Technical Report

Real-Time Particle Level Sets with Application to Flow Visualization

Nicolas Cuntz[1] **and Robert Strzodka**[2] **and Andreas Kolb**[1]
[1]**University of Siegen, Germany**
[2]**Stanford University, Max Planck Center**

Version: May, 15th, 2007
(update: January, 17th, 2008)

Abstract

Level set methods have evolved as powerful tools for the representation and evolution of free surfaces. While the implicit grid based representation deals very well with topological changes, high resolutions and high order integration schemes are required to preserve fine surface details. The particle level set (PLS) method is an important extension which additionally employs particles to correct the numerical dissipation in the grid. This allows to reduce the spatial resolution and the order of the integration, while preserving a time consistent evolution.

While grid-based numerical schemes and particle systems on their own have been efficiently mapped to GPUs, the coupling of these methods is still a challenging task. GPU applications in graphics and visualization continue to utilize particle or grid representations exclusively. This paper presents an enhanced variant of the PLS approach which fully maps to the GPU. Improvements w.r.t. the original PLS technique include a sub-voxel interface representation and a more accurate level set correction using more precise particle radii. Our method achieves both, higher performance *and* superior quality in terms of mass preservation compared to a public CPU-based reference implementation.

As a concrete application we demonstrate that our fast and accurate PLS is well suited for the visualization of dynamic flows. In particular in 3D, where higher dimensional objects (in contrast to points) are tracked, an accurate evolution of time surfaces and representation of path volumes offer a more reliable basis for data interpretation.

ACM Categories: I.3.5 Computer Graphics (Computational Geometry and Object Modeling - Curve, surface, solid, and object representations)

1. Introduction

The level set method, introduced by [OS88], is an important tool in a growing number of areas where the simulation of a moving interface, i.e. a surface between two participating media, plays a key role, e.g. fluid mechanics, computer vision, material science and computer graphics [OF02].

The main idea of the level set method is to represent a surface implicitly as iso-contour of a signed distance function. This interface easily captures changes in the surface topology during its time evolution. But the Euler-based advection of level sets suffers from numerical diffusion, resulting in a mass loss in areas of high curvature. High resolution grids and high order HJ-(W)ENO advection schemes in conjunction with a small time step can be used to improve the accuracy, yet this significantly increases the computational complexity.

The *particle level set (PLS)* method is a hybrid grid/particle method introduced by [EMF02] in order to reduce the numerical diffusion. Marker particles placed near the interface are used to correct the level set representation. Since the particles track the underlying flow characteristics more accurately, the order of the level set advection can be reduced (see [ELF04] and the open source library [MF06]). PLS seeks an ideal balance of both, local accuracy and the number of particles and thus performance.

The PLS method has been used in complex flow visualizations and animations, including special effects in major motion pictures. Despite the high usefulness of the method no fine-grained parallel implementation have been demonstrated.

Traditional concepts of flow visualization are based on *explicit* object representations . Line-based techniques have been expanded to surface and volume animations in order to better convey the flow behavior in 3D [CSM00]. Tracking the generating vertices (or particles) is very accurate, but for inhomogeneous and divergent flow a refinement is required to ensure a proper representation of the underlying continuous flow objects. This leads to an exponentially growing number of particles and outliers, not representative for the surface any more. *Implicit* flow visualization techniques map the flow field to a scalar function and apply volume rendering techniques [vW93].

Contribution: The focus of this paper lies on a fine-grained parallel PLS implementation on the GPU. This requires a modification of the PLS algorithm in order to make best benefit of the parallel capabilities of GPUs. Additionally, due to a precise sub-voxel description of the interface using distance transforms, we achieve a more accurate result. Combining vertex/fragment buffer objects, shaders and blending functionality in an innovative way leads to an extremely compact information exchange between the particles and grids. We trace a manageable number of particles in real-time and show as much detail as possible, while ensuring a minimal average quality. The resulting GPU-based PLS implementation outperforms the public PLS library [MF06], making it ready to be used in real-time applications.

We demonstrate our parallel PLS method in real-time flow visualization. Our implementation includes complex time surfaces and path volumes. These flow primitives have been tested with highly inhomogeneous and divergent flows.

In this paper we are concerned with an efficient parallel grid/particle coupling. For simplicity the level set func-

tion is represented on the entire domain, not just in a narrow band around the interface. Consequently, the examples are restricted to 128^3 to reduce the effect of the full grid. Clearly, for better efficiency on large domains an adaptive representation (cf. [LKHW04]) is desirable.

The remainder of this paper is structured as follows. Sec. 2 discusses related research topics. The classical PLS method is summarized in Sec. 3. Our GPU-based approach is described in Sec. 4. The application of the parallel PLS method to flow visualization is given in Sec. 5. The result section 6 provides performances and accuracy statistics as well as a comparison of our method to the PLS library [MF06].

2. Related Work

In this section we briefly describe GPU-based level set methods for the representation of free surfaces (Sec. 2.1), flow visualization techniques using explicit and implicit flow geometries (Sec. 2.2), and distance fields and distance transforms (DTs), which are major building blocks for our enhanced and GPU-based PLS technique (Sec. 2.3). The original PLS technique is sketched in Section 3.

2.1. GPU-based Level Set Methods

A first GPU based implementation of the level set equation was presented by Rumpf and Strzodka [RS01]. It uses intensity and gradient based forces to drive a segmentation of 2D images. Lefohn et al. [LKHW04] additionally incorporated a curvature term and an adaptive memory model for the narrow band technique and thus could run efficient segmentation in 3D. In these applications, the numerical dissipation was not a problem, on the contrary, a smooth boundary of the segmented 3D region is desirable and the curvature term is used exactly for this purpose. An iterative solution to the level set equation was presented by Griesser et al. [GRNG05].

2.2. Surfaces and Volumes in Flows

Stream (or flow) volumes, the volumetric equivalent to stream lines, have been introduced by Max et al. [MBC93]. Here, an *explicit* representation of the volume based on tetrahedra is used. Silva et al. [SHK97] do not visualize the volume but rather animate the motion of an arbitrary smooth surface in the flow. Instead of explicit quadrilateral or triangular patches Brill et al. [BHR*94] use stream balls to represent the stream surface. When multiple stream balls are near each other they form a closed single surface, when they diverge, the surface splits. Krüger et al. [KKKW05] show a GPU-based implementation of particle based techniques like stream lines and stream ribbons.

