by varying tone. A customary technique that applies such shading is called stippling. *Image-based* approaches, such as presented by Secord [11], define shape by means of a stipple point distribution. The general disadvantage of image-based approaches is the precarious process to ensure frame coherence. Alternatively, object-space information can be combined with procedural textures to achieve frame coherence. Such a *hybrid approach* was presented by Baer et al. [1]. Furthermore, there are *object-based* approaches. Lu et al. [12] presented an interactive approach controlling the stipple density on a voxel basis. In contrast to previous work, we present interactive particle-based stippling, including visualization based on the stipple size.

Another traditional shading style is called hatching, producing tone variations by means of combined stroke patterns. The hatches convey surface shape by their directions, commonly guided by curvature information. Similar to the stippling methods, real-time surface hatching was implemented through procedural textures such as the *hybrid approach* presented by Praun et al. [13]. Besides, there are *object-based* approaches that generate the hatch stroke geometry, e.g., Nagy et al. [14]. As opposed to previous work, our curvature-based real-time hatching approach is based on a generic particle-system, and independent of viewport resolution.

Most illustrative techniques emphasize object boundaries by visualizing the contours or silhouettes. By definition, contour extraction is view-dependent. Apart from *image-based* filtering approaches, *object-based* methods exist that extract contours from volume data. A marching lines method was presented by Burns et al. [15]. A method based on "photic extremum lines," detecting changes in luminance, was presented by Xie et al. [16].

The presented references until now deal with a single illustrative style. Few papers combine several illustrative styles into a generic framework that allows flexible parametrization. Yuan and Chen [17] presented a combined illustrative framework. A more generic framework was presented by Busking et al. [5]. Their particle-based approach is flexible and configurable, and allows to apply all previously mentioned pen-and-ink styles, independent of the viewport resolution. In our work, we have renewed and extended the last mentioned framework. The performance was improved considerably resulting in real-time interaction and parametrization.

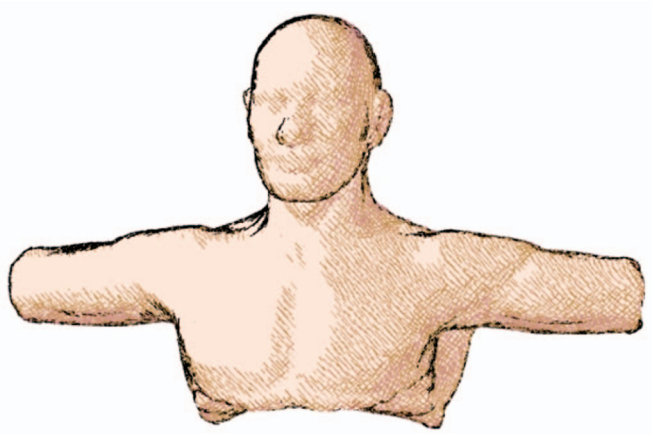


Fig. 1. VolFliesGPU: Combined illustrative styles on a voxelized torso model. The torso model is provided courtesy of Mangon and Dretakkis by the AIM@SHAPE Shape Repository.

The GPU is often employed for mathematical computations [18], [19], while the hardware is geared toward graphics processing. Many algorithms that can be executed data parallel, such as searching and sorting [20], show a substantial performance gain. GPGPU implementations are supported by new software platforms such as the Compute Unified Device Architecture (CUDA). CUDA allows implementation of algorithms using the GPU without knowledge of the hardware. For our work, we have chosen to directly employ the graphics hardware for general computations and rendering.

Particle systems offer a generic and flexible approach for both simulations and visualization. Moreover, operations on individual particles have the potential to be executed in parallel. The behavior of the particles is affected by rules from dynamics resulting in a particle flow [6], [7], [21], [22]. The visualization of the particles can be chosen freely; dots, arrows, and streamlines are common representations in flow simulations. This freedom of visual representation also benefits primarily visualization oriented goals, as presented by Meyer et al. [8]. They present an energy minimization that evenly distributes particles on implicit surfaces, facilitating point-based surface representations and mesh generation.

We present a particle-driven illustrative framework, which allows real-time parametrization and interaction. In this paper, we extend our earlier work [23] by curvature-based transfer functions and multisurface renderings. Furthermore, we elaborate on hidden-surface removal and performance aspects. First of all, we present our GPGPU paradigm describing a generic concept to execute data parallel algorithms on the GPU. The required performance was obtained by engaging our GPU paradigm. Finally, we present the performance results, our conclusions, and views on future work.

3 GPGPU Paradigm

The common GPGPU approach involves rendering a window-size quad, gathering input values from a 2D texture, and performing computations on a fragment basis [19]. Output values are returned through a render-to-texture operation. Although this approach offers a solid

TABLE 1
GPGPU Relations

Algorithm	GPU Implementation
Input	Read from Buffer Object or Texture
Processing	Vertex / Geometry Shading threads
Output	Transform Feedback to Buffer Object

solution for many algorithms [20], it is a rather counter-intuitive manner to use the GPU pipeline. We propose an intuitive and flexible approach to perform general computations on the GPU by employing new extensions in the shader model.

Processing an algorithm generally requires an input, a processing stage, and an output. Implementing these three basic steps on the GPU requires a suitable mapping to the stream processing pipeline. The general relations between an arbitrary algorithm and our GPGPU paradigm are listed in Table 1.

The actual paradigm, depicted in Fig. 2, is to be implemented upon the recently introduced *Unified Instruction Set Architecture*. Programming the *Unified Shader Model* or *Shader Model 4.0* supports flexible use of graphics hardware and relieves manual load-balancing.

Our paradigm implies an intensive use of both *vertex shaders* and *geometry shaders* while *fragment shaders* are not used. This stands in contrast to the commonly applied render-to-texture approaches.

Input: The input side requires a *buffer object* with the data to be processed, accompanied by a *proxy geometry* that commences the algorithm for each buffer element.

In the context of our paradigm, a proxy geometry comprises a set of vertices created CPU-side and stored on GPU memory using a *vertex buffer object* (VBO). The vertex positions encode indexes into the input buffer.

We generally choose a one-dimensional *texture buffer object* (TBO) representing the input data in an array-like form. Data values can now be obtained by means of a *vertex texel fetch* for each of the proxy-geometry vertices. Representing data by means of textures is common within the GPGPU community, however, VBOs suffice as well.

Processing: Active vertex or geometry shader threads, which serve as computation kernels, are triggered by rendering the vertices of the proxy geometry. The single program, multiple data architecture of the GPU enforces parallel processing of the input, applying an identical set of operations to each input value. This approach is only efficient when shader processing is unified.

Output: The output values are returned to a buffer object by means of a *transform feedback*. This transform feedback extension, or stream-out in DirectX terminology records vertex attributes for each of the processed primitives. Hence, the graphics hardware now provides support to return, or scatter, data from a vertex- or geometry-shading stage. All fragment processing can be discarded.

