IEEE/ EG Symposium on Volume and Point-Based Graphics (2008)
H.- C. Hege, D. Laidlaw, R. Pajarola, O. Staadt (Editors)

GPU-based Particle Systems for Illustrative Volume Rendering

R.F.P. van Pelt[1], A. Vilanova [1] and H.M.M. van de Wetering[2]

[1]Department of Biomedical Image Analysis, Eindhoven University of Technology, The Netherlands
[2]Department of Visualization, Eindhoven University of Technology, The Netherlands

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 user-configurable particle systems to produce stylized renderings from the volume data, imitating traditional pen-and-ink drawings. In the following, we present an interactive GPU-based illustrative framework, called VolFlies-GPU, for rendering volume data, 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.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Hardware Architecture]: Parallel processing I.3.6 [Computer Graphics]: Graphics data structures and data types

1. Introduction

There exist various volume rendering techniques that produce images from 3D volumetric data sets. Typical examples of 3D 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 was to generate images that closely resemble reality. However, a new volume rendering branch investigates ways to create illustrative images from 3D 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 on important features, improving data exploration. Futile details are omitted and important aspects are highlighted, resulting in more comprehensible images [BTBP07, BG07].

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 [LM02, HBH03].

We have adopted the illustrative concepts of the VolumeFlies framework, presented by Busking et al. [BVvW07]. This framework offers a general basis to produce illustrative depictions from volumetric data sets. A variety of illustrative styles can be directly applied, based on particle systems that operate on the volume data. Currently included styles imitate traditional pen-and-ink drawing techniques.

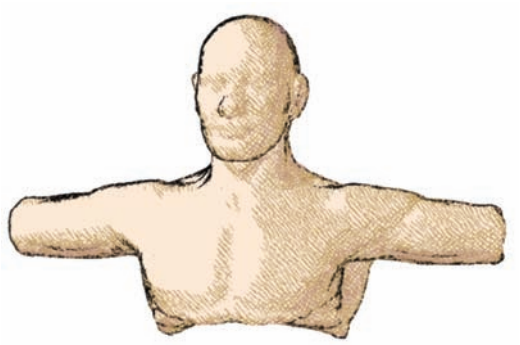


Figure 1: VolFliesGPU: illustrative styles on a voxelized torso. The torso model is provided courtesy of Mangon and Dretakkis by the AIM@SHAPE Shape Repository.

We have chosen the flexible particle-based approach of the VolumeFlies framework, expecting a considerable performance gain. GPU-based particle systems are able to process and visualize hundreds of thousands of particles in real-time [KSW04, KKKW05]. 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 [BVvW07] have been transformed to fit 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. [MGW05] (Section 4.1).
- A real-time illustrative volume rendering framework, initiating particle systems to create stylized depictions (Section 4.2). Styles resembling pen-and-ink illustration techniques, known from VolumeFlies [BVvW07], can now be applied to volume features interactively. Most algorithms were elaborately transformed to use GPU parallelism, achieving fast interaction and parametrization.

2. Previous work

A particular extensive field of research investigates illustrative visualization [VBS*07]. We strive for an interactive framework, offering a variety of illustrative styles. Hence we are mainly concerned with hardware-rendered approaches that produce pen-and-ink style renderings from volume data.

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 et al. [Sec02], 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. [BTBP07]. Furthermore, there are *object-based* approaches. Lu et al. [LMT*03] presented an interactive approach, controlling the stipple density on a voxel basis.

Another traditional shading style is called hatching, producing tone variations by means of combined stroke patterns. The hatches convey surface shape by means of 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. [PHWF01]. Furthermore there are *object-based* approaches, that generate the actual hatch stroke geometry, e.g., Nagy et al. [NSW02].

Most illustrative techniques emphasize on 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 approach was presented by Burns et al. [BKR*05]. A method based on 'photic extremum lines', which detects changes in luminance, was presented by Xie et al. [XHTS07]

A complete framework for illustrating surfaces in volume data was presented by Yuan et al. [YC04]. Another framework was presented by Busking et al. [BVvW07]. Their particle-based approach is flexible and configurable and allows to apply all previously mentioned pen-and-ink styles.