Wijk [vW93] generates *implicit stream surfaces* by looking at level sets of a specific scalar function. The scalar function is obtained from the flow field by back-tracking each 3D-position to an inflow boundary point. Flow visualization results from rendering the level sets of this scalar function. Westermann et al. [WJE00] extend this approach to precomputed arrival times of a particle at each position. Xue et al. [XZC04] introduce a variation and postponing of the assignment of scalar values and also the allowance of other termination surfaces than the inflow boundaries in the back-tracking process. The use of volume rendering and texture mapping techniques can interactively visualize the implicit flow volumes with different choices of the values. The generation of streak volumes with the level set method is discussed by Weiskopf [Wei04]. The level set formulation allows to retain a sharp interface of the volume, whereas the simple advection of a density function (dye, smoke) smears out the density quickly.

2.3. Distance Fields and Transforms

The reinitialization of the level set function requires the construction of a Euclidean 3D Distance Field. Distance Field computation is a well studied problem (see [Cui99] for an overview). Consider a voxel grid and a closed object boundary $\delta\Omega$ in the voxel grid. The Euclidean Distance Field dm represents the minimum distance of a voxel $\mathbf{x}$ to $\delta\Omega$, i.e.

$$\mathrm{dm}(\mathbf{x}) = \min_{\mathbf{x}' \in \delta\Omega} \{\|\mathbf{x} - \mathbf{x}'\|\}.$$

The Euclidian distance transform dt in addition stores the closest reference point on the boundary:

$$\mathrm{dt}(\mathbf{x}) = \left(\min_{\mathbf{x}' \in \delta\Omega} \{\|\mathbf{x} - \mathbf{x}'\|\}, \arg\min_{\mathbf{x}' \in \delta\Omega} \{\|\mathbf{x} - \mathbf{x}'\|\} \right).$$

Depending on the initial object representation, using a voxel grid or an explicit geometric representation, different approaches have been proposed.

Concerning the voxel grid approach, there are two major categories, *propagation methods* and methods based on *Voronoi diagrams*. Propagation methods iteratively propagate the distance information to the neighboring voxels, either by spacial sweeping or by contour propagation.

Different parallel approaches for the computation of a distance transform for a set of sites have been proposed, for 2D pixel sites [ST04, RT06] and for 3D polygonal input data [SPG03, SGGM06]. For our PLS framework, a fast computation of a 3D distance transform is required. The

jump-flooding approach from Rong et al. [RT06] can be extended to 3D. An alternative hierarchical propagation method has been proposed by Cuntz et al. [CK07], computing an approximate distance transform (see Sec. 4.2). In comparison to the previously mentioned works, the main benefits of this approach are a fully hierarchical design and the possibility to balance speed and accuracy.

3. Particle Level Set Method

This section is a résumé of the PLS method based on [ELF04] and serves as starting point for our GPU-based PLS approach.

The key idea of the level set method is the representation of the typically lower dimensional interface I in a domain D by the iso-contour $I(\phi) := \{x \in D | \phi(x) = 0\}$ of the level set function $\phi : D \rightarrow \mathbb{R}$. The motion of the interface is performed by evolving ϕ within a velocity field.

Typically, ϕ is initialized to be a signed distance field. However, after advecting ϕ with different velocities, this property is lost, and ϕ needs to be reinitialized, i.e. the distance field is re-computed ideally not changing the location of the zero level-set $I(\phi)$.

Enright et al. [ELF04] use a fast first order accurate semi-Lagrangian method to evolve ϕ. Without particle corrections, this leads to high inaccuracies after multiple time steps, because the committed errors quickly accumulate, e.g. resulting in a mass loss (see left image in Fig. 8).

The PLS method aims to prevent the numerical diffusion caused by the advection, utilizing a particle tracing approach. Two sets of particles are placed near to the interface $I(\phi)$. Positive particles are located in the $\phi > 0$ region and negative particles in the $\phi < 0$ region. A particle p at position $\mathbf{x}_p$ has a radius r_p defined as:

$$r_p = \begin{cases} r_{max} & \text{if } s_p\phi(\mathbf{x}_p) > r_{\max} \\ s_p\phi(\mathbf{x}_p) & \text{if } r_{\min} \leq s_p\phi(\mathbf{x}_p) \leq r_{\max} \\ r_{min} & \text{if } s_p\phi(\mathbf{x}_p) < r_{\min} \end{cases}$$

where s_p is the sign of the particle, i.e. $s_p = \mathrm{sgn}(\phi(\mathbf{x}_p))$. Thus, the circle around $\mathbf{x}_p$ with radius r_p touches the interface, if $|\phi(\mathbf{x}_p)| \in [r_{\min}, r_{\max}]$.

Particle tracking is much more accurate than the grid-based advection of the level set, especially if a high order Runge-Kutta integration is used. Thus, it is reasonable to rely on the particles to correct the interface representation. The correction step involves the definition of a temporary level set function ϕ_p around each particle:

$$\phi_p(\mathbf{x}) = s_p(r_p - \|\mathbf{x} - \mathbf{x}_p\|). \tag{1}$$

After level set and particle advection, *escaped* particles, i.e. those that are further away than their radius on the wrong side of the interface, are used for the level set correction. Each escaped particle p contributes to the eight surrounding grid voxels through intermediate level set functions ϕ^+ and ϕ^- that are initialized to ϕ and updated according to the formulas

$$\begin{aligned} \phi^+(\mathbf{x}) &\leftarrow \max(\phi_p(\mathbf{x}), \phi^+(\mathbf{x})), \\ \phi^-(\mathbf{x}) &\leftarrow \min(\phi_p(\mathbf{x}), \phi^-(\mathbf{x})) \end{aligned} \tag{2}$$

After processing of all escaped particles, the new (corrected) level set is constructed according to

$$\phi(\mathbf{x}) = \begin{cases} \phi^+(\mathbf{x}) & \text{if} \quad \|\phi^+(\mathbf{x})\| \leq \|\phi^-(\mathbf{x})\| \\ \phi^-(\mathbf{x}) & \text{else} \end{cases} \tag{3}$$

After the particle correction, ϕ is reinitialized in order to restore a signed distance function.

The PLS method is known to produce good results even when performing a first order semi-Lagrangian level set advection. The algorithm according to Enright et al. [ELF04] is summarized below. Note that the level set correction is performed two times.