Computations often require to return multiple outputs, in that case a geometry shader thread is employed as processing kernel. The recently introduced geometry shading stage operates on the level of geometry primitives allowing creation and destruction of vertices. A point primitive from the proxy geometry encodes a single input data value from the input TBO. After processing the algorithm, a line strip

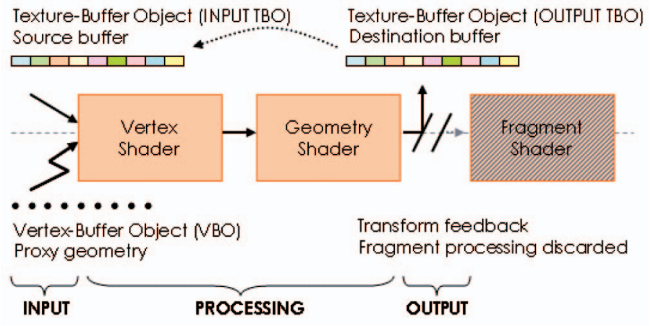


Fig. 2. GPGPU paradigm using transform feedback.

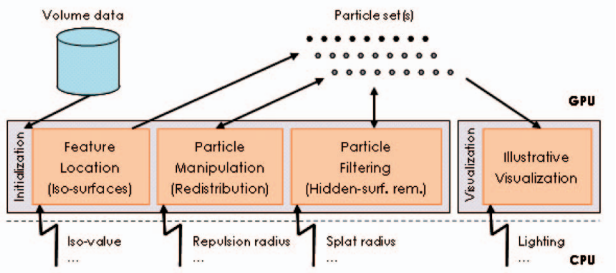


Fig. 3. The VolFliesGPU framework.

serves as an array of output values where the data elements are encoded by the line-strip vertices.

In the case where multiple output values are returned, the transform feedback offers a more flexible approach compared to rendering to multiple render targets (MRT). Not only can the output values be recorded to separate buffers, also the values can be recorded interleaved into a single buffer. Be aware that the performance of these methods varies for different hardware architectures.

This paradigm supports easy implementation of iterative approaches. The output texture buffer, containing the results of one computation stage, can be used as an input for the subsequent stage. This is depicted in Fig. 2 by the dashed arrow. Be aware that the correct input buffer for each computation stage is determined CPU-side.

The next section describes how the GPGPU paradigm was employed using particle systems, in order to create an interactive illustrative volume-rendering framework, called VolFliesGPU.

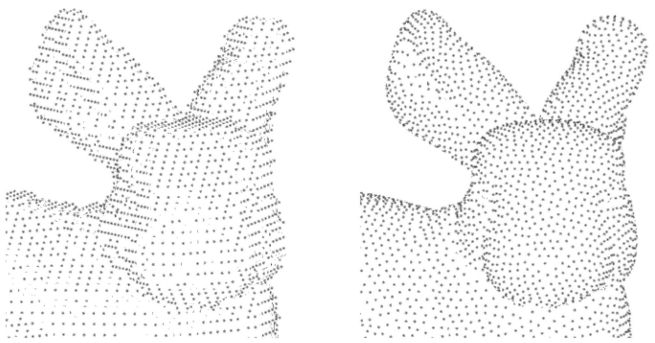
4 The VolFliesGPU Framework

The VolFliesGPU framework comprises an illustrative visualization framework for real-time pen-and-ink-style rendering of volume data. First, we present the initialization of the particle system followed by the various illustrative styles. The framework is based on the work by Busking et al. [5], and is schematically depicted in Fig. 3. Each of the framework modules are parallelized for which we have employed our GPGPU paradigm.

4.1 Initializing the Particle System

4.1.1 Feature Location

Initially the framework simply places a set of particles near a feature in the volume. For this paper, a feature is an isosurface at a user-selected isovalue, and the initialization samples the volume data at a user-defined grid. In a marching cubes like



(a) (b)

Fig. 4. (a) Before redistribution. (b) After redistribution. Particles redistribute evenly over the surface by exerting repulsive forces to nearby particles. The bunny model is the courtesy of Stanford University.

approach (Lorensen and Cline [24]), a particle is created between the sample positions and its neighboring grid point if the isosurface lies in between, see Algorithm 1.

Algorithm 1. FEATURE LOCATION
Input: Volume
Processing per grid point r:
1: **for** each neighboring grid point n **do**
2: sample Volume at grid points n and r
3: **if** isosurface is crossed **then**
4: output particle between grid points n and r
Output: Initial Particle Set
Complexity: $\mathcal{O}(v)$, with $v =$ number of voxels

The complexity of the initialization is in the order of the number of grid points. We aim for a real-time exploration of the volume data, and employ our GPGPU paradigm as a basis for the brute-force initialization.

Input: The proxy geometry comprises a 3D grid of equally spaced vertices, and the volume data consists of a 3D texture.

Processing: Each active shader thread determines the location of the new particles near an isosurface. For each grid point, values are compared to sampled values of the front, right, and top neighboring grid points. This comparison might result in zero, one, two, or three particle positions. Since the algorithm has a varying number of output values, the geometry shader is used to perform the comparisons.

The geometry shader thread returns an array of at most three vertices. Each vertex encodes the object-space position of a new particle.

Output: Finally, the vertices of the line-strip primitives are recorded into a texture buffer object. Each vertex represents a particle position (x, y, z).

4.1.2 Redistribution

The initial particle placement results in particles on a rectilinear grid, as depicted in Fig. 4a. A redistribution step moves the particles toward the actual isosurface location evenly spreading them over the surface. This comprises an energy minimization scheme similar to the work presented by Meyer et al. [8]. They present a general approach where particles exert repulsive forces to nearby particles while restraining them to the surface. The behavior of the particles can be adjusted by using different energy functions.

In this section, we present a GPU-driven equivalent of the redistribution approach based on our GPGPU paradigm. We aim at a fast and reliable redistribution, which terminates when the system reaches an equilibrium. The main challenge lies in the interparticle communications because neighbor interactions counteract parallel processing of the particles.

Repulsive forces between particles only operate within a user-defined influence radius. A rectilinear binning structure is introduced to provide locality within the volume. The bins are uniquely numbered and their size equals the radius of repulsion. Based on the spatial location, each particle obtains a bin number. We propose a four-step iterative redistribution scheme:

I. Sort the particles by bin number.
II. Create a bin lookup table.
III. Minimize energy.
IV. Verify if the system reached an equilibrium.

I) The particles will be sorted with their bin number as sorting key. Sorting has been applied to particle systems for depth ordering and collision detection [7]. GPU-based sorting [20] is typically data-independent, exploiting computational parallelism. We have adopted the *odd-even merge sort* algorithm by Kipfer and Westermann [20].

Replacing their fragment-based approach with our GPGPU paradigm, the particles in the input buffer are processed in parallel, performing the comparison operations. The intermediate results are stored into a new buffer through a transform feedback omitting any fragment processing. After each iteration step, the input and output buffers are swapped creating a simple ping-pong memory scheme.

II) Addressing all particles within the repulsion radius requires to examine the space taken by an environment of 27 bins surrounding a particle. Because a typical system contains considerably more particles than bins, the duration of this search process can be reduced by means of a bin lookup table. We create such a lookup table by performing a *binary search* for each bin, searching in the sorted particle array for the lowest index of any particle in that bin.

We use our GPGPU paradigm to engage vertex shader threads that perform a binary search through the particle buffer in parallel for all bins, searching for the corresponding particle index. The results are recorded to a newly created texture buffer object: The actual lookup table.