The GPU is often being used for mathematical computations [OLG*07, GÖ5], even though the hardware is geared towards graphics processing. A variety of algorithms that can be executed data parallel, such as searching and sorting [KW05], typically show a substantial performance gain. The increasing interest in GPGPU is supported by the new software platform, named Compute Unified Device Architecture (CUDA). CUDA allows developers to execute algorithms using the GPU, without knowledge of the underlying hardware architecture. For our illustrative framework we have decided to directly employ the graphics hardware for both general computations and rendering algorithm.

In general, 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. Typically, the behavior of the particles is affected by rules from dynamics, resulting in a particle flow [KSW04, KKKW05, VF07, Dro07]. 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. [MGW05]. They present a 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 with features in volumetric data. 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 view on future work.

3. GPGPU paradigm using transform feedback

The common GPGPU approach involves rendering a window-size quad, gathering input values from a 2D texture and performing computations on a fragment basis [GÖ5]. Output values are returned through a render-to-texture operation. Although this approach offers a solid solution for many algorithms [KW05], it is a rather counterintuitive 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.

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

Table 1: GPGPU relations

The actual paradigm, depicted in figure 2, heavily relies on the recently introduced *Unified Instruction Set Architecture*. Programming the *Unified Shader Model* or *Shader Model 4.0* allows for more flexible use of the graphics hardware, and takes over the task of load-balancing from the developer.

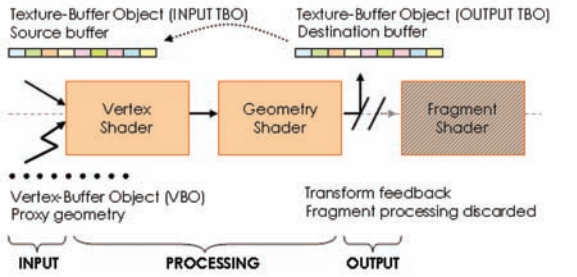


Figure 2: GPGPU paradigm using transform feedback.

Our paradigm implies an intensive use of both *vertex shaders* and *geometry shaders*, while *fragment shaders* are completely discarded. This stands in contrast to the commonly applied render-to-texture approaches.

Input: The input side requires a *buffer object* containing 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 by means of a *vertex buffer object* (VBO). The vertex positions encode indices for elements in 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.

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. Be aware that his 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. As a result all fragment processing can be discarded.

Computations often require the return of multiple outputs, in which 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 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 figure 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 to create an interactive illustrative volume rendering framework, called VolFliesGPU.

4. The VolFliesGPU framework

The VolFliesGPU framework comprises an interactive 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. [BVvW07], and is schematically depicted in figure 3. Each of the framework modules are parallelized, for which we have employed our GPGPU paradigm.

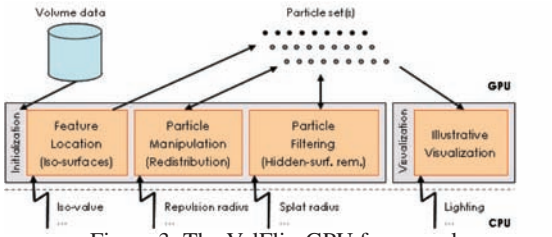


Figure 3: The VolFliesGPU framework.

4.1. Initializing the particle system

Feature location

Initially the framework simply places a set of particles near a feature in the volume. For this paper a feature is an iso-surface at a user selected iso-value, and the initialization samples the volume data at a user defined grid. In a marching cubes like approach (Lorensen and Cline [LC87]), a particle is created between the sample positions and its neighboring grid point if the iso-surface lies inbetween, 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** iso-surface is crossed **then**
4: output particle between grid points *n* and *r*

Output: Initial Particle Set

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 iso-surface. 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.

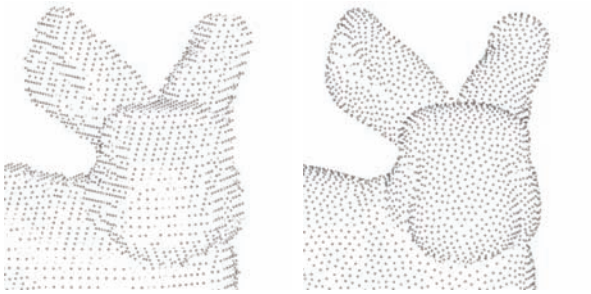
The geometry shader thread creates an output array by constructing a line-strip of at most three vertices. Each vertex encodes the object-space position of a new particle.