Algorithm 1 *(PLS algorithm)*

1. *Definition of the interface location and velocity field*
2. *Initialization of the level set based on the interface*
3. *First order semi-Lagrangian level set advection*
4. *Second order Runge-Kutta particle advection*
5. *Correction of the level set function using the particles*
6. *Level set reinitialization*
7. *Correction of the level set function using the particles*
8. *Particle positioning and reinitialization*
9. *Go to 3*

4. GPU-based PLS Method

An analysis of the PLS algorithm reveals several challenges concerning an efficient implementation on the GPU, especially as we want to further enhance the accuracy of the method:

- Representing and processing of PLS data requires grid and particle data structures on the GPU and a two-way data exchange between them.
- The level set (re-)initialization requires a fast parallel method to compute distance fields or distance transforms.
- The level set correction requires a fast selection of escaped particles and an efficient construction of the intermediate level set functions ϕ^+ and ϕ^-.
- Sub-voxel accuracy is required to achieve less numerical diffusion and thus less mass loss.

These tasks are addressed in detail in the following sections. The first part covers necessary data structures and the corresponding GPU analogies and outlines the PLS data flow as it is realized in our GPU-based system. The remaining parts discuss the individual steps of the algorithm. For orientation we will refer to the steps in Alg. 1.

As the whole PLS algorithm runs completely on the GPU, we assume some familiarity with the graphics hardware and GPU programming in the following description of the implementation. For an introduction to GPU programming, see [Buc05, Har05].

4.1. Data Structures

The level set reinitialization algorithm used in our approach produces a distance transform, including the reference to the nearest point. This information is useful for the particle reinitialization (Alg. 1, step 8 and Sec. 4.4) and in addition to level set reinitialization at sub-voxel precision (Sec. 4.2). Finally, the DTs can efficiently be computed on GPUs using a very fast hierarchical method.

The DT is stored in a voxel grid Φ_{xyzd} with 4 components per voxel storing the actual level set value $\Phi_d(\mathbf{x}) = \phi(\mathbf{x})$ and a sub-voxel reference to the interface in $\Phi_{xyz}(\mathbf{x})$. The sub-voxel references are of interest not only for the level set reinitialization, but also for particle reinitialization (see Sec. 4.4). Note that in some stages of the algorithm, e.g. after the level set advection, Φ_{xyzd} may not represent a precise and/or complete distance transform.

On the technical side, we represent Φ_{xyzd} by a tiled 2D `GL_RGBA_FLOAT16_ATI` floating point texture and use frame buffer objects for render-to-texture functionality. A simple double buffering approach is used to separate input from output data in the data parallel GPU-processing. The 16 bit components of Φ_{xyzd} reduce the bandwidth requirements in contrast to full floats, are absolutely sufficient for the interface and DT representation, and in addition allow a faster execution of the level set correction stage through clever use of blending modes (see Sec. 4.7).

The velocity field $\vec{\mathbf{v}}$ can be given in any form and format that allows parallel point sampling. We use as a 3 component 16-bit texture with a tiled 2D representation.

Only for the particle system we want to use higher precision values, as the same particles are tracked for a considerable time in the velocity field, and we want to reduce the accumulation of errors in this process. The particle system is stored in a 2D 32-bit texture (`GL_RGBA32FLOAT_ARB`). In $\mathbf{P}_{xyz}(\mathbf{p})$ we store the position of the particle $\mathbf{p}$, and $\mathbf{P}_d(\mathbf{p})$ is a composed number containing a signed radius and a bit-flag determining whether the particle has to be used during the PLS correction (Alg. 1, step 5 and Sec. 4.7). Moreover, the sign of the radius determines whether the particle lies on the inner (negative) or outer (positive) side of the interface.

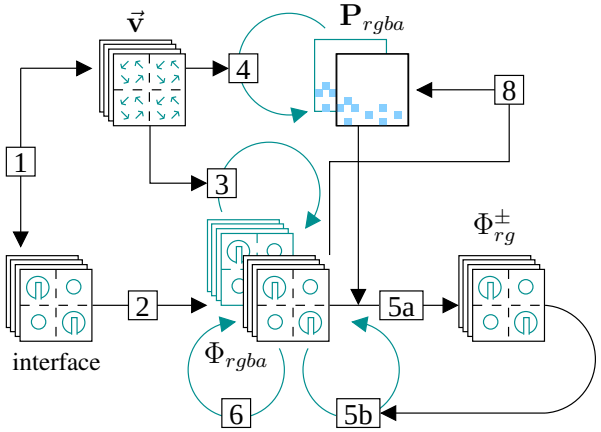

Figure 1: *The data flow in the GPU-based PLS framework – use of double buffering is marked by cyclic, blue arrows; numbers indicate steps in Alg. 1; step 7 is omitted in our GPU-based framework.*

The data flow with the corresponding steps of Alg. 1 is outlined in Fig. 1. In- and outgoing edges represent data input and output. Cyclic arrows involve double buffering, as input and output data has to be distinct on the GPU. The error correction (step 5) involves two passes and is split into two separate entities 5a and 5b.

4.2. Level Set Reinitialization

Enright et al. [ELF04] realize the reinitialization of the level set function via the fast marching method [Set99]. There is no feasible way to do this on the GPU. Using distance transforms, instead, allows us to achieve sub-voxel accuracy and enhances the particle reinitialization (Sec. 4.4).

Level set reinitialization is performed on Φ_{xyzd} at the beginning of the algorithm and later on the advected and corrected level set. In the initial state, Φ_{xyzd} contains the interface, which is given by means of the voxel classification as interior or exterior or, alternatively, by a gray-level voxel function, defining the interface on a sub-voxel level.

First, the interface needs to be identified, i.e. voxel close to the boundary are assigned suitable reference points on the interface. This is done in a fragment program which is executed for each voxel $\mathbf{x}$. $\Phi_{xyzd}(\mathbf{x})$ is updated if and only if there exists a direct neighbor voxel $\mathbf{y}$ with $\operatorname{sgn}(\Phi_d(\mathbf{x})) \neq \operatorname{sgn}(\Phi_d(\mathbf{y}))$. The update is done by calculating a sub-voxel position using linear interpolation:

$$\Phi_{xyz}(\mathbf{x}) \leftarrow \mathbf{x} + \frac{\|\Phi_d(\mathbf{x})\|}{\|\Phi_d(\mathbf{x})\| + \|\Phi_d(\mathbf{y})\|}(\mathbf{y} - \mathbf{x}). \quad (4)$$