III) The third step performs the actual energy minimization scheme moving the system one step closer to an equilibrium. Every particle p_i has energy E_i and is expected to move to locally lower energy state by a steepest descent along the energy gradient direction. We adopt the two-step update scheme and the energy function E_i from the work presented by Meyer et al. [8]

$$\vec{g}_i = \nabla f(p_i); \quad \vec{n}_i = -\frac{\vec{g}_i}{|\vec{g}_i|}; \quad v_i = -\nabla E_i.$$

$$\text{Step 1}: p_i \leftarrow p_i + \left(I - \vec{n}_i \cdot \vec{n}_i^T\right)\vec{v}_i; \quad \vec{g}_i \leftarrow \nabla f(p_i). \tag{1}$$

$$\text{Step 2}: p_i \leftarrow p_i - f(p_i)\frac{\vec{g}_i}{|\vec{g}_i|^2}. \tag{2}$$

The gradient descent vector $\vec{v}_i$ is projected on the tangent plane by the matrix $I - \vec{n}_i \cdot \vec{n}_i^T$. Here, I is the identity matrix and $\vec{n}_i$ is the local normalized gradient direction $\vec{g}_i$ of the surface. Algorithm 2 performs a single minimization step.

Algorithm 2. ENERGY MINIMIZATION
Input: Volume, SortedParticles, BinLookup
Processing per particle SortedParticles:
1: Calculate displacement vector v_i (requires Volume),
Neighboring particles are obtained through BinLookup
2: Update particle position in tangent plane (Step 1)
3: Reproject position back to the isosurface (Step 2)
Output: Updated particles with lowered energy state

Complexity: (per iteration)
(I) Sorting: $\mathcal{O}(m\ log^2(m))$,
(II) Searching: $\mathcal{O}(d\ log_2(m))$,
(III) Two-step repulsion: $\mathcal{O}(mb)$, with
$m =$ the number of particles, and
$d =$ the average number of bins, and
$b =$ average number of particles in a bin

Input: Unlike Meyer et al., we execute this algorithm on the GPU employing our GPGPU paradigm. The volume data, stored in a 3D texture, the sorted particle texture buffer (I) and the bin-lookup texture buffer (II) are input to the computational kernels that perform the update scheme.

Processing: The actual two-step algorithm is executed in parallel by the GPU employing vertex shader threads.

Output: The transform feedback records the updated particle positions into the output texture buffer.

The algorithm performs faster compared to the software-driven approach despite the required additional steps. The scheme is iterative, which means that at this point the process could start over, moving the particle even closer to a steady state.

IV) In order to determine if the system has reached a steady state, we observe the difference of the total system energy from one iteration to the next. This global system energy can be calculated by summing the energy values at all particle locations. This summation is executed by means of a reduction operation, again using the GPGPU paradigm. Iterative pairwise addition of values in a 1D texture buffer containing the energy values results in the global system energy value. The texture buffer containing the global energy level is memory mapped such that it becomes available CPU-side. The energy level for each redistribution iteration is stored, and compared with the value of the previous iteration. If the difference is below user-defined threshold, the system has reached a steady state.

4.2 Visualizing the Particle System

A wide variety of illustrative styles can be applied to a particle set. We apply styles that resemble pen-and-ink illustrations on the visible particles. This section will address hidden-surface removal (Fig. 5), point-based stippling techniques (Fig. 8a), stroke-based hatching techniques (Fig. 8b), and contours (Fig. 8c).

4.2.1 Hidden-Surface Removal

The redistributed particle set conveys the shape of the isosurface typically depicted by point primitives. Since point primitives rarely occlude each other, particles on the whole surface area will be visible. This introduces visual clutter which can be resolved by means of hidden-surface removal. Without hidden-surface removal, it becomes hard to interpret the surface structure, and to decide if an object is facing forward or backward from the viewer.

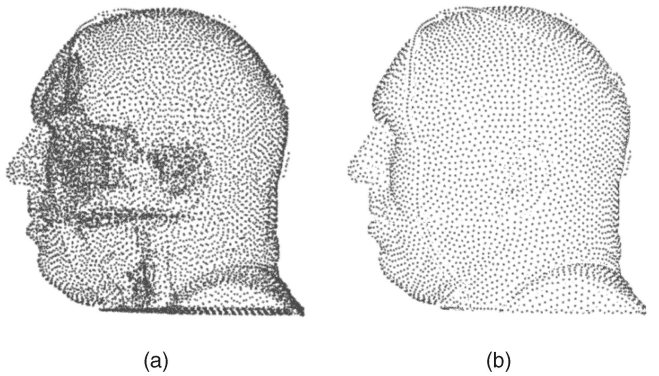


Fig. 5. (a) Before hidden-surface removal. (b) After hidden-surface removal. Hidden-surface removal through cone-splatting. The data set is courtesy of National Library of Medicine.

The hidden-surfaces are the parts of the surface area that become occluded when opaquely reconstructing a surface mesh. Visibility of the particles can, therefore, be determined by reconstructing a polygonal mesh of the isosurface, where occluded particles will be expelled from visualization.

Selecting multiple isosurfaces results in multiple particles sets. In that case, merely reconstructing and rendering of the mesh does not suffice, since it is likely for particles from distinct sets to become occluded erroneously.

Polygonal mesh reconstruction from particle sets was presented by Meyer et al. [25]. Unfortunately, it is a computationally expensive task which we prefer to omit.

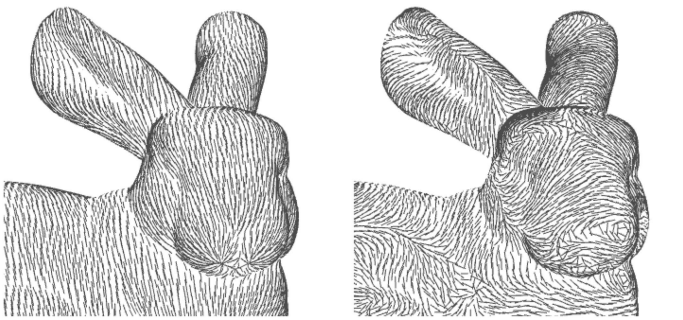
Katz et al. [26] presented an approach to calculate particle visibility without explicit surface reconstruction. They propose a *Hidden Point Removal* operator, computing visibility by extracting particles that reside on a convex hull of a transformed set of particle positions.

Alternatively, the isosurface can be estimated by means of surface splatting. We adopt the approach presented in the VolumeFlies framework [5], and splat the surface using cones (Fig. 5).

The *"cone-splatting"* algorithm determines particle visibility in two steps. In the first step, uniquely colored cones are splat on the isosurface and rendered to an offscreen buffer where apices of the cones coincide with the particle positions.

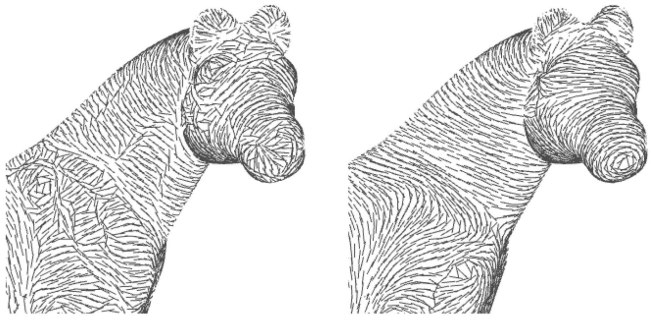
In the second step, visibility of the particles is determined by testing their corresponding color against the offscreen color buffer that was rendered in the first step. If there is no corresponding color, the particle is occluded by the splatted surface, and will be discarded. The approach is summarized in Algorithm 3.