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).

Redistribution

The initial particle placement results in particles on a rectilinear grid, as depicted in figure 4a. A redistribution step moves the particles towards the actual iso-surface location, evenly spreading them over the surface. This comprises an energy minimization scheme, similar to the work presented by Meyer et al. [MGW05]. 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 inter-particle communications, because neighbor interactions counteract parallel processing of the particles.



(a) Before redistribution (b) After redistribution

Figure 4: Particles redistribute evenly over the surface by exerting repulsive forces to nearby particles. The bunny model is the courtesy of Stanford University.

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 repulsion radius. We propose a four-step iterative redistribution scheme: (I) Sort the particles, (II) create a bin look-up table, (III) minimize energy, and (IV) verify if the system is stable.

I) The particles will be sorted, with the bin number as sorting key. Sorting has been applied to particle systems for depth ordering and inter-particle collision detection [KKKW05]. GPU-based sorting [KW05] is typically data-independent, exploiting computational parallelism. We have adopted the *odd-even merge sort* algorithm by Kipfer et al. [KW05].

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 look-up table. We create such a look-up 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 look-up 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. [MGW05].

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

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

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

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 iso-surface (Step 2)

Output: Updated particles with lowered energy state

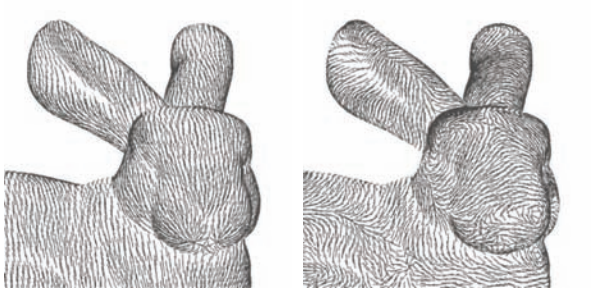


Figure 5: Hatches trace a single direction over the surface (left), or smoothed principal curvature directions (right). The bunny model is the courtesy of Stanford University.

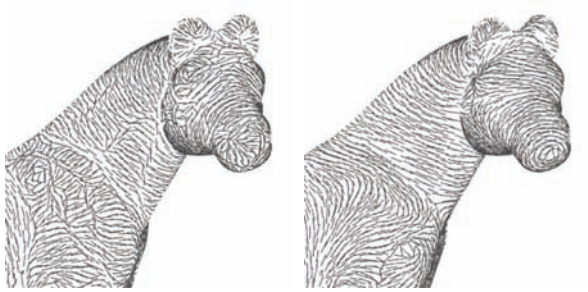


Figure 6: Hatches follow the main principal curvature directions (left), or the smoothed directions (right). The horse model is the courtesy of Georgia Institute of Technology.

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 pair-wise 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 a 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 point-based stippling techniques (figure 7a), stroke-based hatching techniques (figure 7b) and contour visualization (figure 7c).

For all styles, the VolumeFlies [BVvW07] hidden surface removal filter is applied: The iso-surface is splatted with uniquely colored cones, generated by the geometry shader, and particle visibility is determined by an off-screen framebuffer.

Stippling

Stippling is a technique where points are used to convey object shape. Therefore we render our particles 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 [BVvW07] (figure 7a). The point size is determined during vertex shading, which allows to set the size of the point primitive per particle.

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 pre-computed 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 subsequent rendering pass, the line-strips can be fetched from the texture buffer and rendered in real-time. The appearance of the hatches can now be adjusted interactively.

The *curvature-based* approach (figure 5b) 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 [SH05] presented a fast cubic B-spline filtering approach in order 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 (figure 6b). The field of principal curvature directions should therefore be smoothed.

