Interactive Streak Surface Visualization on the GPU

Kai Bürger, Florian Ferstl, Holger Theisel, and Rüdiger Westermann

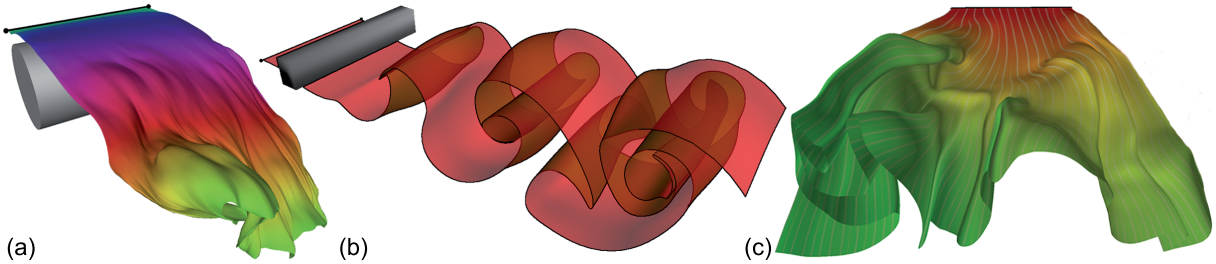

Fig. 1. Our method generates adaptively refined integral surfaces in 3D flows on the GPU. The shown surfaces consist of 800k, 300k and 400k particles, respectively, and they were generated and rendered in less than 50ms. Figure a) and b) show streak surfaces in unsteady flows. Figure c) shows a stream surface.

Abstract—In this paper we present techniques for the visualization of unsteady flows using streak surfaces, which allow for the first time an adaptive integration and rendering of such surfaces in real-time. The techniques consist of two main components, which are both realized on the GPU to exploit computational and bandwidth capacities for numerical particle integration and to minimize bandwidth requirements in the rendering of the surface. In the construction stage, an adaptive surface representation is generated. Surface refinement and coarsening strategies are based on local surface properties like distortion and curvature. We compare two different methods to generate a streak surface: a) by computing a patch-based surface representation that avoids any interdependence between patches, and b) by computing a particle-based surface representation including particle connectivity, and by updating this connectivity during particle refinement and coarsening. In the rendering stage, the surface is either rendered as a set of quadrilateral surface patches using high-quality point-based approaches, or a surface triangulation is built in turn from the given particle connectivity and the resulting triangle mesh is rendered. We perform a comparative study of the proposed techniques with respect to surface quality, visual quality and performance by visualizing streak surfaces in real flows using different rendering options.

Index Terms—Unsteady flow visualization, streak surface generation, GPUs.

1 Introduction

In interactive flow visualization, the integration and visualization of stream lines has been a standard tool from its very beginning. With the consideration of time-dependent flows, path lines and streak lines have moved into the focus of research because they reflect important properties of the flow: while a path line describes the path of a massless particle in the flow, a streak line shows the positions of particles that have been released continuously at a fixed location in the past.

The visualization of integral surfaces has been proven to be common and useful in visual flow exploration. In the case of stream and path surfaces, their extraction is well-understood. The main idea is to integrate the front line of the surface and apply if necessary an adaptive refinement/coarsening to it. After the front has passed, the generated surface remains unchanged.

Streak surfaces have a strong relation to experimental flow visualization where external materials such as dye, hydrogen bubbles or heat energy are injected into the flow. The advection of these external materials creates streak lines and shows the flow patterns. Due to this reason, analogues to these experimental techniques have been adopted by researchers in computer-aided scientific visualization for flow exploration. However, up to now streak surfaces are rarely applied because of the computational complexity of streak surface generation. Since streak surfaces may change their shape everywhere and at any time of the integration, every part of the surface has to be monitored at any time of the integration for adaptive refinement/coarsening. Due to this fundamental difference to stream and path surfaces, the consideration of streak surfaces makes only sense if their evolution over time is shown, e.g., in a pre-computed video sequence or in interactive applications with a real-time performance.

The only approach so far to address the real-time requirement was proposed by Funck et al. [20]. It combines the streak surface integration with a smoke metaphor, leading to cancelation effects of problematic surface parts: parts of the streak surface where an adaptive refinement is necessary are rendered less opaquely. In this way, smoke like structures are obtained by a streak surface integration without any adaptive refinement. On the other hand, the value of this approach for visual flow exploration is limited because it cannot guarantee to find all relevant flow structures and fine structures can only be revealed if the initial tessellation of the mesh already respects these subtleties.

2 Contribution

In this paper, to the best of our knowledge, we present the first real-time approach for adaptive streak surface integration and high-quality rendering. We achieve this by using particle-based approaches in which either the surface is represented as a set of surface patches that can be handled independent of each other, or a closed surface triangulation is computed from the given particle set. For both approaches we have developed methods for interactive surface refinement and coarsening based on local surface properties.

While the former approach is elegant in its simplicity, it requires redundant particle computations and lacks flexibility in the rendering process. Even though we use an advanced rendering method similar to high-quality point-splatting [2], rendering artifacts at patch boundaries

- *K. Bürger (E-mail: buerger@tum.de) and R. Westermann (E-mail: westermann@tum.de) are with the Computer Graphics & Visualization group, Technische Universität München.*
- *Holger Theisel (E-mail: theisel@isg.cs.uni-magdeburg.de) is with the Visual Computing group, University of Magdeburg.*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

can not be avoided entirely. The second approach, on the other hand, yields a closed surface representation providing a variety of rendering options, but it can result in deformed triangulations and rendering artifacts thereof.

Our paper makes the following specific contributions:

- A patch-based scheme for the adaptive generation of streak surfaces, and a high-quality patch-based surface rendering technique.
- A particle-based adaptive refinement/coarsening scheme for streak surface generation, and a novel method to construct a closed triangular streak surface from a set of particles.

These approaches run entirely on the GPU and allow for real-time performance if certain requirements are fulfilled. In particular, for an efficient processing of time-dependent fields, as many consecutive time steps as are required by the numerical integrator are supposed to fit into GPU memory. In addition, our methods assume the flow fields to be given on Cartesian grids that can be stored in 3D texture maps on the GPU. As it was shown in Krueger et al. [10], particle tracing can then be mapped efficiently on the SIMD architecture underlying current GPUs. In principle, the extension of our approaches to tetrahedral grids is straight forward [15], but it comes at an additional expense due to more complex operations for point location and interpolation, as well as the fact that particles might have to integrate over many elements in one time step.