Algorithm 3. HIDDEN-SURFACE REMOVAL
Input: Particles
Processing per particle (Particles):
Render cone with a unique color,
scaled and oriented towards the view plane.
Each cone starts from the particle position.
Output: offscreen color buffer



(a) (b)

Fig. 6. Hatches trace a single direction over the surface (a), or smoothed principal curvature directions (b). The bunny model is the courtesy of Stanford University.



(a) (b)

Fig. 7. Hatches follow the main principal curvature directions (a), or the smoothed directions (b). The horse model is the courtesy of Georgia Institute of Technology.

Input: Particles, offscreen color buffer
Processing per particle (Particles):
  if Corresponding color in offscreen buffer **then**
    Add to VisibleParticles
  else Discard
Output: VisibleParticles
Complexity: $\mathcal{O}(m)$, with $m =$ number of particles

The original approach of Busking et al. [5] is adapted to exploit modern graphics hardware capabilities. As opposed to other parts of the framework, the hidden-surface removal is not based on the GPGPU paradigm.

Rendering the cones in the first step is executed by the geometry shader. The geometry shader processes particle positions as input, and generates uniquely colored cones by means of triangle fans. The cones are oriented perpendicular to the viewplane and are anisotropically scaled, as proposed by Busking et al. [5]. Creating and rendering the geometry now benefits from the graphics hardware parallelism. The offscreen color buffer is captured, and serves as input for the second step.

In the second step, the corresponding color for each particle has to be tested against the offscreen color buffer efficiently. The original approach was to scan through the buffer to find the corresponding color. Instead, we project the particle position to the view plane, directly defining the index into the offscreen buffer.

Hidden-surface removal turns out to be of utmost importance to deliver the desired illustrative results and has been applied to all of the presented figures. Particle visibility can now be determined in real time, prior to rendering the particles. Since hidden-surface removal is view-dependent, it is significant to be able to execute this algorithm on a frame-to-frame basis, without loosing interactivity. Furthermore, performance of the illustrative styles applied to the particle set will increase since less particles need to be processed.

4.2.2 Stippling

Stippling is a technique where points are used to convey object shape. Our particles are rendered using point primitives, while varying the scale of the point primitives. Parameters can be configured interactively, adjusting brightness and contrast of the stipple visualization.

The point primitives are scaled with the result of the basic diffuse lighting equation [5] (Fig. 8a). The point size is determined during vertex shading, which allows to set the size of the point primitive per particle.

4.2.3 Hatching

Hatching highlights curved areas while shading a surface by gradual variation of the hatch stroke density. We present an approach that traces hatches as a polyline over the surface along either a fixed direction or a smoothed principal curvature direction.

Input: The particle positions are passed on to the geometry shader threads, as the seed point for a hatch trace.

Processing: The hatches are represented by line-strip primitives, which can be precomputed since the approach is not view-dependent. The geometry shader thread determines the vertices connecting the hatch segments by projecting the direction to the tangent plane.

Output: The resulting line-strip primitives are recorded.

In a next rendering pass, the line strips can be fetched from the texture buffer and rendered in real time. The appearance of the hatches can be adjusted interactively.

The *curvature-based* approach (Fig. 6b) improves the way hatches convey object shape by guiding the hatches into the direction of the principal curvature on the implicit surface.

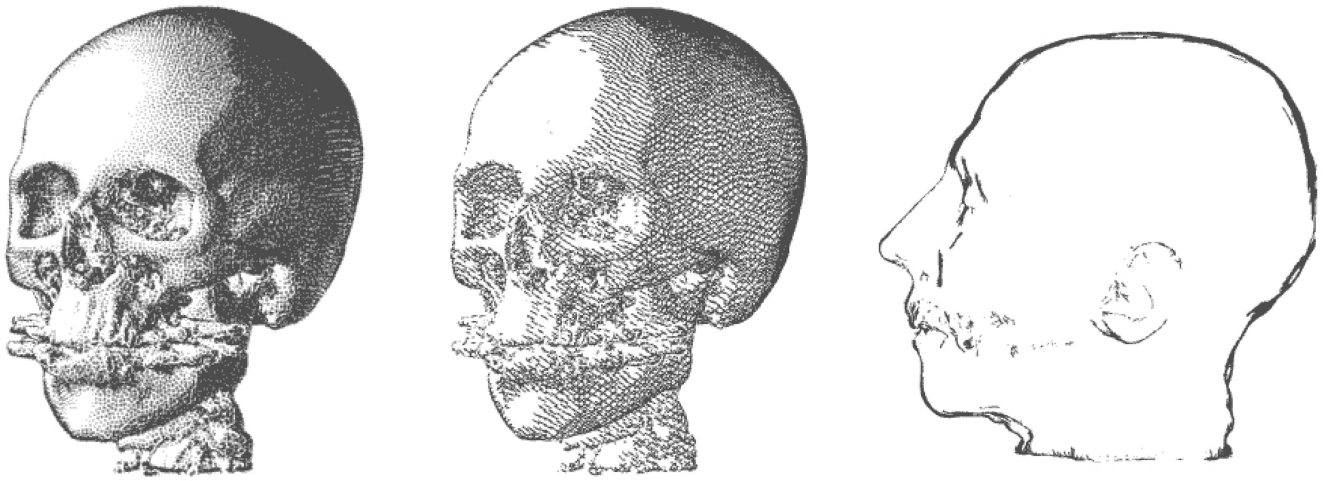
Sigg and Hadwiger [27] presented a fast cubic B-spline filtering approach to reconstruct partial derivatives from the volume data. These derivatives are used to compute the principal curvature directions in real time.

Curvature information is computed on demand while tracing the hatches. However, directly tracing along the principal curvature directions yields to messy results when the main direction is not robustly defined (Fig. 7a). The field of principal curvature directions should, therefore, be smoothed.

Smoothing of the curvature directions comprises computing a weighted average $\vec{s}_i$ based on the main principle curvature directions $\vec{k}_{1_j}$ of the particles p_j in the neighborhood. This smoothing is executed on the GPU estimating curvature information of neighboring particles on the fly.

$$\vec{s}_i = w_T \frac{\sum \rho_j \vec{k}_{1_j}}{\sum \rho_j} + (1 - w_T)\vec{s}_T, \text{ where } w_T = \frac{\sum \rho_j}{\sum_j 1}.$$

The trace reliability weight w_T is determined by averaging the surface reliability ρ_j of all neighboring particles. This



(a) (b) (c)

Fig. 8. Various illustrative visualization styles. (a) Stippling allows surface shading by changing the stipple scale. (b) Hatching also conveys shape by tracing hatch stroke in one or two directions. (c) Contours highlight the feature boundaries. The CT head data set is the courtesy of the University of North Carolina, Chapel Hill.

surface reliability determines whether the local main curvature direction is suitable for hatching. If it is suitable, the curvature directions will be weighted with reliabilities ρ_j; otherwise, the hatch is guided along a user-defined fixed direction $\vec{s}_T$. The surface reliability ρ is determined as:

$$\rho(\kappa_1, \kappa_2) = \begin{cases} 0, & \text{if } |\kappa_1| < \epsilon \text{ and } |\kappa_2| < \epsilon, \\ 1 - \left|2\left(|s| - \frac{1}{2}\right)\right|, & \text{otherwise.} \end{cases}$$

Here, κ_1 and κ_2 are the principal curvature magnitudes, while s is the shape index indicating the shape of the local surface. The ϵ parameter defines which nearly flat surfaces are considered flat. The shape index $s \in [-1, 1]$ was introduced by Koenderink and van Doorn [28], and is defined as:

$$s = \frac{2}{\pi} arctan \frac{\kappa_2 + \kappa_1}{\kappa_2 - \kappa_1} \quad (\kappa_1 \geq \kappa_2 \text{ and } |\kappa_1| + |\kappa_2| > 0).$$

The hatch strokes may now be traced along the smoothed curvature field. We use a simple weighting scheme, tracing the hatches based on the smoothed curvature directions and the number of segments from the seed point. This approach might lead to intersections for long hatch strokes.