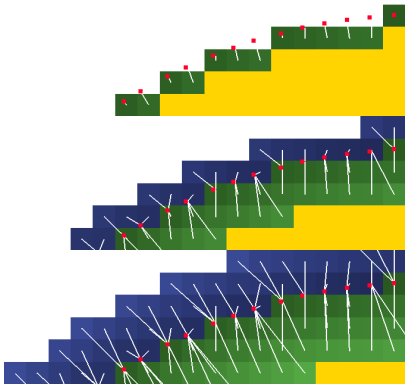

Figure 2: *Two propagation steps in a volume slice – Φ_{xyzd} is initialized with precise sub-voxel references. Wrong references, e.g. the crossing reference vectors in the third image, are rectified by a future propagation step.*

The level set value $\Phi_d(\mathbf{x})$ is set to the distance between $\mathbf{x}$ and $\Phi_{xyz}(\mathbf{x})$, multiplied by the sign previously stored at position $\mathbf{x}$. Note that the sub-voxel refinement performed in Eq. 4 is especially important after advecting a coarse grid. Using a simple binary interface representation disallows small time step for advection, since the changes might be too small to move the interface resulting in an unchanged interface. Additionally, the refinement can be used for a sub-voxel initialization of Φ_{xyzd}, if distance values are present for voxels directly next to the interface.

After identification of the interface, the complete distance transform Φ_{xyzd} is determined using the hierarchical approach proposed in [CK07], which is based on a propagation technique, both on a single hierarchy level and between the levels.

The propagation method can be implemented as a fragment program, where $\Phi_{xyzd}(\mathbf{x})$ is updated by computing alternative distances according to the reference points of neighboring voxels:

$$\Phi_{xyz}(\mathbf{x}) \leftarrow \arg \min_{\mathbf{x}' \in \mathcal{M}(\mathbf{x})} \{\|\Phi_{xyz}(\mathbf{x}') - \mathbf{x}\|\}$$

$$\Phi_d(\mathbf{x}) \leftarrow \min_{\mathbf{x}' \in \mathcal{M}(\mathbf{x})} \{\|\Phi_{xyz}(\mathbf{x}') - \mathbf{x}\|\}$$

Here, $\mathcal{M}$ is a structure element containing the direct neighborhood of $\mathbf{x}$. This algorithm, using a $3 \times 3 \times 3$ structure element, is known to produce the correct result if applied a sufficient often on all voxels in parallel (see [CK07] for details; Fig. 2 shows two sequential propagation steps).

4.3. Particle Positioning

For an efficient use of particles for the PLS correction (Alg. 1, step 5), all particles should be located near the interface (see Sec. 3). This, however, is difficult to achieve in parallel on the GPU, since one cannot iterate over the interface neighborhood and place particles accordingly. The positioning of particles close to the interface is not only necessary during the initialization in the beginning, but also occasionally during the execution of the algorithm, because more and more particles will drift away from the interface with an increasing number of time steps. Frequent repositioning has the negative side effect that inaccuracies in Φ_{xyzd} are constantly transferred into the new set of particles, thus annihilating the advantage gained by the particle correction. According to the literature [EMF02], a reasonable trade-off is to reposition the particles every 20 time steps on average, e.g. by repositioning 5% of all particles in each iteration.

A major advantage of the distance transform used for interface representation is the utilization of the references Φ_{xyz} for the particle reinitialization. First, the location $\mathbf{P}_{xyz}(\mathbf{p})$ for each particle p is chosen randomly within $[0,1]^3$. Afterward, the particle is pushed towards the interface by following the reference at the initial particle location $\Phi_{xyz}(\mathbf{P}_{xyz}(\mathbf{p}))$. The new position $\mathbf{P}_{xyz}(\mathbf{p})$ is computed according to:

$$\mathbf{P}_{xyz}(\mathbf{p}) \leftarrow \Phi_{xyz}(\mathbf{P}_{xyz}(\mathbf{p})) + \epsilon \cdot \hat{\mathbf{v}}_{\text{rand}}, \qquad (5)$$

where $\hat{\mathbf{v}}_{\text{rand}}$ is a normalized random vector pointing to an arbitrary direction and ϵ is a predefined small constant scalar. In our implementation, conforming with [ELF04], we choose ϵ such as to build a band of few cells around the interface. The random vector can be fetched from a texture with precomputed random values.

Pushing the particles towards the interface according to the above scheme can lead to sparsely populated regions, especially in areas with high curvature. In our experiments, this effect could always be sufficiently reduced by using a higher number of particles, but more efficient, even distributions of particles should be investigated in the future.

4.4. Particle Reinitialization

The radius of reinitialized particles should touch the interface (see Sec. 3). As the interface is represented by sub-voxel references in Φ_{xyz}, we fetch the nearest reference by reading Φ_{xyz} at the new particle position. Because the particle will, in general, not lie exactly on a grid voxel, neighborhood-sampling is used to fetch the optimal reference point:

$$\mathbf{P}_d(\mathbf{p}) \leftarrow \min_{\mathbf{x}' \in \mathcal{N}(\mathbf{P}_{xyz}(\mathbf{p}))} \{\|\Phi_{xyz}(\mathbf{x}') - \mathbf{P}_{xyz}(\mathbf{p})\|\}, \quad (6)$$

where $\mathbf{P}_{xyz}(\mathbf{p})$ is the new particle position resulting from Eq. 5 and $\mathcal{N}$ denotes the neighborhood consisting of the

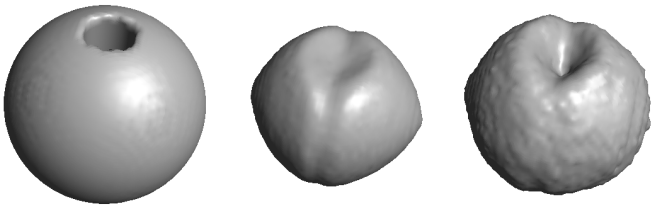

Figure 3: *Comparing particle radii using tri-linear distance interpolation and neighborhood-sampling of nearest reference point. Left: A notched sphere (first) is rotated 8 times by reseeding 5% particles in each frame using interpolated radii (second) and nearest reference (third).*