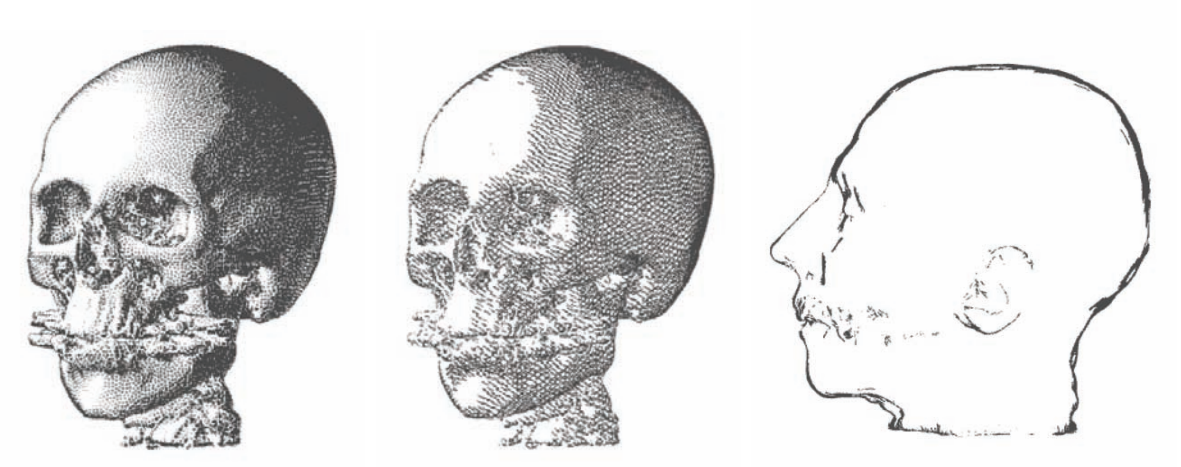


Figure 7: Various illustrative visualization styles. Stippling allows surface shading by changing the stipple scale (left). Hatching also conveys shape by tracing hatch stroke in one or two (middle) directions. Contours (right) highlight the feature boundaries. The CT head data set is the courtesy of University of North Carolina, Chapel Hill.

Smoothing of the curvature directions comprises computing a weighted average $\vec{s}_i$, based on the main principle curvature directions $\vec{\kappa}_{1_j}$ 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 surface reliability determines whether the local main curvature direction is suitable for hatching. If it is suitable for hatching 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:

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

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 [KvD92], 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 figure 5a the hatches are traced nearly vertical along the surface, while the curvature-based hatches in figure 5b 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 (figure 7b).

Contours

Contours are known for their ability to convey object shape by emphasizing object boundaries (figure 7c). 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 a texture buffer, but directly rendered to screen.

The original VolumeFlies framework [BVvW07] 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.

Combined styles and context visualization

The illustrative styles can be combined, while keeping interactive framerates. Moreover, multiple particle-set can be created, visualizing different styles on multiple iso-surfaces. Adding particle sets comes with a performance cost, which eventually leads to loss of interactivity. Approximately