Observe that in Fig. 6a, the hatches are traced nearly vertically along the surface, while the curvature-based hatches in Fig. 6b indeed follow the smoothed field. Especially, consider the strokes on the surface of the ears of the bunny.

Both hatching approaches are extended with cross-hatching functionality. A second hatch stroke is generated at each particle position, departing under a user-defined angle from the original hatch stroke. The increase of hatch density results in a darker tone. Using a two-level threshold on the basic diffuse lighting equation, three tones can be used to shade the surface. The brightest areas contain no hatches, intermediately illuminated areas are hatches with single strokes, and the darkest areas are cross-hatched (Fig. 8b).

4.2.4 Contours

Contours are known for their ability to convey object shape by emphasizing object boundaries (Fig. 8c). The contours of an object are defined by the set of lines, demarcating areas where the objects surface turns away from the viewer.

The contours are generated starting from particle positions near the contours, similar to the creation of the hatch strokes. In contrast, the contours cannot be generated prior to rendering, since they are by definition view-dependent.

Particles within a user-defined distance from the contours are selected and segments are traced along the contours by a geometry shader. Particles near the contour are now considered to be point primitives, transformed by the geometry shader into line strips that resemble a part of the contour. In contrast to the hatches, the line strips are not recorded to buffer, but directly rendered to screen.

The VolumeFlies framework [5] presents constant-width contours by placing a threshold on a curvature-dependent measure τ. Both the trace direction and the measure for constant-width contours were adopted.

4.2.5 Curvature-Based Transfer Functions

Curvature information was used to guide the direction of the hatches. Besides that curvature information can be employed to enhance surface detail.

To that end, Kindlmann et al. [9] proposed curvature-based transfer functions for illustrative rendering. They apply two-dimensional transfer functions to the space of principal curvature magnitudes (κ_1, κ_2). For example, valleys and ridges can be emphasized.

For our purpose, curvature-based transfer functions are engaged to color the hatch strokes. Valleys and ridges of the isosurface are highlighted through the colored hatches, where valleys are darkened and ridges are lightened.

This coloring scheme requires curvature information to be available at each position along the hatch. Since curvature estimation is computationally expensive, there is a trade-off between processing and memory usage.

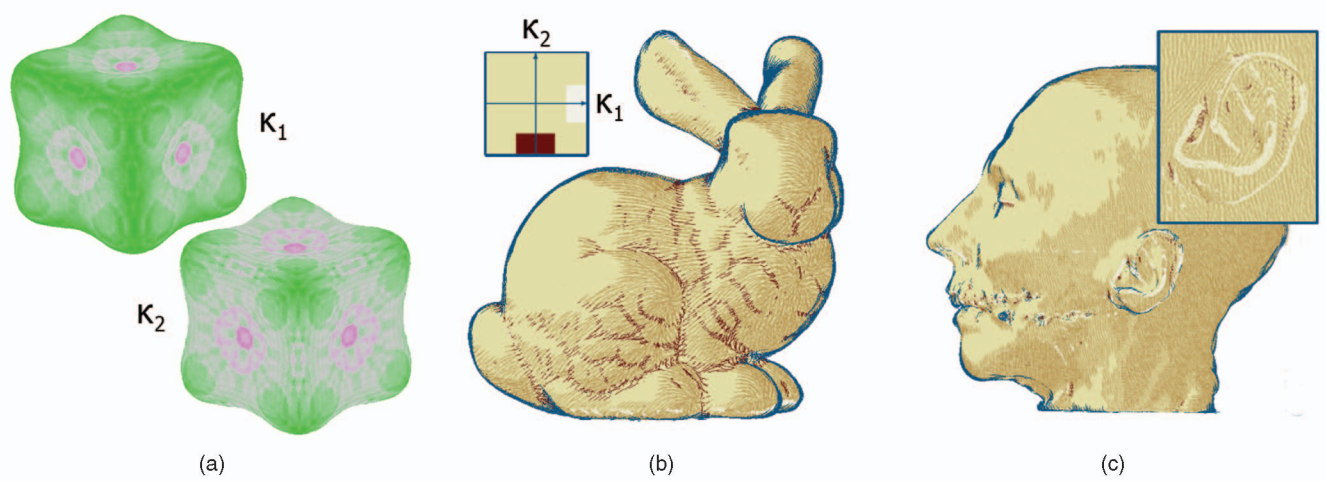


Fig. 9. Various illustrative styles with curvature-based coloring. (a) 1D transfer function on principal curvature magnitudes (κ_1, κ_2) on a splatted surface; in comparison to the work by Kindlmann et al. [9]. (b) Cone-splatted isosurface, with direction-based hatches emphasizing on ridges and valleys using a 2D transfer function on the principal curvature magnitudes (κ_1, κ_2). (c) Same as b, with detail on the ridges and valleys of the human ear.

Precomputing principal curvature on an arbitrary point on the isosurface requires memory for at least the main principal curvature direction with magnitude together with the magnitude of the orthogonal direction. Memory usage grows rapidly if more points are generated, for instance, when hatch strokes are traced more densely. Consequently, a real-time approach is preferred.

We previously mentioned that the real-time curvature method by Sigg and Hadwiger [27] was adopted to estimate principal curvature. To validate the approach, a one-dimensional transfer function was implemented on the curvature magnitudes κ_1 and κ_2. The resulting images using an identical transfer function and synthetic volume as presented by Kindlmann et al. [9] is depicted in Fig. 9a. Whereas Kindlmann et al. oversample the volume to represent the color coding, we depict the color coding on a coarser cone-splatted isosurface.

Subsequently, a two-dimensional transfer function is mapped to the space of curvature magnitudes (κ_1, κ_2), emphasizing ridges and valleys. Fig. 9b depicts a hatched isosurface together with the applied transfer function. Fig. 9c shows a similar approach where a detail of the ear clearly expresses the contrast between valleys and ridges.

4.2.6 Combined Styles and Multisurface Rendering

Illustrative styles on an isosurface can easily be combined. This is demonstrated in Fig. 1, where the splatted isosurface is rendered without shading. Black contours are enabled and directional hatches are applied in combination with a faint scale-based stippling.

Different isosurfaces reveal diverse information contained in the volume data. Multiple particle sets can be employed to visualize various isosurfaces. Fig. 10 shows three examples of multisurface illustrations. Note that visibility of the surfaces within these renderings strongly benefits from the sparseness of the pen-and-ink-style visualization approaches.

In order to create a multisurface rendering, particles are created for each selected isosurface in the volume during the feature-location step. After redistribution, cone-splatting is applied to remove particles that reside on hidden-surfaces without occluding particles from particle sets bound to other isosurfaces. Last, an illustrative style can be applied to each of the particle sets.