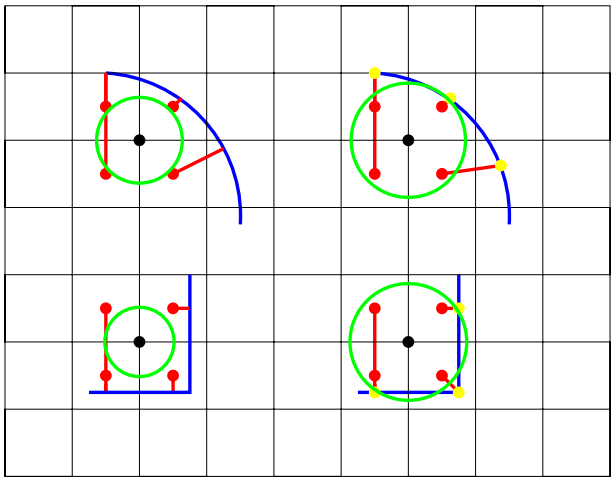

Figure 4: *More accurate distance computation with the neighborhood-sampling of reference points (right) than with tri-linear distance interpolation (left)*

eight voxels around a given location. The sign of the particle is determined by the sign Φ_d for the corresponding voxel. Note that there is no obvious way to obtain the sign on a sub-voxel level as it is the case for the radius.

$$\mathbf{P}_d(\mathbf{p}) \leftarrow \mathbf{P}_d(\mathbf{p}) \cdot \mathrm{sgn}(\Phi_d(\mathbf{P}_{xyz}(\mathbf{p}))).$$

At a first glance, the sampling method described in Eq. 6 appears too expensive compared to a simple tri-linear texture look-up of the distance component $\Phi_d(\mathbf{p})$. However, a closer observation of the mass loss reveals that the overall correction is more accurate when using the more precise distance based on the reference point $\Phi_{xyz}(\mathbf{x}')$. Fig. 3 illustrates the difference between the neighborhood-sampled reference-based radii computation and the tri-linear interpolation of the distances. Tri-linear distance interpolation leads to undesirable smoothing of important surface features (see Fig. 6 for quantitative results).

In Fig. 4, the grid points (red point) contain the length of the normal (red line) to the interface (blue). On the left, the distance at the particle position (black point) is computed with a bilinear interpolation of the grid point distances; it is simply an average in these examples. In case of a convex interface the resulting distance (green circle) is too small, because distances corresponding to very different normals are averaged. On the right, the distance is computed by finding the closest reference point (yellow point). Here, we also commit an error, as the selected reference point is not exactly the closest point on the interface to the particle (black point), but for highly convex interface parts, this is more accurate than the interpolation. For (almost) flat interface parts the interpolation is better because all normals point in (almost) the same direction.

In [ELF04], the radii of the particles are reset after level set reinitialization while [MF06] omit this step. The given reason for this is similar to the explanation why particle repositioning after each frame should be avoided: As particle radii have to be reset with information stored in the level set, inaccuracies of the level set are propagated into the particle model. Thus, it is reasonable to wait until the next particle repositioning before the radii are updated, and we follow this approach.

4.5. Level Set Advection

Level set advection (Alg. 1, step 3) is easily ported to the GPU due to its parallel nature. A velocity texture $\vec{\mathbf{v}}$ is read using tri-linear interpolation within a fragment program. New level set values are computed according to a semi-Lagrangian approach

$$\Phi_d(\mathbf{x}) \leftarrow \Phi_d(\mathbf{x} - \vec{\mathbf{v}}(\mathbf{x})\Delta t),$$

where Δt is the time step used for both level set advection and particle tracking.

4.6. Particle Tracking

In principle, particle tracking is performed according to the traditional PLS algorithm. A second order accurate Runge-Kutta integration is used to calculate the new positions in a fragment program.

In addition to the position, the sign and the radius, a flag has to be stored in the particle texture $\mathbf{P}_{xyzd}$, determining whether a particle $\mathbf{p}$ escapes the interface. In contrast to the original PLS approach (Sec. 3), we refine the particle correction by involving more particles in the level set correction. This is done by defining a particle as escaped if its radius is greater than its distance to the interface, i.e.

$$\mathbf{P}_d(\mathbf{p}) \overset{\text{bit}}{\leftarrow} \begin{cases} \text{true} & \text{if} \quad \|\mathbf{P}_d(\mathbf{p})\| > \|\Phi_d(\mathbf{P}_{xyz}(\mathbf{p}))\| \\ \text{false} & \text{else} \end{cases} \tag{7}$$

where $\overset{\text{bit}}{\leftarrow}$ stands for a bit-encoding operator which stores the boolean value into one bit of the float mantissa.

4.7. Level Set Correction

For level set correction (Alg. 1, step 5), we need a way to select particles marked as "escaped" and, based on this selection, to construct the intermediate level set functions ϕ^+, ϕ^-. To reduce bandwidth requirements we do this differently than in the original formulation, postponing the combination with ϕ until the final update.

The construction of ϕ^+, ϕ^- requires particle-to-grid coupling by scattering into the grid data structure. The key idea is to render a marker-geometry at the particle positions to trigger a computation for the 8 grid-voxel around the particle. To process all particles, the particle texture $\mathbf{P}_{xyzd}$ is copied into a Vertex Buffer Object. For the level set correction, all particles are sent through the graphics pipeline and a vertex program detects whether a particle $\mathbf{p}$ has escaped by checking a the encoded bit in $\mathbf{P}_d(\mathbf{p})$ (see Eq. 7). If the particle has not escaped it does not contribute to the level set correction and is discarded by moving it outside of the volume.

Point scattering approaches are known to be rather time-exhaustive, e.g. Kolb and Cuntz [KC05] could handle only a few thousand particles at interactive rates, using relatively large point sprites. Evaluating different types of marker-geometries, i.e. point sprites, quads and individual points, rendering four individual points into two subsequent slices of the grid is the most effective variant in our situation. Here, no explicit rasterization needs to be performed, since all relevant pixels, i.e. grid voxels, are addressed directly.

The intermediate level sets ϕ^+ and ϕ^- given in Eq. 1 are stored in a two-component voxel grid $\Phi^{\pm}_{xy} = (\phi^+, -\phi^-)$.

The accumulation of particle contributions in $\Phi^{\pm}_{xy}$ uses min-max blending, provided by the OpenGL extension `GL_EXT_blend_minmax`. This turned out to be a very efficient method to compute the max and min terms (see Eq. 2 in Sec. 6). Storing $-\phi^-$ instead of ϕ^- allows a single pass update of $\Phi^{\pm}_{xy}$ using maximum blending only. Initially, $\Phi^{\pm}_{xy}$ is set to $(-\infty, -\infty)$ for all voxels. Listing 1 in Appendix A shows an efficient way to implement the computation of $\Phi^{\pm}_{xy}$ in a fragment program.