The remainder of this paper is organized as follows. In the next chapter we review previous work that is related to ours. An introduction to streak surfaces is given in Section 4. Section 5 presents a novel technique to construct and render a patch-based streak surface representation. In Section 6 we describe the particle-based technique for streak surface generation in which local connectivity information is used to build a surface triangulation. In Section 7 we evaluate the performance of our approaches, and we discuss their advantages and limitations. We conclude the paper with an outline of future research in the field.

3 Related Work

We do not attempt here to survey the vast number of approaches in stream line and path line integration because they are standard in flow visualization. For a thorough overview, however, let us refer to the report by Post et al. [11]. Graphics hardware related algorithms for interactive flow exploration were presented in [3, 4, 5, 10, 15, 17]. The usefulness of streak lines for the exploration of time-dependent flow fields has been proven in several applications, but due to the computational complexity of streak line integration and adaptive stream surface construction streak surfaces have only rarely been used in practice.

Hultquist [9] presented the first adaptive stream surface integration approach, which was later extended in different ways: The approach by Stalling [18] uses local topological information to increase accuracy. Scheuermann et al. [14] compute exact solutions of stream surfaces inside piecewise linear vector fields. In the work by van Wijk [19] a global implicit approach for certain stream surfaces is given. Recently, a construction method for stream surfaces of high polynomial precision has been introduced by Schneider et al. [16]. Garth et al. [8] discussed a number of enhancements in the context of vortex extraction. In another work by Garth et al. [7], improved integral surface accuracy was achieved by separating characteristic line integration and integral surface triangulation. A particle-based approach for the generation and rendering of stream surfaces was proposed by Schafhitzel in [13].

The methods proposed by Schafhitzel [13] and Garth et al. [7] are also the only approaches describing the surface extraction in a time-dependent context for path surfaces. The generalization from stream surfaces to path surfaces is rather straightforward because only the kind of integration at the surface front has to be replaced.

The first approach for streak surface integration is the smoke surface approach presented by Funck et al. in [20]. There, the adaptivity problem of streak surfaces is solved by extracting the surface with a smoke metaphor and therefore avoiding any adaptive refinement. While this gives interesting smoke-like structures, it is unable to produce fully adaptive opaque streak surfaces.

4 Streak Surfaces

Streak surfaces are defined by repeatedly setting out particles on a line-shaped seeding structure over a certain time interval. The collection of all these particles at a certain time denotes the streak surface. Technically, a streak surface can be obtained in the following way for a 3D time-dependent flow field $\mathbf{v}(\mathbf{x},t)$: the seeding structure is considered to be a polyline consisting of the points $\mathbf{s}_0,...,\mathbf{s}_n$. At the time $t_i = t_0 + i\,\Delta t$ we start a path line integration of the particle $\mathbf{x}_{i,j}$ from the seeding point $\mathbf{s}_j$ and observe its behavior over t:

$$\mathbf{x}_{i,j}(t) = \mathbf{x}_{i,j}(t_i) + \int_{t_i}^{t} \mathbf{v}(\mathbf{x}_{i,j}(s),s)\,ds \tag{1}$$

with $\mathbf{x}_{i,j}(t_i) = \mathbf{s}_j$, $i = 0,...,m$ and $j = 0,...,n$. For $t \geq t_m = t_0 + m\,\Delta t$, the streak surface can be considered as a rectangular vertex array $(\mathbf{x}_{i,j}(t))$. We call a column $(\mathbf{x}_{i,0},...,\mathbf{x}_{i,n})$ a time line, while a row $(\mathbf{x}_{0,j},...,\mathbf{x}_{m,j})$ is a streak line. The vertices are the surface points from which a closed surface representation has to be built.

During the integration, the distance between both adjacent time lines and streak lines may vary at any location of the surface. Thus, after every integration step the surface has to be checked everywhere for adaptive refinement or coarsening. This means that, based on an appropriate refinement/coarsening criterion, new particles have to be seeded between adjacent points along a particular time or streak line, or adjacent points have to be merged. This process is computationally very complex because streak surfaces appear to have a rather large distortion after their seeding. An increase of the surface area by a factor of 100 or more is usual, leading to a high number of refinement steps. It is worth noting that in an interactive application, the adaptive refinement/coarsening has to be monitored and carried out at any time simultaneously with real-time performance.

5 Patch-based Streak Surface Generation

By using a patch-based approach, the streak surface generation and rendering process is split into a set of independent operations on each patch. These operations can then be executed in parallel, and all the patches can be rendered independent of each other. The computation of adjacency information between surface points, as it is required for the computation of a surface triangulation, can be avoided.

5.1 Patch Generation and Refinement

As described in Section 4, a streak surface can be constructed by repeatedly setting out particles on a line-shaped seeding structure over a certain time interval and by connecting these particles to form a closed surface. All particles $(\mathbf{x}_{i,0},...,\mathbf{x}_{i,n})$ released at time $t_i = t_0 + i\,\Delta t$ reside on one advancing front. We call this front the time line tl_i.

Whenever a new time line tl_i is released, our approach computes n quadrilateral patches $\mathbf{p}_{i,v}$, with $v = 0,...,n-1$. Each patch consists of four vertices $(\mathbf{s}_v, \mathbf{s}_{v+1}, \mathbf{x}_{i,v}, \mathbf{x}_{i,v+1})$, which are duplicated and stored separately for each patch. The initial time line is removed. The patch vertices are then advected through the flow as described before, and the shape changes a patch undergoes due to the particles movements are used to steer the refinement process.

The refinement of surface patches is performed for each patch separately wrt. an area-based criterion. Specifically, we set a threshold to Ξ^2, where Ξ is the distance between adjacent points on the seeding structure. If, at any time, the area of a patch is greater than $\alpha\Xi^2$, where α is a real number larger than 1 controlling the subdivision strength, the patch is subdivided into two quadrilaterals. This is performed by splitting the patch along its longest edge and the edge opposite to it. The two new patches and their vertices are stored separately, and the refined patch is removed (see Figure 2).