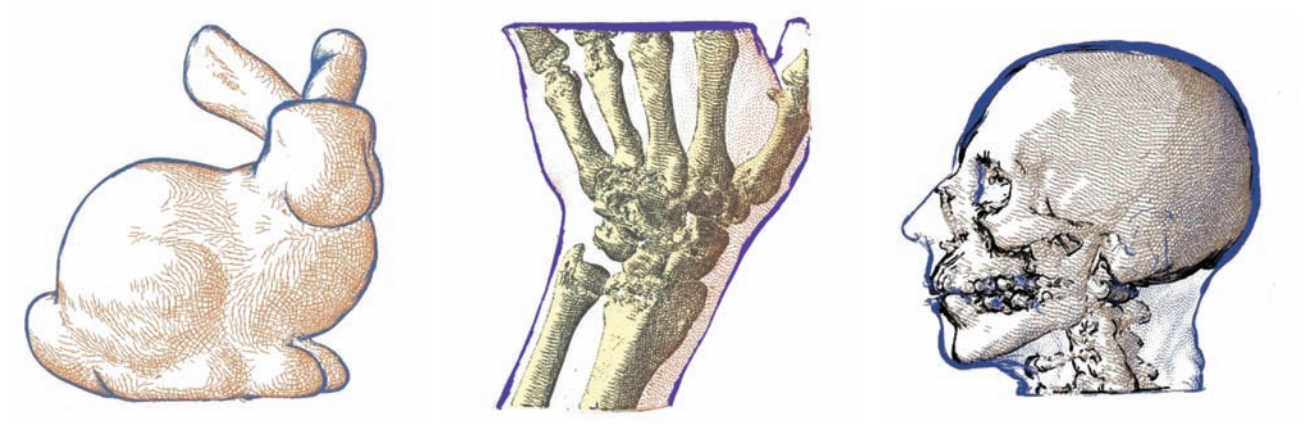


Figure 8: Various illustrative styles. Curvature-based hatching with contours (left). Direction-based hatching on cone-splatted bone tissue (middle), combined with a scale-based stippled skin iso-surface with contours. Similar styles, with added contours on the bone surface (right).

200.000 particles can be rendered interactively, which in practice is sufficient to illustrate the desired features.

Figure 1 demonstrates a splatted iso-surface without shading. Black contours are enabled and directional hatches are applied in combination with a faint scale-based stippling.

In contrast to figure 1, figure 8c includes two particle sets, visualizing two iso-surfaces. This demonstrates that illustrative rendering is particularly useful for context visualization, originating from the sparseness of elements.

5. Results

We have shown a particle-based illustrative volume rendering framework for which we employed our GPGPU paradigm. Various illustrative depictions are presented in figure 8. 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, currently bound to the NVidia 8 series.

The GPU was brought forward to increase performance of general purpose computations. In table 2, we show the actual performance gain for each of the operations performed by the hardware-rendered framework, in comparison with the software-rendered framework.

From these results we can conclude that all elements of the framework show a big performance gain. In particular the steps without inter-particle communication show a striking increase of speed. Computation times of the pre-processing steps are decreased significantly. For example the particle placement now allows real-time change of iso values. Also the particle redistribution step shows a big performance gain, for which we believe no GPU-based solution was available.

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

Table 2: Performance comparison

The visualization of the illustrative styles requires frame-to-frame processing. Also here we achieve interactive framerates. Volumes can be inspected in real-time, while applying multiple styles and changing associated parameters.

Table 3 shows the performance of the figures presented throughout this paper. The performance of an illustrative rendering depends heavily on the amount of particles in the system, while the volume dimensions and quantification hardly harm the overall performance. The amount of particles in the system depends on the surface area of the selected iso-surface, yet the amount can be changed interactively by changing the spacing of the sampling grid. The dimensions and quantification of the volume is limited to the amount of memory available on the GPU.

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
Skull a (fig. 7a)	256^3 (8 bits)	57030	51.3 FPS
Skull b (fig. 7b)	256^3 (8 bits)	26808	29.6 FPS
Skull c (fig. 7c)	256^3 (8 bits)	57030	11.3 FPS
Bunny (fig. 5)	128^3 (16 bits)	27420	45.1 FPS
Horse (fig. 6)	128^3 (16 bits)	18492	63.6 FPS
Bunny (fig. 8a)	128^3 (16 bits)	17242	14.0 FPS
Hand (fig. 8b)	256^3 (8 bits)	41122	12.6 FPS
Head (fig. 8c)	256^3 (8 bits)	75668	3.6 FPS

Table 3: Performance of the GPU-based framework

6. Conclusions and Future work

We have presented an interactive particle-based illustrative volume rendering framework, based on a GPGPU paradigm. The paradigm allows data parallel execution of both the particle-system and algorithms in the framework.

The flexible and generic GPU-based particle system can be used in different types of applications. We presented a particle redistribution scheme, which to the best of our knowledge was not yet realized on a GPU basis.

Illustrative styles that resemble pen-and-ink drawings can be applied interactively to iso-surfaces in volumetric data. Different iso-surfaces can be inspected in real-time, and visualization parameters can be adjusted easily. Furthermore, the performance of the pre-processing steps is improved.

The GPU currently implies memory limitations. The volume and intermediate buffers should fit in GPU memory. In the future, rendering larger data could be investigated.

The amount of particles should be restricted, because it strongly influences performance of the algorithms. Large screen resolutions do not affect interactivity. Currently, 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. [KTB07], might increase hidden surface removal performance.

Particles are sparsely distributed, which makes them suitable for context visualizations (figure 8c). We are interested in combining direct volume rendering with the presented indirect illustrative methods for focus-and-context rendering.