Illustrative rendering is particularly useful for context visualization. For instance, Fig. 10a shows a cone-splatted surface rendering of the skull where the context is defined by a cross-hatched rendering of the skin with added contours. In medical applications, anomalies are typically the focus of attention. These anomalies could be depicted more realistically by means of direct volume rendering while the anatomical context is presented illustratively. Such an anatomical frame of reference will appeal to surgeons, since they require proper understanding of the positioning of the considered anomaly.

5 Results

We have shown a particle-based illustrative volume-rendering framework for which we employed our GPGPU paradigm. Various illustrative renderings are presented in Figs. 9 and 10. The framework was implemented in C++ using the OpenGL 3D graphics API in combination with the GL shading language (GLSL). All algorithms are entirely GPU-driven; supporting NVidia 8 series and upwards.

5.1 Performance Comparison

The following performance comparison illustrates the significantly increased interactivity of the hardware-rendered framework, as compared to the software-rendered framework. The actual performance gain for each of the operations is presented in Table 2.

From these results, we can conclude that all elements of the framework show a major performance gain. In particular, the steps without interparticle communication show a striking increase in speed. Computation times of the preprocessing steps are decreased significantly. For example, the particle placement now allows interactive change of isovalues. Also the particle redistribution step shows a large performance gain for which we believe no GPU-based solution was available.

The visualization of the illustrative styles requires frame-to-frame processing. Here, we achieve interactive framerates

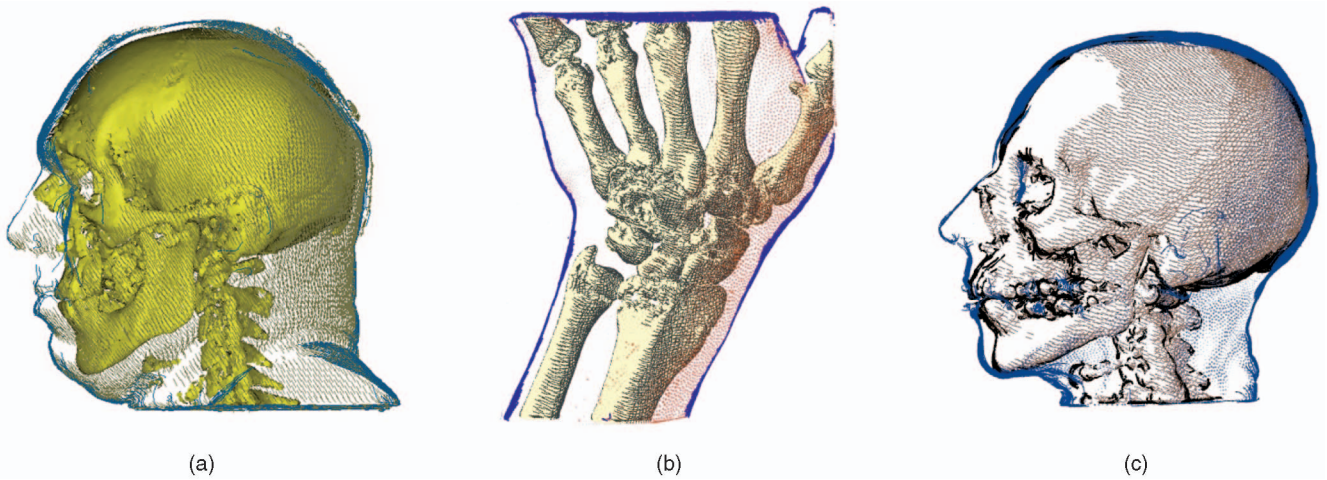


Fig. 10. Various illustrative styles with multisurface rendering. (a) Cone-splatted bone isosurface with diffuse lighting combined with direction-based hatching and contours on skin isosurface. (Data set equal to Fig. 5.) (b) Direction-based hatching on cone-splatted bone tissue combined with a scale-based stippled skin isosurface with contours. (c) Similar as b, with added contours on the bone surface. (Data set equal to Figs. 8 and 9c.)

as well. Volumes can be inspected in real time applying several styles to multiple isosurfaces, while changing associated parameters.

5.2 Performance Scalability

The VolFliesGPU framework has virtually no dependence on the resolution of the render window. Therefore, the window size can be scaled with minimal loss of performance. This is attained by discarding fragment shading while vertex or geometry shaders carry out time-consuming algorithms massively parallel.

Conversely, the framework performance is directly related to the amount of particles in the system. The performance scalability can, therefore, be measured by varying the amount of particles used for a particular illustrative rendering. To that end, the spacing of the proxy geometry is adapted during the feature location stage. A larger spacing results in a larger sampling distance within the volume texture, and therefore, generates fewer particles.

The most serious performance bottleneck in the software-rendered framework was the particle redistribution. In a first experiment, the duration of the redistribution is measured for particle sets of increasing size. The results are depicted in Fig. 11a. The amount of particles cannot be determined a priori, since this amount directly depends on the isosurface structure. Therefore, Fig. 11a is undersampled for larger particle set. For particle sets up to 100,000 particles, the redistribution time stays well under five seconds. For the majority of the renderings with on average 60,000 particles, this is a satisfactory result. Although the redistribution can not yet be executed in real time, this is a significant improvement.

In a second experiment, the framerate is measured for particle sets of increasing size. A curvature-based cross-hatching style is applied with a threshold on basic diffuse lighting. For this experiment, hidden-surface removal is disabled. This way, all particles will be taken into account during the measurement instead of only the visible particles.

The results for the second experiment are depicted in Fig. 11b. Rendering of the hatches remains interactive for particle sets up to 100,000 particles. The framerate decrease is approximately exponential. Take into account that the hatches typically have a length of six segments, which requires additional geometry to be rendered for each added particle. Finally, Fig. 11b shows that the framerate becomes unacceptably low when rendering over 200,000 particles. This is primarily a limitation of the available memory of the graphics card, which can be resolved by incorporating an elaborate memory management. Currently, the dimensions and quantification of the volume and supporting buffers are limited to the amount of GPU memory available.

5.3 Presented Figures

Several figures, based on different data sets, were presented throughout the paper. Various parameterizations of the illustrative styles were applied to both synthetic volumes and CT acquired data. The performance results for the presented figures are given in Table 3.

The illustrated torso, depicted in Fig. 1, exemplifies that volume dimensions and quantification hardly harm the overall performance of the framework. The volume consists of 512^3 voxels with 16-bits precision, which is the current limit without additional memory management. Nevertheless,

TABLE 2
Performance Comparison

General information:			
Processor	Intel Core 2 Duo 2.4 GHz; 3GB RAM		
Graphics hardware	NVidia GeForce 8800GTX		
Dataset	CT Head (256^3 voxels x 8 bits)		
Number of particles	60.000		
Initialization:	CPU	GPU	speed-up
Load volume data	3.92 sec	0.14 sec	28x
Brute-force particle placement	9.83 sec	0.14 sec	70x
Redistribution (25 steps)	253.34 sec	4.00 sec	63x
Redistribution (15 steps)	*	2.58 sec	-
Visualization:	CPU	GPU	speed-up
Stippling (Scale-based)	7 fps	1135 fps	162x
Hatch smooth field directions	7.73 sec	1.16 sec	6x
Hatch generation (Direction-based)	52.65 sec	0.07 sec	752x
Hatch generation (Curvature-based)	53.05 sec	0.29 sec	183x
Hatch visualization	1 fps	18 fps	18x
Contours	$<$ 1 fps	15 fps	324x