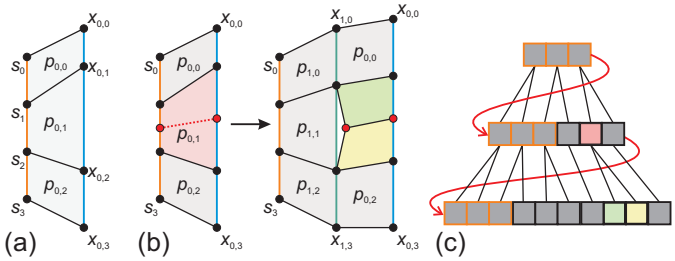

Fig. 2. (a) A patch-based streak surface representation after the first time line has been released. (b) Left: Patch $p_{0,1}$ meets the refinement criterion and is split into two patches. Right: The surface patches after the second integration step. The generation of new surface points due to the splitting operation has lead to a hole in the surface representation. (c) The corresponding layout of the linear memory segments storing the surface patches in each time step.

5.2 GPU Implementation

Today, advanced GPU functionality [1] provides new possibilities to efficiently perform patch-based streak surface generation: the *geometry shader* can manipulate a primitive stream by appending or removing primitives, and the *stream out stage* can be used to direct the resulting stream to intermediate buffers in GPU memory. Since buffer resources cannot be bound as pipeline input and stream out target simultaneously, we use two instances and toggle between them in a ping-pong fashion.

Each surface patch is represented by its four vertices, a scalar value counting the number of integration steps, and a counter indicating its refinement depth. On the GPU, for every patch this information is stored as one contiguous data block in a vertex array buffer. Since current GPUs cannot change the size of a resource residing in GPU memory dynamically, a buffer that is large enough to store the entire streak surface has to be allocated before the surface construction begins. By letting the user select the number n of patches that are released in each time step, the maximum refinement depth d, as well as the maximum number of integration steps m a patch can perform until it is removed, the buffer must be able to store $n \times 2^d \times (m-d+1)$ patches.

The streak surface construction starts by storing n zero area patch primitives $\mathbf{p}_j^0$, with $j = 0,\ldots,n-1$, at the beginning of the vertex array buffer. In the following we assume n to be an even number. These elements are used in every time step to repeatedly release a new patch front into the flow. The respective vertices of patch $\mathbf{p}_j^0$ are $(\mathbf{s}_j, \mathbf{s}_{j+1}, \mathbf{s}_j, \mathbf{s}_{j+1})$. In each integration step, all buffer elements are passed to the geometry shader and processed as follows: For each of the first $n/2$ elements $\mathbf{p}_j^0$ with $j = 0,\ldots,n/2-1$ the shader writes the two zero area patches $\mathbf{p}_{2\times j}^0$ and $\mathbf{p}_{(2\times j)+1}^0$ to the output buffer. Access to the vertices of these patches is achieved by binding the input stream buffer as shader resource. Since these n patches are written first, they are always at the beginning of the buffer. For each of the remaining $n/2$ elements the shader appends two patch elements to the buffer, which represent the currently released patch front. These patches are then expanded by integrating their last two vertices to new positions.

For the remaining buffer elements, which contain patches that were released into the flow at previous time steps, the refinement criterion is evaluated before the integration is performed. If a refinement is not carried out, the geometry shader advects the patch vertices, increments the integration step counter and appends the patch element to the output stream. Otherwise, the geometry shader splits the element as described, advects the four original as well as the two new vertices, and appends the two new primitives to the output stream. The refinement counters of the new primitives are set to the counters of the refined patch and incremented by one. Figure 2(c) illustrates the growth of the vertex array buffer due to the seeding and refinement of surface patches.

5.3 Patch-based Streak Surface Rendering

The patch-based surface representation can be rendered directly by sending the vertex array buffer through the graphics pipeline and rasterizing the patches separately. However, since T-vertices are introduced by the particular refinement strategy, holes in the surface representation can occur. To cover these holes, we adopt a rendering technique that was introduced by Botsch and co-workers in the context of point splatting [2]. Figure 3 shows an adaptively refined patch-based streak surface ($\alpha = 1.2$), which was rendered using simple point rendering of the patch centroids (left) and the patch-based splatting approach (right).

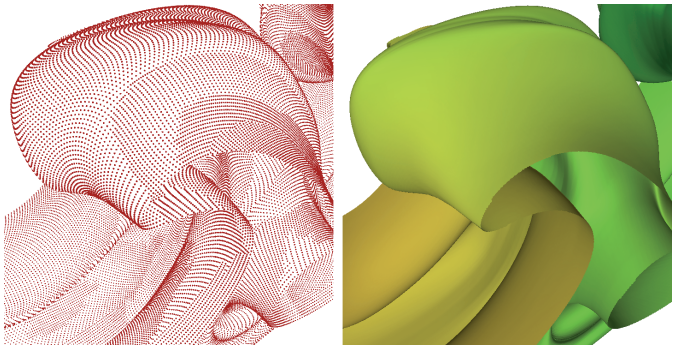

Fig. 3. (Left) Rendering of the patch centroids of a patch-based streak surface. (Right) The same streak surface rendered via quad-splatting.

A two pass rendering approach is performed before deferred per-pixel surface lighting is computed. Therefore, all patches in the vertex buffer are rendered twice. In each pass a geometry shader enlarges every patch by changing its vertices px_k $(k = 0,\ldots,3)$ to $px_k + \beta||px_k - c||$. Here, c is the patch centroid and β is a user defined scaling factor. As shown in Figure 4, patches are then split into the four triangles spanned by their centroid and the patch vertices before they are rendered.

In the first rendering pass, commonly referred to as visibility pass, a depth imprint of the enlarged surface patches closest to the viewer is generated. In the second pass, also known as attribute pass, the patch-based surface representation is rendered again using a biased depth test on the generated depth imprint. In this way, only patch samples close to the first rendered surface survive. In a pixel shader, the patch attributes like color and normal are weighted by a Gaussian kernel centered at the patch centroid, and these contributions are finally accumulated via additive blending and normalization. In this way, a smooth transition of patch attributes is obtained in regions where multiple enlarged patches overlap.