References

[BG07] BRUCKNER S., GRÖLLER M. E.: Style transfer functions for illustrative volume rendering. *Computer Graphics Forum 26*, 3 (2007), 715–724.

[BKR*05] BURNS M., KLAWE J., RUSINKIEWICZ S., FINKELSTEIN A., DECARLO D.: Line drawings from volume data. *ACM Transactions on Graphics (Proc. SIGGRAPH) 24*, 3 (Aug. 2005), 512–518.

[BTBP07] BAER A., TIETJEN C., BADE R., PREIM B.: Hardware-accelerated stippling of surfaces derived from medical volume data. In *EuroVis* (2007), pp. 235–242.

[BVvW07] BUSKING S., VILANOVA A., VAN WIJK J.: Particle-based non-photorealistic volume visualization. *The Visual Computer Journal* (2007).

[Dro07] DRONE S.: Real-time particle systems on the gpu in dynamic environments. In *ACM SIGGRAPH '07: courses* (2007), ACM, pp. 80–96.

[GÖ5] GÖDDEKE D.: *GPGPU–Basic Math Tutorial*. Tech. rep., FB Mathematik, Univ. Dortmund, Nov. 2005.

[HBH03] HADWIGER M., BERGER C., HAUSER H.: High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *VIS '03: Proceedings* (2003), IEEE Computer Society, p. 40.

[KKKW05] KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3d flows. *IEEE TVCG 11*, 6 (2005), 744.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: a gpu-based particle engine. In *HWWS '04 proceedings* (2004), ACM, pp. 115–122.

[KTB07] KATZ S., TAL A., BASRI R.: Direct visibility of point sets. In *SIGGRAPH '07 papers* (2007), p. 24.

[KvD92] KOENDERINK J. J., VAN DOORN A. J.: Surface shape and curvature scales. *Image Vision Computations 10*, 8 (1992), 557–565.

[KW05] KIPFER P., WESTERMANN R.: Improved GPU sorting. In *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Pharr M., (Ed.), pp. 733–746.

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

[LM02] LUM E. B., MA K.-L.: Hardware-accelerated parallel non-photorealistic volume rendering. In *NPAR '02: Proceedings* (2002), ACM, pp. 67–ff.

[LMT*03] LU A., MORRIS C., TAYLOR J., EBERT D., HANSEN C., RHEINGANS P., HARTNER M.: Illustrative interactive stipple rendering. *IEEE TVCG 9*, 2 (2003), 127–138.

[MGW05] MEYER M., GEORGEL P., WHITAKER R.: Robust particle systems for curvature dependent sampling of implicit surfaces. In *Proceedings of the Int. Conference on Shape Modeling and Appl.* (June 2005), pp. 124–133.

[NSW02] NAGY Z., SCHNEIDER J., WESTERMANN R.: Interactive volume illustration. In *Proceedings of Vision, Modeling and Visualization Workshop '02* (2002).

[OLG*07] OWENS J., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A., PURCELL T.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[PHWF01] PRAUN E., HOPPE H., WEBB M., FINKELSTEIN A.: Real-time hatching. In *SIGGRAPH '01 proceedings* (2001), Fiume E., (Ed.), pp. 579–584.

[Sec02] SECORD A.: Weighted voronoi stippling. In *NPAR '02: Proceedings* (2002), pp. 37–43.

[SH05] SIGG C., HADWIGER M.: Fast third-order texture filtering. In *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Pharr M., (Ed.), pp. 313–329.

[VBS*07] VIOLA I., BRUCKNER S., SOUSA M. C., EBERT D., CORREA C.: IEEE visualization tutorial on illustrative display and interaction in visualization, 2007.

[VF07] VENETILLO J. S., FILHO W. C.: Gpu-based particle simulation with inter-collisions. *The Visual Computer 23*, 9-11 (2007), 851–860.

[XHTS07] XIE X., HE Y., TIAN F., SEAH H.-S.: An effective illustrative visualization framework based on photic extremum lines. *IEEE TVCG 13*, 6 (2007), 1328.

[YC04] YUAN X., CHEN B.: Illustrating surfaces in volume. In *VisSym* (2004), pp. 9–16, 337.