* Uses stop criterion, which is not included in software-rendered VolumeFlies

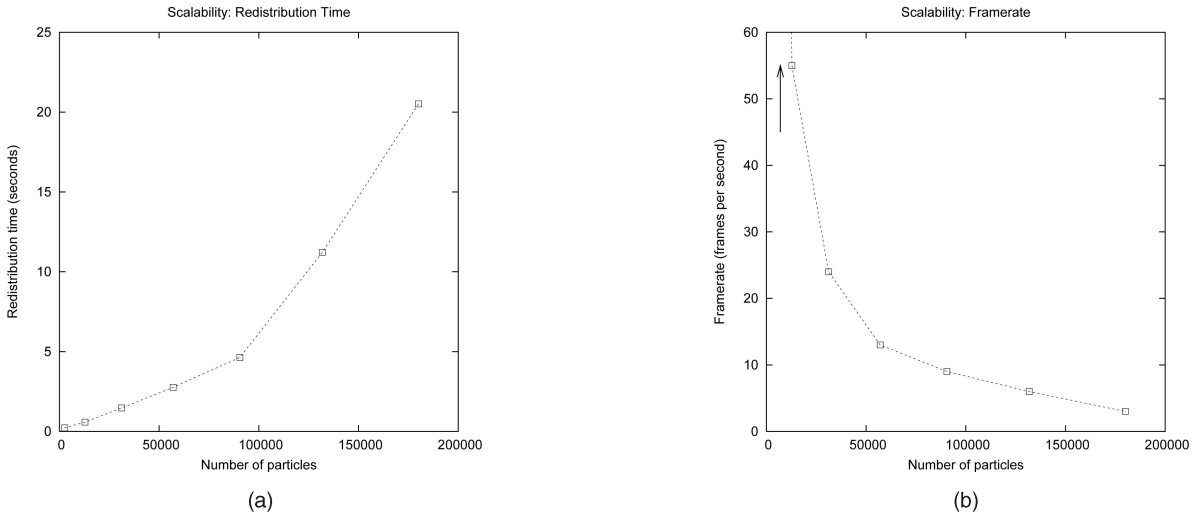


Fig. 11. Perfomance scalability when increasing the amount of particles.

interactive framerates are achieved while several illustrative styles are applied simultaneously.

The main pen-and-ink styles were depicted in Fig. 8. Satisfactory results for a customary CT data set can be achieved with less than 60,000 particles. All renderings perform at interactive framerates. The contours are more demanding since they are view-dependent.

Fig. 6 showed the difference between direction-based hatches and smoothed curvature-based hatches. The performance results for the rendering of both hatch approaches are identical, while generating the smooth curvature-based hatches requires additional preprocessing time. The time required to generate these hatches typically takes up to two seconds for an average sized particle set. For Fig. 7, a similar reasoning holds where the smoothing requires additional preprocessing time.

Figs. 9b and 9c presented colored hatches using a curvature-based transfer function. For these figures, a dense hatching approach is applied to emphasize the effect of the curvature-based transfer function. Generally, a less dense hatching suffices resulting in better framerates. Note that curvature is estimated on a frame by frame basis for all visible points along the hatches. For instance, the rendering presented in Fig. 9b achieves a framerate of five frames per second, while it performs over 15 frames per second when contours are disabled.

Last, Fig. 10 depicted various multisurface renderings. Since particle sets are bounds to surfaces, the amount of particles will grow rapidly when adding isosurfaces. For most acquired data sets, however, no more than three meaningful isosurface can be extracted. Even when a surface is densely sampled and splatted, as presented in Fig. 10a, interactive framerates are achieved. When adding more illustrative styles and denser illustrative approaches, interactivity will be lost as presented by Fig. 10c.

At present, the results of the presented techniques have not been subject to a thorough evaluation study. However, medical illustrators have responded positively to the illustrative renderings. In particular, the interactively adjustable parameters, such as the isovalue, were received enthusiastically. In their comparison to hand-made drawings, the illustrative renderings were perceived slightly more rectilinear and artificial. However, the general positioning of the stipples, the directions of the hatch strokes, as well as the lighting were said to be in accordance to manual illustrations.

TABLE 3
Performance of the GPU-Based Framework

Figure	Resolution	#Particles	Framerate
Torso (fig. 1)	512^3 (16 bits)	20324	15.1 FPS
Bunny (fig. 4)	128^3 (16 bits)	25354	104.2 FPS
Male (fig. 5)	256^3 (8 bits)	11594	242.4 FPS
Skull a (fig. 8a)	256^3 (8 bits)	57030	51.3 FPS
Skull b (fig. 8b)	256^3 (8 bits)	26808	29.6 FPS
Skull c (fig. 8c)	256^3 (8 bits)	57030	11.3 FPS
Bunny (fig. 6)	128^3 (16 bits)	27420	45.1 FPS
Horse (fig. 7)	128^3 (16 bits)	18492	63.6 FPS
Bunny (fig. 9b)	128^3 (16 bits)	29732	5.1 FPS
Head (fig. 9c)	256^3 (8 bits)	89439	2.9 FPS
Male (fig. 10a)	256^3 (8 bits)	70154	9.9 FPS
Hand (fig. 10b)	256^3 (8 bits)	41122	12.6 FPS
Head (fig. 10c)	256^3 (8 bits)	75668	3.6 FPS

6 Discussion

The GPU currently implies memory limitations. The volume and intermediate buffers should fit in GPU memory. Rendering of larger data could be investigated incorporating more advanced memory management.

The amount of particles should be restricted because it strongly influences performance of the algorithms. Large screen resolutions do not affect interactivity. At present, the particle density does not scale with the zoom factor.

The presented framework is flexible and extensible with new styles and techniques. Incorporating the single operator for particle visibility determination by Katz et al. [26], might increase hidden-surface removal accuracy and performance.

Particles are sparsely distributed which makes them suitable for context visualizations (Fig. 10c). It would be

valuable to combine direct volume-rendering with the presented indirect illustrative methods for focus-and-context rendering. Regions of interest can be realistically rendered, and will be the focus of attention while the illustrative styles provide the context.

With the presented techniques at hand, it is worthwhile to evaluate the effectiveness of illustrative rendering. Initial informal feedback by medical illustrators on the presented styles is positive. In the future, we intend to execute a more elaborate evaluation of the perception and applicability of the illustrative renderings.

7 Conclusions

We have presented an interactive particle-based illustrative volume-rendering framework based on a GPGPU paradigm. Below, we describe the conclusions based on the list of contributions, given at the end of Section 1.

First, a novel GPGPU paradigm allows data parallel execution of both the particle system and algorithms in the framework. In this paper, we have demonstrated searching and sorting algorithms implemented upon our GPGPU paradigm. The flexible GPU-based particle system can be used in different types of applications. Apart from visualization related applications, fast and generic GPU-based particle systems are valuable to various simulation applications, for instance, for fluid dynamics, weather phenomena, and earthquakes.

Using the GPGPU paradigm, we present a particle redistribution scheme which has been adapted to benefit from the parallel processing capacity of the GPU. To the best of our knowledge, this redistribution scheme had not yet been realized on a GPU basis.

Various illustrative styles that resemble pen-and-ink drawings can be applied interactively to isosurfaces in volumetric data. Density- and scale-based approaches were presented to apply stippling on an isosurface. Similarly, two hatching approaches were presented, namely a direction-based and a curvature-based approach. The hatch strokes are either colored uniformly or by a curvature-based transfer function emphasizing ridges and valleys. In addition, we presented object-space contours. Using these illustrative styles, multiple isosurfaces can be inspected simultaneously in real time and visualization parameters can be adjusted easily.