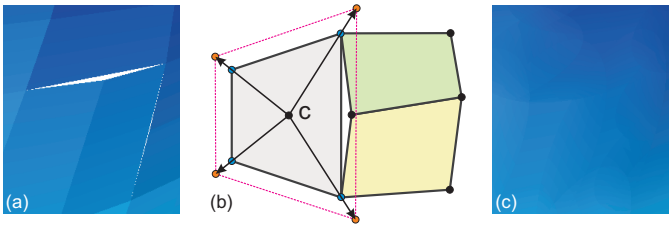

Fig. 4. (a) Separate triangulation and rendering of each patch leads to holes in the streak surface. (b) Holes are covered by rendering enlarged patches. (c) Rendering enlarged patches in a way similar to high-quality point splatting yields a closed and smooth surface visualization.

Due to the bending of streak surfaces, it can happen that surface samples having a large geodesic distance from each other become close to each other and fall into the same pixel. In this case, the biased depth test might let both samples pass and accumulate in the pixel buffer. To avoid this we assign two additional parameter values to each patch. The first value indicates a patch's position in the ordered set of

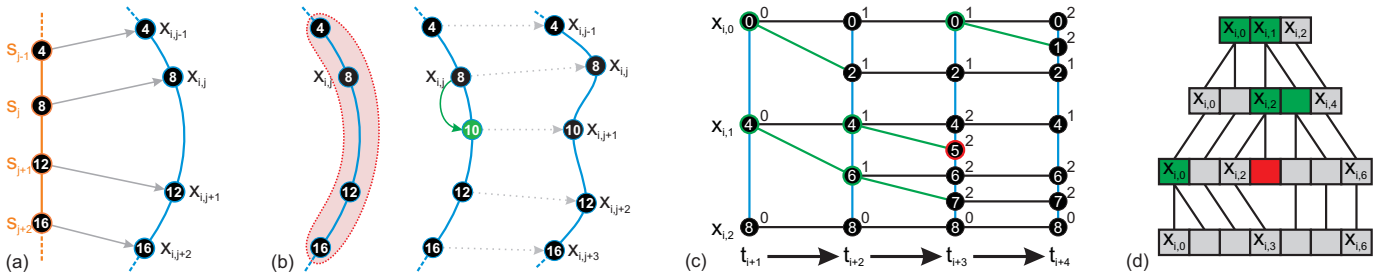

Fig. 5. (a) In each time step a new time line is released from the seeding line. Node values show particle ids. (b) Left: Prior to integration, each particle $x_{i,j}$ evaluates refinement criteria based on its local neighborhood (red). Middle: $x_{i,j}$ satisfies a refinement criterion and performs a particle split. Right: The resulting time line after the subsequent integration step. (c) Evolution of a time line over four integration steps. Green and red nodes indicate refinement and coarsening events, respectively. Numbers next to the nodes indicate the refinement level. (d) Changes in the linear memory segment b_i due to vertex refinement/coarsening.

all possible patches along the seeding structure. Starting with the initial patches $\mathbf{p}_{i,v}$, which are assigned the positions $v \times 2^d$, in every refinement step the first new patch keeps the position of the refined patch and the second patch adds 2^{d-k} to this position. Here, k is the current refinement level. The index i of each patch is assigned as the second parameter value. In the visibility pass, these values are rendered into a separate render target, and they are then used in the attribute pass to discard those fragments that are close to the rendered surface samples but have parameter values that differ more than a given threshold.

6 Mesh-based Streak Surface Generation

Patch-based streak surface generation entirely avoids to store and update any connectivity between the patches. On the other hand, because every patch stores its own set of vertices even though they might be shared between patches, a considerable amount of memory is wasted and numerical integration of the same vertex is performed up to four times. To overcome this overhead we propose a novel GPU approach to construct an adaptive triangular streak surface representation from the set of seeded particles.

Similar to the data layout that was used in the patch-based approach, all particles seeded into the flow are stored in a linear vertex buffer. Each initially seeded particle $x_{i,j}$ is assigned an index $id_{i,j} = j \times 2^d$, where d is the maximum refinement depth. The particle set belonging to a particular time line tl_i is stored in a contiguous block b_i in this buffer. The blocks are ordered such that block b_{i-1} follows block b_i, with block b_0 being the last in the buffer.

In every time step the particles are processed in the order of their occurrence in the buffer, and they are written to the output buffer in the same order. If a new particle is generated due to the splitting of an existing particle, it is placed directly behind this particle in the output buffer. If a particle is removed, it is simply not written into this buffer. On the GPU this is realized by executing a geometry shader with a variable primitive output of 0-2 elements for each incoming primitive. In the same way as described in the previous section, the maximum buffer size has to be computed up front depending on the number of particles per time line, the maximum refinement depth, and the maximum number of integration steps. Then, two ping-pong buffers of this size have to be allocated.

6.1 Particle Refinement

Our method for generating an adaptive streak surface triangulation from a given set of subsequently released time lines can be separated into three passes:

- *Time line refinement*: Every time line is refined/coarsened based on local criteria like stretching, compression, and line curvature, as well as a global criterion taking into account the change in surface area.
- *Connectivity update*: The connectivity between particles on adjacent time lines is established.
- *Streak line refinement*: The connectivity information is used to compute local streak line properties, which are considered to steer the refinement of streak lines.

6.1.1 Time line refinement

Time line refinement adapts the particle density along each time line prior to the particle integration. The refinement/coarsening criteria we apply have been adopted from previous work in the field. The first criterion considers the flow divergence at a particle position as introduced in [9]. Let $\Phi(x,y)$ be the distance between particles x and y, and Ξ the initial distance between two adjacent seed points, then the particle $x_{i,j}$ spawns a new particle between $x_{i,j}$ and $x_{i,j+1}$—we call this operation particle splitting—if

$$\Phi(x_{i,j}, x_{i,j+1}) > \alpha\, \Xi \tag{2}$$

Similar to [8], the second criterion considers the approximate local curvature along a time line. Let $\Theta(u,v,w)$ be defined as

$$\Theta(u,v,w) = \frac{\left(\frac{u-v}{||u-v||} \cdot \frac{w-v}{||w-v||}\right) + 1}{2} \tag{3}$$

where u,v,w are three particle coordinates. A particle $x_{i,j}$ is split if

$$\Theta(x_{i,j-1}, x_{i,j}, x_{i,j+1}) + \Theta(x_{i,j+2}, x_{i,j+1}, x_{i,j}) > \beta \tag{4}$$

In this way, the deviation of the time line from a straight line is approximated and used to steer the local time line refinement.