A second pass rasterizes the complete level set, computing the corrected level set Φ_{xyzd} using $\Phi^{\pm}_{xy}$ according to Eq. 3, including the postponed combination with ϕ (see Listing 2 in Appendix A). Fig. 5 shows the steps involved in the error correction.

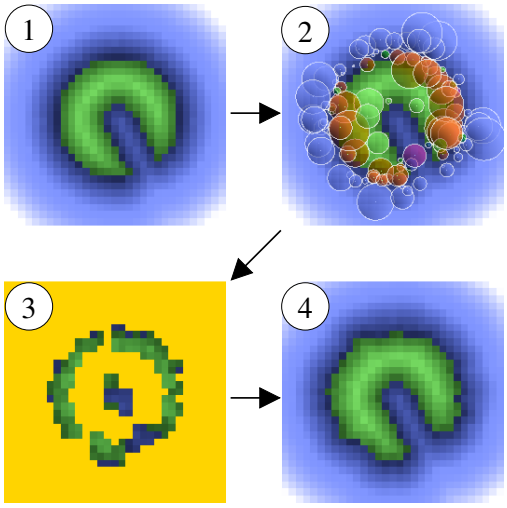

Figure 5: *Low resolution level set correction example with a small number of particles – (1) level set after advection by 30°, (2) the particle system (escaped particles are marked in magenta (positive) and orange (negative)), (3) the temporary level set containing ϕ^+ and ϕ^- (the component with highest magnitude is displayed), (4) the corrected level set. Note that the shrinking of the object due to inaccuracies in the advection is partially reverted. Usually, more particles are used in order to achieve better results after the correction.*

5. PLS-based Flow Visualization

Our parallel GPU-based PLS system offers an inherent balancing mechanism for accuracy and performance. Flow visualization can clearly benefit from the performance and the improved accuracy, allowing interactive navigation and parameters adjustment. The PLS technique is especially useful in case of complex flows. Where purely particle-based approaches generate sparse or overpopulated regions, PLS presents a more consistent picture as it does not rely on the particle distribution alone.

5.1. Settings

We concentrated on two types of entities: *Time surfaces* and *path volumes*. Time surfaces result from the evolution of arbitrary surfaces within a flow field. This is directly supported by the PLS method by representing the surface as a level set. Path volumes can be seen as the union of all path-lines starting at any point within an initial surface or volume. In the implicit volume representation, the interior is given by $\phi(\mathbf{x}) < 0$, thus path volumes can easily be determined by taking the minimum of the distance transforms for the evolving time surfaces. The result is a volume describing the path which has been traversed by the tracked shape. The minimum is stored in a separate voxel grid.

An example for time surfaces is shown in Fig. 9. It visualizes the internal structure of a complex typhoon flow by evolving a transparent shape. Further flow visualization examples are given in Fig. 7 and 10.

5.2. Rendering

The volume renderer is based on a back-to-front slicing technique using view-aligned polygons. Only fragments within a small iso-value range $[-\varepsilon, \varepsilon]$ around the interface are colored in a shader using Phong lighting. Applying alpha blending makes internal structures visible, which is important e.g. for complex time surfaces. This approach is similar to semi-transparent interval volume rendering [FMST96].

6. Results

This section evaluates the performance and the accuracy of our GPU-based PLS system on its own and in comparison to the public PLS library [MF06]. Additionally, various examples in the context of flow visualization are given.

The hardware used for testing is a PC with an AMD Athlon 64 X2 Dual Core Processor 4200+ (2.21 GHz), 4 GB RAM with a GeForce 8800 GTS graphics chip.

For performance evaluation, we use Zalesak's sphere (Fig. 8) used in previous work about PLS [ELF04], which is also available in the PLS library [MF06].

We compare our GPU method with the PLS library [MF06] using exactly the same Zalesak sphere and with equal parameters for both implementations. The test consists of 100 evolution steps in a vortex flow field to a total of 360°. Fig. 6 shows the resulting frame-rate and relative mass loss as function of the number of particles.

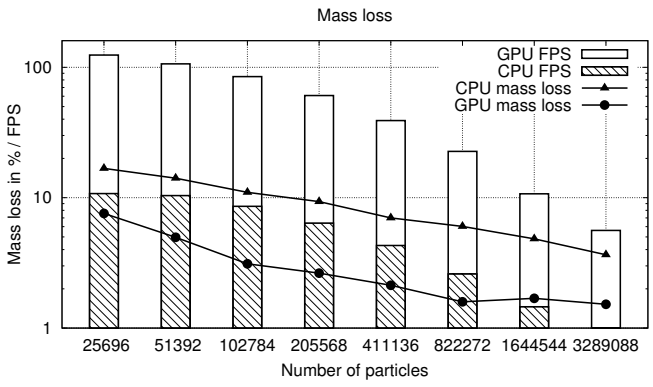

Figure 6: *Comparison of CPU and GPU method for Zalesak's sphere (same logarithmic scale for both) – the results have been taken after* 360° *rotation, grid resolution:* 64^3*, no particle reinitialization*

Following prior works (e.g. Enright et al. [ELF04]), the resulting mass loss is measured by counting the number of interior voxels of the object. It should be pointed out, that this method is not very accurate, since it does not take sub-voxels into account. Also, PLS tends to produce bumps on the interface resulting in a slightly fluctuating volume size. For some high resolutions (128^3 and 524,288 particles), we even get a slight volume *gain*.

In Fig. 8, two examples are given, one where the initial state and the state after rotation by 360° in a simple vortex field without and with PLS correction is visualized, the other showing a deformation using PLS within a distorting vortex field. Fig. 7 shows the advantage of PLS for a time surface after 122 evolution steps in the typhoon flow. Note that fine features close to the typhoon's vortex disappear in the version without the particle correction.

Table 1 lists the time consumption for all steps of our GPU-based algorithm separately for Zalesak's sphere. One can see that, depending on the resolution, the level set reinitialization and correction are the most time-consuming steps of the application. Due to the large size of the structure element used in the propagation method, the level set reinitialization is texture-fetch-bound (see Sec. 4.2).

The usage of distance transform instead of a distance field increases the memory footprint of the level set texture. However, the advantage of using references for particle reinitialization is clearly noticeable in the direct comparison in Fig. 3, see also the discussion in Sec. 4.2.

Fig. 9 shows an evolution of a complex time surface in the typhoon flow. Note how the cylindrical holes in the inner region of the volume are pulled into the vortex of the typhoon. The interior flow structure is clearly visible due to the semi-transparent interval volume rendering (see Sec. 5).