Last, we present an elaborate performance analysis. The performance comparison shows the substantial performance improvements, in particular, in the preprocessing steps of the framework. This comparison provides an intuition on the difference in interactivity of the systems. Furthermore, we describe the performance scalability of the framework based on an increasing amount of particles.

References

[1] A. Baer, C. Tietjen, R. Bade, and B. Preim, "Hardware-Accelerated Stippling of Surfaces Derived from Medical Volume Data," *Proc. Eurographics/IEEE-VGTC Symp. Visualization (EuroVis),* pp. 235-242, 2007.

[2] S. Bruckner and M.E. Gröller, "Style Transfer Functions for Illustrative Volume Rendering," *Computer Graphics Forum,* vol. 26, no. 3, pp. 715-724, http://www.cg.tuwien.ac.at/research/publications/2007/bruckner-2007-STF/, 2007.

[3] E.B. Lum and K.-L. Ma, "Hardware-Accelerated Parallel Non-Photorealistic Volume Rendering," *Proc. Second Int'l Symp. Non-Photorealistic Animation and Rendering (NPAR '02),* pp. 67-ff, 2002.

[4] M. Hadwiger, C. Berger, and H. Hauser, "High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware," *Proc. IEEE Visualization Conf. (VIS '03),* pp. 301-308, 2003.

[5] S. Busking, A. Vilanova, and J. van Wijk, "Particle-Based Non-Photorealistic Volume Visualization," *Visual Computer,* vol. 24, no. 5, pp. 335-346, May 2007.

[6] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: A GPU-Based Particle Engine," *Proc. ACM SIGGRAPH/Eurographics Conf. Graphics Hardware (HWWS '04),* pp. 115-122, 2004.

[7] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, "A Particle System for Interactive Visualization of 3D Flows," *IEEE Trans. Visualization and Computer Graphics,* vol. 11, no. 6, pp. 744-756, Nov. 2005.

[8] M. Meyer, P. Georgel, and R. Whitaker, "Robust Particle Systems for Curvature Dependent Sampling of Implicit Surfaces," *Proc. Int'l Conf. Shape Modeling and Applications,* pp. 124-133, 2005.

[9] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller, "Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications," *Proc. IEEE Visualization Conf.,* pp. 513-520, Oct. 2003.

[10] I. Viola, S. Bruckner, M.C. Sousa, D. Ebert, and C. Correa, "IEEE Visualization Tutorial on Illustrative Display and Interaction in Visualization," http://vis.computer.org/vis2007/session/tutorials.html, 2007.

[11] A. Secord, "Weighted Voronoi Stippling," *Proc. Second Int'l Symp. Non-Photorealistic Animation and Rendering (NPAR '02),* pp. 37-43, citeseer.ist.psu.edu/secord02weighted.html, 2002.

[12] A. Lu, C. Morris, J. Taylor, D. Ebert, C. Hansen, P. Rheingans, and M. Hartner, "Illustrative Interactive Stipple Rendering," *IEEE Trans. Visualization and Computer Graphics,* vol. 9, no. 2, pp. 127-138, Apr. 2003.

[13] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein, "Real-Time Hatching," *Proc. ACM SIGGRAPH '01* , E. Fiume, ed., pp. 579-584, citeseer.ist.psu.edu/article/praun01realtime.html, 2001.

[14] Z. Nagy, J. Schneider, and R. Westermann, "Interactive Volume Illustration," *Proc. Vision, Modeling, and Visualization Workshop '02,* citeseer.ist.psu.edu/nagy02interactive.html, 2002.

[15] M. Burns, J. Klawe, S. Rusinkiewicz, A. Finkelstein, and D. DeCarlo, "Line Drawings from Volume Data," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 512-518, Aug. 2005.

[16] X. Xie, Y. He, F. Tian, and H.-S. Seah, "An Effective Illustrative Visualization Framework Based on Photic Extremum Lines," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 6, pp. 1328-1335, Nov. 2007.

[17] X. Yuan and B. Chen, "Illustrating Surfaces in Volume," *Proc. Joint IEEE/Eurographics Symp. Visualization (VisSym '04),* O. Deussen, C.D. Hansen, D.A. Keim, and D. Saupe, eds. pp. 9-16, 337, http://dtc.umn.edu/xyuan/research/publication/isv.htm, 2004.

[18] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, no. 1, pp. 80-113, http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x, 2007.

[19] D. Göddeke, "GPGPU-Basic Math Tutorial," technical report, FB Math., Univ. Dortmund, Nov. 2005.

[20] P. Kipfer and R. Westermann, "Improved GPU Sorting," *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation,* M. Pharr, ed., pp. 733-746, Addison-Wesley, 2005.

[21] J.S. Venetillo and W.C. Filho, "GPU-Based Particle Simulation with Inter-Collisions," *Visual Computer,* vol. 23, nos. 9-11, pp. 851-860, http://dblp.uni-trier.de/db/journals/vc/vc23.html#VenetilloF07, 2007.

[22] S. Drone, "Real-Time Particle Systems on the GPU in Dynamic Environments," *Proc. ACM SIGGRAPH '07 Courses,* pp. 80-96, 2007.

[23] R. van Pelt, A. Vilanova, and H. van de Wetering, "GPU-Based Particle Systems for Illustrative Volume Rendering," *Proc. IEEE/EG Symp. Volume and Point-Based Graphics,* H.-C. Hege, D. Laidlaw, R. Pajarola, and O. Staadt, eds., pp. 89-96, http://www.eg.org/EG/DL/WS/VG/VG-PBG08/089-096.pdf, 2008.

[24] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Proc. ACM SIGGRAPH '87,* pp. 163-169, 1987.
[25] M. Meyer, R.M. Kirby, and R. Whitaker, "Topology, Accuracy, and Quality of Isosurface Meshes Using Dynamic Particles," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 6, pp. 1704-1711, Nov.-Dec. 2007.
[26] S. Katz, A. Tal, and R. Basri, "Direct Visibility of Point Sets," *Proc. ACM SIGGRAPH '07 Papers,* p. 24, 2007.
[27] C. Sigg and M. Hadwiger, "Fast Third-Order Texture Filtering," *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation,* M. Pharr, ed., pp. 313-329, Addison-Wesley, 2005.
[28] J.J. Koenderink and A.J. van Doorn, "Surface Shape and Curvature Scales," *Image Vision Computations,* vol. 10, no. 8, pp. 557-565, 1992.

Roy van Pelt received the MSc degree in computer science and engineering in 2007 from Eindhoven University of Technology. He is a PhD candidate in the Department of Biomedical Image Analysis at Eindhoven University of Technology. His current research interests include medical visualization, volume visualization, and medical image analysis.

Anna Vilanova received the PhD degree in 2001 from the Vienna University of Technology. She is an assistant professor in the Biomedical Image Analysis section of the Biomedical Engineering Department at the Eindhoven University of Technology. Her research interests include medical visualization, volume visualization, and medical image analysis. She is a member of the IEEE Computer Society.

Huub van de Wetering received the PhD degree in 1991 from Eindhoven University of Technology. He is an assistant professor in the visualization group of the Department of Mathematics and Computer Science at the same university. His research interests include information visualization and computer graphics.