Particle splitting is performed by fitting a cubic polynomial $p(t)$ through $x_{i,j-1}$, $x_{i,j}$, $x_{i,j+1}$ and $x_{i,j+2}$, and by evaluating $p(t)$ at $t = \frac{1}{2}$:

$$p(1/2) = -\frac{1}{16}(x_{i,j-1} + x_{i,j+1}) + \frac{9}{16}(x_{i,j} + x_{i,j+2}) \tag{5}$$

Based on the indices $id_{i,j}$ of the initially seeded particles $x_{i,j}$, every new particle on a time line gets assigned its index in the ordered set of all possible particles along this line as described in the previous section for surfaces patches. We will subsequently call these indices the particle ids. Figure 5 illustrates the changes in the particle layout on a time line due to refinement and coarsening events.

To prevent the streak surface from unlimited stretching, we adapt a criterion that was proposed for stream surfaces in [9]. We compare the current distance between two particles to their distance in the last time step in relation to the distance a particle has moved due to the integration. Let $\Psi(x,y,t)$ be the distance between particles x and y at time t, and $x_{i,j,t}$ the position of particle $x_{i,j}$ at time t. We mark an edge as invalid, meaning that it will not be refined any further, if the following expression evaluates to true:

$$\Psi(x_{i,j}, x_{i,j+1}, t) - \Psi(x_{i,j}, x_{i,j+1}, t-1) > \gamma\, \Phi(x_{i,j,t}, x_{i,j,t-1}) \tag{6}$$

If an edge has been classified as invalid or cannot be refined any further, it is not considered in the triangulation of the streak surface

described below. In this way, the surface is cut in regions where it stretches too much, e.g., if it evolves around obstacles in the flow as demonstrated in Figure 6. Finally, in addition to inserting new particles we remove a particle $x_{i,j}$ if the following condition is met:

$$(\Phi(x_{i,j},x_{i,j-1})+\Phi(x_{i,j},x_{i,j+1})<\delta\Xi)\ \wedge$$
$$(\Theta(x_{i,j-1},x_{i,j},x_{i,j+1})+\Theta(x_{i,j},x_{i,j+1},x_{i,j+2})<\zeta) \qquad (7)$$

Due to this coarsening we avoid vertex clustering in regions of high convergence, and we prevent the generated triangles from becoming too small.

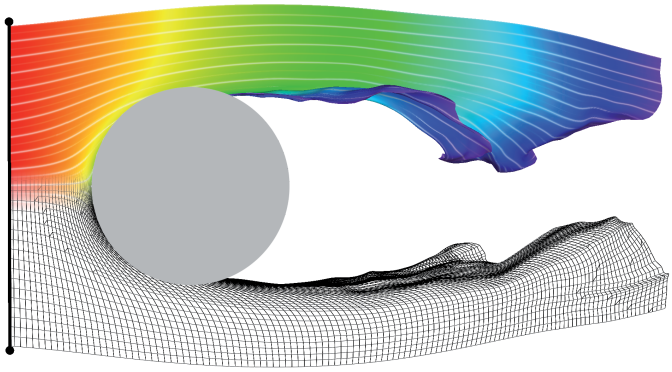

Fig. 6. Application of criterion (6) prevents a streak surface from unlimited stretching by cutting edges if no additional refinement can be performed.

6.1.2 Connectivity update

Due to time line refinement and coarsening the connectivity between particles on adjacent time lines has to be computed in each integration step. Therefore, every particle on time line tl_i searches for the particle on tl_{i+1} and the one on tl_{i-1} having the id closest to its own one on the respective time line. We will call these two particles the predecessor and the successor of a particle. In particular, for a particle $x_{i,j}$ we select the successor $x_{i+1,succ}$ with the closest id $\leq$ the particle's id and the predecessor $x_{i-1,pred}$ with the closest id $\geq$ the particle's id (see Figure 7 (a)). Once the predecessor and the successor have been determined, references to them are stored as offsets to the absolute position of the particle in the vertex array buffer, and they are used as described below to build a closed surface representation.

Finding the two particular neighbors requires every particle to search the vertex buffer to the left and to the right of it, with the search radius depending on the number of particles on time lines tl_{i-1}, tl_i and tl_{i+1}. We will describe in Section 6.3 how to determine these numbers in a very efficient way on the GPU.

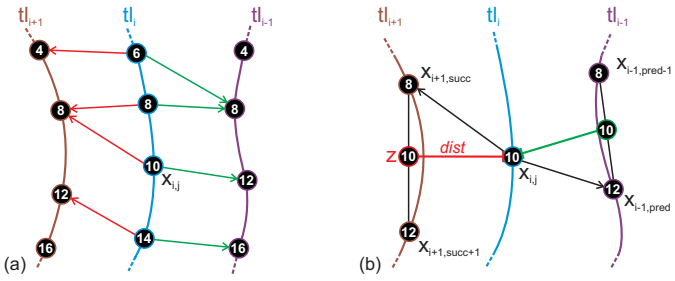

Fig. 7. (a) Each particle on time line tl_i selects its successor (red arrows) and predecessor (green arrows) on adjacent time lines based on the closest matching particle id. (b) The distance estimate of a particle $x_{i,j}$ to its adjacent time line tl_{i+1} is based on an intermediate particle z, exhibiting the same particle id.

6.1.3 Streak line refinement

In this pass, a complete time line is added to or removed from the streak surface. The criterion to steer the refinement/coarsening is based on the maximum Euclidean distance between neighboring time lines.