A path volume consisting of 9 spheres is given in Fig. 9. Without particle correction, the path volumes do not

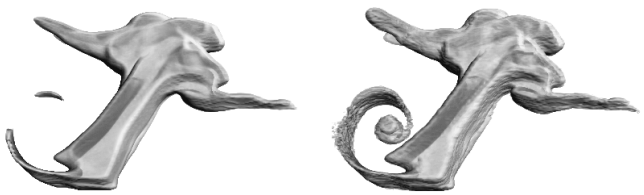

Figure 7: *Time Surface in a dynamic typhoon flow field. The flow data consists of* $106 \times 53 \times 39$ *voxels at* 32 *distinct time steps – state after 122 evolution steps without PLS correction (left) and without PLS correction (right); 524,288 particles, resolution* 128^3*, frame-rate* 9.23 *FPS*

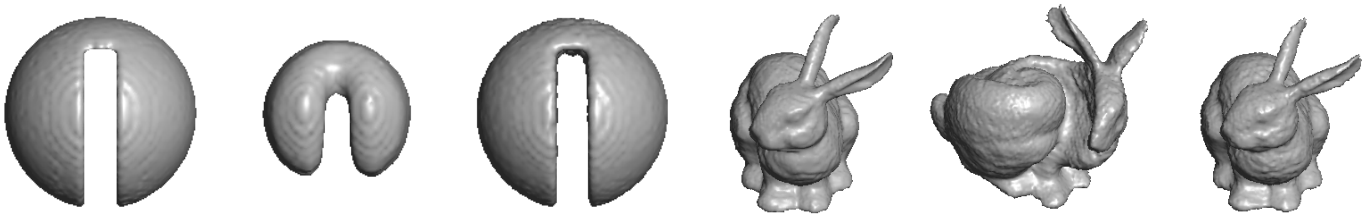

Figure 8: *Left:* 360° *rotation of Zalesak's sphere: initial, rotated without and with correction; 524,288 particles, 14.74 FPS, grid resolution:* 128^3 *– right: Stanford bunny in a vortex flow using PLS: initial, after 40 advection steps, after backward advection; 131,072 particles, 96x96x96, 40.4467 FPS*

Table 1: *Run-time of the steps involved in the PLS algorithm – object: Zalesak's sphere (see Fig. 8), 262,144 particles. In each frame,* 5% *of the particles are reinitialized.*

step	grid res.	time in ms
framework	64^3	5.01
level set advection	64^3	0.41
particle tracking	64^3	0.31
LS correction	64^3	5.95
level set reinitialization	64^3	1.29
particle reinitialization	64^3	0.22
framework	128^3	5.05
level set advection	128^3	0.27
particle tracking	128^3	0.44
LS correction	128^3	7.64
level set reinitialization	128^3	29.52
particle reinitialization	128^3	0.06

evolve properly through the complete flow volume. Applying particle correction yields a well-behaved path volume, especially around the flow source (central path volume).

7. Conclusions and Future Work

We have presented an enhanced and purely GPU-based Particle level set method with a beneficial application to surface and volume based flow visualization. The presented method shows that surface evolution can be performed efficiently and accurately on the GPU. We achieve a convincing performance and far superior quality of results over both, CPU-based PLS methods and grid-only GPU-methods. The use of distance transforms instead of distance fields alleviates the difficult problem of even particle distribution in a dynamically changing region and offers sub-voxel accurate distances for higher quality representation of moving surfaces.

Examples of accurate and interactive flow visualizations, including path volumes and time surfaces, have been presented. Especially for inhomogeneous and divergent flow fields such as a complex unsteady typhoon flow, the method allows for a good balance between accuracy and performance by means of grid resolution and number of particles.

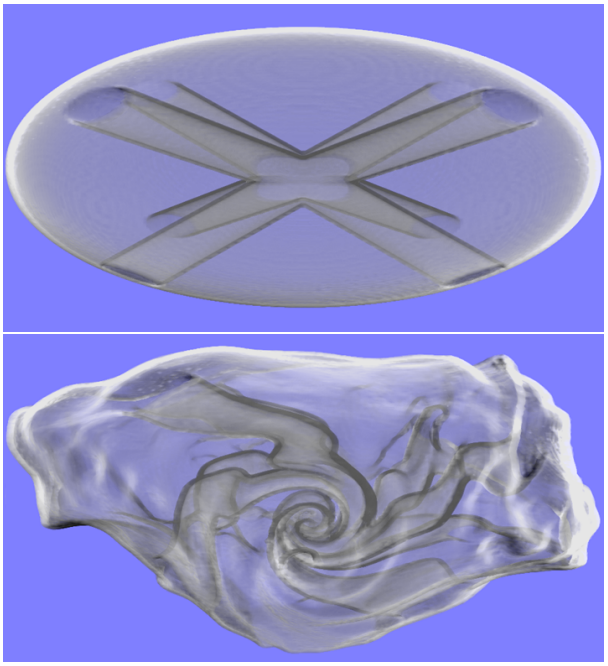

Figure 9: *Semi-transparent time surface visualizing the typhoon flow; about* 4 *mill. particles, grid res.:* $160 \times 160 \times 128$

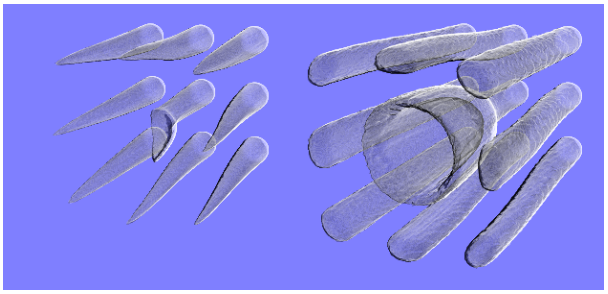

Figure 10: 9 *spheres forming path volumes in an analytical flow field without (left) and with PLS correction (right);* 524288 *mill. particles, resolution:* $160 \times 160 \times 140$

We will continue work on the optimization of the implementation, e.g. on the particle reinitialization scheme and the scattering of particles. In principle, any dynamically generated velocity field can be used to drive the interface motion in a physically based or artistically controlled way. Consequently, we want to explore how our GPU-based PLS method can enable interactive design of special effects, e.g. with velocity fields generated by a CFD GPU engine. The parameters of the effects could be easily recorded and later employed in real-time applications like computer games.