A new time line is inserted between tl_i and tl_{i+1} if the maximum of the shortest distances between particles on tl_i and the time line tl_{i+1} exceeds a user defined threshold. An existing time line is removed if the maximum of the shortest distances to both adjacent time lines falls below a given threshold. Unfortunately, since we do not know the exact time line between the given vertices, computing the shortest distance from a particle to this line is not possible in general. Therefore, we proceed as follows: Since $x_{i+1,succ}$ is the closest existing control point on tl_{i+1} with $id_{i+1,succ} \leq id_{i,j}$ and its adjacent particle $x_{i+1,succ+1}$ has a larger particle id, we first interpolate an intermediate position z on the line segment spanned by $x_{i+1,succ}$ and $x_{i+1,succ+1}$ as follows:

$$a = \frac{id_{i,j} - id_{i+1,succ}}{id_{i+1,succ+1} - id_{i+1,succ}}$$
$$z = x_{i+1,succ} + a(x_{i+1,succ+1} - x_{i+1,succ}) \qquad (8)$$

We then compute the Euclidian distance between $x_{i,j}$ and z, and we use this distance as the shortest distance of $x_{i,j}$ to the time line tl_{i+1}. The distance to the preceding time line tl_{i-1} is determined analogously (see Figure 7(b)).

A new particle front that is added due to streak line refinement contains the same number of particles as the time line triggering the refinement event. As shown in Figure 8, the new front is stored as a contiguous block in the vertex buffer right before this time line. Particle positions and normal values are linearly interpolated between $x_{i,j}$ and intermediate values on tl_{i+1} as described in Equation (8).

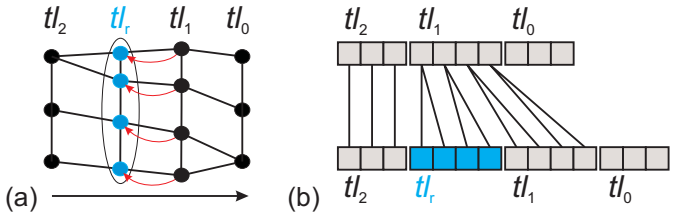

Fig. 8. Streak line refinement: (a) The time line tl_1 satisfies the refinement criterion and spawns the new time line tl_r. The corresponding changes in the vertex array buffer are illustrated in (b).

6.2 Streak Surface Triangulation and Rendering

To render the surface as a watertight triangle mesh a final pass is executed. Prior to triangulation, a geometry shader validates the connectivity and updates the neighborhood for all particles residing on time lines whose adjacent time lines have been removed due to streak line refinement.

A closed surface representation is generated by using the particle connectivity to compute a triangulation of adjacent time lines. For each particle that is sent to the rendering pipeline the geometry shader creates two triangles and appends them to the output stream. The first triangle is spanned by the vertex $x_{i,j}$, its local right neighbor $x_{i,j+1}$, and its successor on the time line tl_{i+1}. The second triangle consists of the vertex $x_{i,j}$, its local left neighbor $x_{i,j-1}$, and its predecessor on the time line tl_{i-1}. Since this process is performed for every vertex, a watertight surface is generated. Figure 10 illustrates this triangulation process.

Triangles containing an edge that was marked invalid due to the criterion in Equation (6) are excluded from the output stream. Note that particles on the surface border ($i=0 \vee i=n \vee j=0 \vee j=m$) contain at least one invalid neighbor, such that the corresponding triangle is also excluded from the stream out.

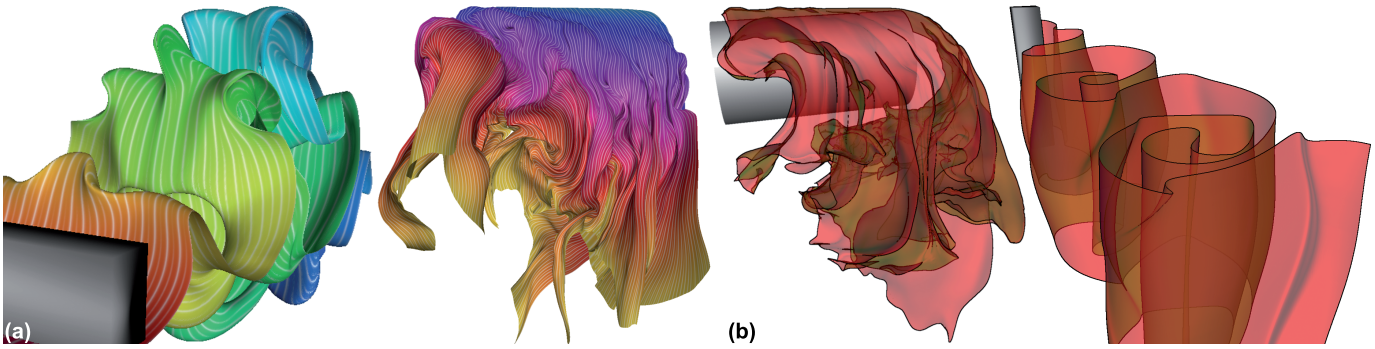

Fig. 9. (a) The surface parametrization, consisting of time line and particle ids, is used to color the surfaces with stream lines. (b) Visualization of transparent streak surfaces by application of depth peeling.

Once the triangulation has been generated it can be rendered directly using various rendering styles. Since the tupels $i, id_{i,j}$ that are stored for each particle correspond to a surface parametrization, they can be used to texture the streak surface. In Figure 9 (a) this parametrization was used to color the surface with streak lines. In (b) depth peeling was applied to create a semi transparent visualization of the streak surface, which was combined with image based edge detection to amplify sharp features on the streak surface.

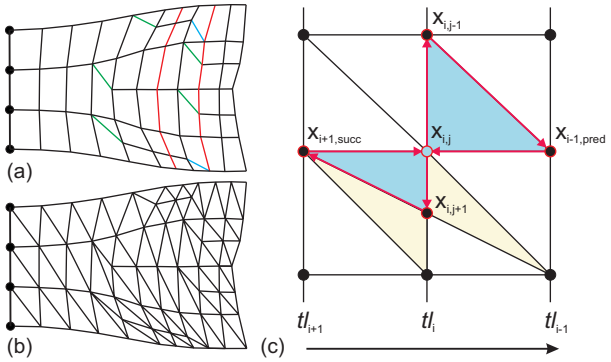

Fig. 10. Streak surface triangulation: (a) Vertex connectivity and refinement events: Green edges indicate vertex splitting, blue edges indicate vertex merging, and red edges indicate streak line refinement. The resulting triangulation is shown in (b). In (c) the two triangles generated by the vertex $\mathbf{x}_{i,j}$ are colored blue. Colored yellow are the two triangles generated by vertex $\mathbf{x}_{i,j+1}$.

6.3 GPU Implementation

In mesh-based streak surface generation, analogously to the particles, each time line gets assigned a unique id and a counter indicating its refinement depth. For a time line released at time $t_i = t_0 + i\,\Delta t$ the id is set to 0 and incremented by 2^d in each time step. New time lines that are added due to a refinement event adapt this key in the same way as it was described for particles before. This key is then used by the particles on each time line to index into a 1D array—having as many entries as there can be time lines—that stores time line specific information. On the GPU, this array is realized as 2D texture to avoid texture resolution limits. Figure 11 shows the content of this array for a set of time lines before (a) and after one integration step (b).

Furthermore, each particle carries two additional offsets, which are used in combination with the time line id to determine the id of adjacent time lines. These offsets are initialized with 2^d and changed accordingly whenever streak line refinement adds/removes an adjacent time line. Figure 11 (c) depicts the change of offsets due to the refinement of time line tl_{i-1}.

Parallel to the buffer update during time line refinement (see 6.1.1), we bind a texture render target to the rendering pipeline and rasterize

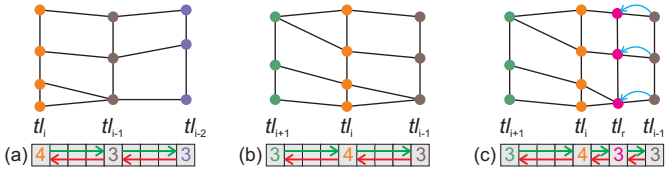

Fig. 11. Three time lines of nine possible time lines exist. The number of vertices on each time line is stored in corresponding entries in a 1D array. Red/green arrows indicate the offsets every time line stores to its neighbors in the array. (a) Array indices before and (b) after one integration step. (c) Offsets to adjacent time lines change due to streak line refinement.

each particle as a point primitive into the texel indexed by the respective time line id. By using additive blending, the number of particles residing on each time line is obtained and can be accessed by the particles during the connectivity update and streak line refinement passes.

In the connectivity update pass every particle writes to a second array its absolute position in the vertex array buffer in the same way. By using a maximum blend operator, the second array contains for each time line the absolute vertex buffer position of the last particle on the respective time line. These values are needed in the streak line refinement pass to append all particles on a new time line as contiguous block to the vertex array buffer. Additionally, for each particle its distance to neighboring time lines is computed during the connectivity update, and the maximum distance to each adjacent time line is stored separately in an additional texture target. These values are then used in streak line refinement to evaluate the refinement criterion.

To find successors for particles on tl_i, the connectivity pass has to search in an interval containing as much elements as there are on tl_i and tl_{i+1} because the absolute position of a particle in its respective memory block b_i is not yet known. The tuple of time line and particle ids forms a strictly monotonic increasing key over the whole vertex array buffer that is used in a binary search in the interval to the left of a particle to find its successor. The predecessor is determined analogously.

In streak line refinement, new time lines are appended as contiguous blocks to the vertex array buffer. Each particle on a time line tl_i that triggered a streak line refinement decides based on its absolute position in the memory block b_i whether it should contribute two particles to the new time line or account for two particles of tl_i.

During both refinement passes, we do not remove neighboring particles/time lines at once. If multiple adjacent particles satisfy the coarsening criterion in the time line refinement pass, we remove only every second particle. The decision which particle will be removed is based on a modulo criterion applied to the tuple of particle id and depth counter. Analogously we do not remove adjacent time lines at once during the streak line refinement pass.

7 Results and Discussion

To validate our methods for GPU-based streak surface generation and rendering, we have realized the developed algorithms with the DirectX 10 API. Performance tests were carried out on a 2.66 GHz Core 2 Duo processor, equipped with a NVIDIA GTX280 graphics card with 1024 MB local video memory. Results were rendered to a 2560×1600 viewport. In all of our experiments an explicit fourth-order Runge-Kutta scheme at single floating point precision was used for numerical particle integration. Detailed timings for interactive streak surface construction and rendering are given below. For the efficient handling of time-varying flow fields on the GPU we utilize the two-stage streaming approach that was presented in [5].

We have tested the proposed approaches in two real-world scenarios consisting of time dependent 3D simulation results given on Cartesian grids:

- *Flow around a square cylinder:* Result of a DNS simulation of the three-dimensional flow around a square cylinder between parallel walls at $\mathrm{Re} = 220.0$ [12]. The simulation was carried out on an unstructured tetrahedral grid. We used a resampled version with a uniform grid resolution of $192 \times 64 \times 48$ and a temporal resolution of 102 steps in the course of our work.

- *Flow around a cylinder*: Large eddy simulation of an incompressible unsteady turbulent flow around a wall-mounted cylinder at $\mathrm{Re} = 200.000$ [6]. 22 time steps were simulated. The size of the data grid is $256 \times 128 \times 128$.

7.1 Performance

Representative timings in milliseconds (ms) for integration, adaptive refinement and rendering using the patch-based approach are listed in Table 1. Values in the first three columns show the number of patches n, the maximum particle lifetime m, and the refinement depth d. The values in column labeled *Pts* show the average number of surface patches. Column *Int* contains timings for integration and refinement, *Vis* for the rendering of the resulting surface, and column *Ttl* the total amount of time required for the construction of the adaptively refined streak surface and subsequent rendering. As some of the presented settings require buffers larger than the available GPU memory, we used static buffer sizes independent of the chosen parameters but increased their size in case of a buffer overflow and restarted the performance test.

n	m	d	Pts	Int	Vis	Ttl
50	500	4	40k	1.3	5.0	7.5
50	500	8	55k	1.8	6.6	9.9
100	1000	4	128k	3.6	5.4	10.5
100	1000	8	167k	4.7	7.0	13.5
200	1000	4	365k	9.4	9.9	20.6
200	1000	8	545k	13.9	14.6	29.9
400	1000	4	1.28M	28.5	30.0	59.7
400	1000	8	2.08M	48.8	49.8	99.8

Table 1. Performance statistics for the patch-based streak surface generation and rendering. Timing statistics in milliseconds are listed in columns 5-7. Even for more than one million surface patches the streak surface construction and rendering took less than 60 milliseconds.