Appendix A: Fragment Shader

The following GLSL code is used to calculate ϕ^+ and ϕ^- during the particle correction. The result is stored in $\Phi^{\pm}_{xy}$. The particle information is given by `_particle`, which corresponds to $\mathbf{P}_{xyzd}(\mathbf{p})$, the voxel position is passed from the vertex program via `_voxel`

Listing 1 *(Calculating $\Phi^{\pm}_{xy}(x)$)*

```
vec2 phi_pm; // temp. pos./neg. level sets

// Compute phi_p(position)
float phi_p = sign(_particle.w)*
  (abs(_particle.w)-
          distance(_voxel, _particle.xyz));

phi_pm = vec2(phi_p, -INFTY);  // pos. side
if (_particle.w < 0.0) // change to neg. side
  phi_pm.xy = vec2(phi_pm.y, -phi_pm.x);

return phi_pm;
```

The following GLSL code is used to update the level set function at the end of the correction step. The function `mix` computes a linear interpolation between two values.

Listing 2 *(Updating $\Phi_d(x)$)*

```
// The level set which is to be corrected
float phi_r=texture2DRect(phi_r_sampler,
                          gl_TexCoord[0].xy);
// Intermediate level set
vec2 phi_pm=texture2DRect(phi_pm_sampler,
                          gl_TexCoord[0].xy);

phi_pm.x = max(phi_pm.x,  phi_r.x);
phi_pm.y = max(phi_pm.y, -phi_r.y);
float alpha;
alpha = float(abs(phi_pm.r)>=abs(phi_pm.g));

phi_r = mix(-phi_pm.g, phi_pm.r, alpha);

return phi_r;
```

References

BHR*94. BRILL M., HAGEN H., RODRIAN H.-C., DJATSCHIN W., KLIMENKO S.: Streamball techniques for flow visualization. In *Proc. IEEE Conf. on Visualization* (1994), pp. 225–231.

Buc05. BUCK I.: Taking the plunge into GPU computing. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 32, pp. 509–519.

CK07. CUNTZ N., KOLB A.: Fast hierarchical 3d distance transforms on the GPU. *Technical Report* (2007). http://www.cg.informatik.uni-siegen.de/Publications/.

CSM00. CRAWFIS R., SHEN H., MAX N.: Flow visualization techniques for cfd using volume rendering. In *Int. Symp. on Flow Visualization* (2000).

Cui99. CUISENAIRE O.: Distance transformations: Fast algorithms and applications to medical image processing. *Ph.D. Thesis, UCL, Louvain-la-Neuve, Belgium* (1999).

ELF04. ENRIGHT D., LOSASSO F., FEDKIW R.: A fast and accurate semi-lagrangian particle level set method. *Computers & Structures, Volume 83, Issues 6-7* (2004), 479–490.

EMF02. ENRIGHT D., MARSCHNER S., FEDKIW R.: Animation and rendering of complex water surfaces. In *ACM Proceedings SIGGRAPH* (2002), pp. 736–744.

FMST96. FUJISHIRO I., MAEDA Y., SATO H., TAKESHIMA Y.: Volumetric data exploration using interval volume. *IEEE Trans. on Visualization and Computer Graphics2*, 2 (1996), 144–155.

GRNG05. GRIESSER A., ROECK S. D., NEUBECK A., GOOL L. V.: Gpu-based foreground-background segmentation using an extended colinearity criterion. In *Proc. Vision, Modeling and Visualization* (2005), pp. 319–326.

Har05. HARRIS M.: Mapping computational concepts to GPUs. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 31, pp. 493–508.

KC05. KOLB A., CUNTZ N.: Dynamic particle coupling for GPU-based fluid simulation. In *Proc. 18th Symposium on Simulation Technique, ISBN 3-936150-41-9* (2005), pp. 722–727.

KKKW05. KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3d flows. *IEEE Trans. on Visualization and Computer Graphics11*, 6 (11 2005).

LKHW04. LEFOHN A., KNISS J., HANSEN C., WHITAKER R.: A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces. *IEEE Transactions on Vi-*

sualization and Computer Graphics 10, 4 (2004), 422–433.

MBC93. MAX N., BECKER B., CRAWFIS R.: Flow volumes for interactive vector field visualization. In *Proc. IEEE Conf. on Visualization* (1993), pp. 19–24.

MF06. MOKBERI E., FALOUTSOS P.: A particle level set library. *Technical Report* (2006). http://www.magix.ucla.edu/software/levelSetLibrary/.

OF02. OSHER S., FEDKIW R.: Level set methods and dynamic implicit surfaces. *Springer, ISBN 0-387954-82-1* (2002).

OS88. OSHER S., SETHIAN J.: Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics 79* (1988), 12–49.

RS01. RUMPF M., STRZODKA R.: Level set segmentation in graphics hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)* (2001), vol. 3, pp. 1103–1106.

RT06. RONG G., TAN T.-S.: Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *ACM Symposium on Interactive 3D Graphics and Games, 14–17 March, Redwood City* (2006), pp. 109–116.

Set99. SETHIAN J. A.: *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.

SGGM06. SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3D distance field computation using linear factorization. In *Proc. Symp. on Interactive 3D graphics & games* (2006), pp. 117–124.

SHK97. SILVA C., HONG L., KAUFMAN A.: Flow surface probes for vector field visualization. In *Scientific Visualization, Overviews, Methodologies, and Techniques* (1997), pp. 295–310.

SPG03. SIGG C., PEIKERT R., GROSS M.: Signed distance transform using graphics hardware. In *Proc. IEEE Conf. on Visualization* (2003), p. 12.

ST04. STRZODKA R., TELEA A.: Generalized distance transforms and skeletons in graphics hardware. In *VisSym, Symposium on Visualization* (2004), pp. 221–230.

vW93. VAN WIJK J.: Implicit stream surfaces. In *Proc. IEEE Conf. on Visualization* (1993), pp. 245–252.

Wei04. WEISKOPF D.: Dye advection without the blur: A level-set approach for texture-based visualization of unsteady flow. In *Proc. EUROGRAPHICS* (2004), pp. 479–488.

WJE00. WESTERMANN R., JOHNSON C., ERTL T.: A level-set method for flow visualization. In *Proc. IEEE Conf. on Visualization* (2000), pp. 147–154.

XZC04. XUE D., ZHANG C., CRAWFIS R.: Rendering implicit flow volumes. In *Proc. IEEE Conf. on Visualization* (2004), pp. 99–106.