Timing statistics for mesh-based streak surface generation and rendering are given in Table 2. The maximum depth for both refinement strategies were equally set to d. Values in the column labeled *Pts* contain the number of surface particles, column *Int* and *Con* show the times that were required for particle integration including time-line refinement and the connectivity update, respectively. Column *Slr* gives timings for streak line refinement and column *Vis* gives the time required for surface triangulation and rendering. Finally, column *Ttl* shows the total time required for the construction and rendering of the adaptively refined triangular mesh.

n	m	d	Pts	Int	Con	Slr	Vis	Ttl
30	500	4	49k	2.4	1.1	0.9	2.2	8.1
30	500	8	64k	3.1	1.4	1.0	2.8	9.5
50	500	4	116k	5.2	2.3	1.8	4.6	14.8
50	500	8	188k	8.0	3.8	2.5	7.3	22.3
100	1000	4	295k	12.0	5.8	3.8	11.2	34.2
100	1000	8	351k	14.1	7.1	4.4	13.3	39.8
200	1000	4	952k	36.7	20.3	11.3	35.5	105.0
200	1000	8	1.18M	46.0	25.7	14.1	44.7	132.2

Table 2. Performance statistics for the mesh-based streak surface generation and rendering. Columns 5-9 present timings in milliseconds. The construction and rendering of a mesh-based streak surface consisting of more than 350K particles took less than 40 milliseconds.

7.2 Quality Comparison

To compare the visual quality, we have used both approaches to generate the same streak surfaces at comparable sample densities. As shown in Figure 12, the patch-based approach suffers from artifacts that are common to point-splatting approaches. In particular, the patch alignment in regions of high curvature tends to produce rather rough surface structures. While increasing the patch areas can cure those artifacts, it tampers with the actual extracted streak surface and requires to increase the bias of the attribute pass. This, however, in turn leads to the accumulation of incoherent surface parts. In addition, blending of overlapping patch attributes tends to blur high frequent surface features. The mesh based approach, on the other hand, avoids all these problems and delivers a closed surface representation that can be rendered using standard polygon rasterization. Sharp features and high frequent geometric details are preserved and the interpolation of vertex normals results in a smooth illumination.

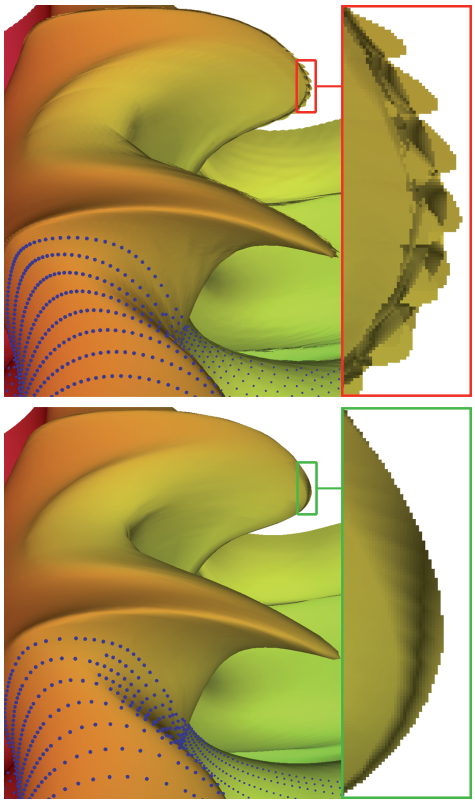

Fig. 12. This image shows the same streak surface that was generated using the patch-based (top) and the mesh-based (bottom) approaches at comparable sample density. While patch-based splatting results in artifacts and blurring at fine surface details and silhouettes, the mesh-based approach yields a high-quality surface representation.

To achieve comparable quality, the patch-based approach requires a significantly higher sampling density. The following plot shows the sample density of both approaches, extracting streak surfaces at comparable visual quality.

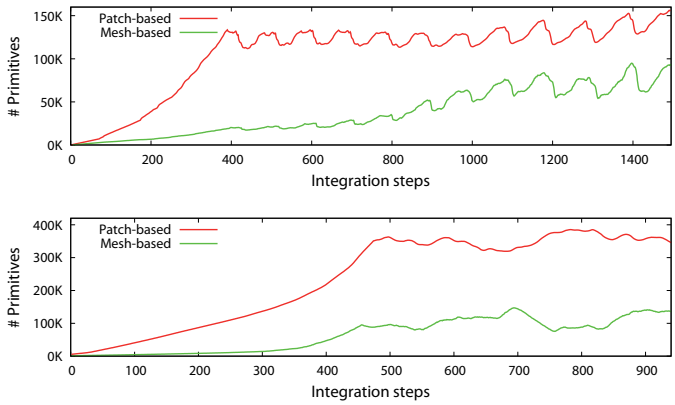

Fig. 13. The plots show the sample density of both approaches during streak surfaces generation at comparable visual quality. Top: Statistics for the square cylinder data set. Bottom: Statistics for the LES data set.

7.3 Conclusion

In this paper, we have presented two real-time techniques for constructing and rendering adaptively refined streak surfaces on the GPU. The patch-based approach performs particle integration and adaptive refinement in one step. In the proposed setup we tried to minimize additional complexity regarding the refinement criterion, integration expense and the maximum output performed by the geometry shader, resulting in real time performance even for huge amounts of patches traced in parallel. We also presented visualization methods for this representation by adapting point-splatting techniques to render the loose patch set as closed surface.

The mesh-based approach addresses the increased integration expense by introducing connectivity information between the surface samples. This does not only remove redundant particle integration but also allows the application of more sophisticated adaption criteria as well as coarsening the particle set during surface construction. On that account, the mesh-based approach delivers visually comparable streak surfaces to the patch-based approach with a much smaller set of surface samples. Furthermore, the closed surface representation can be rendered outright and a multitude of rendering styles can be applied efficiently.

We are aware of the fact that the current triangulation can lead to distorted triangles in highly diverging flow regions or areas of high shear strain between adjacent time lines. Thus, we will investigate alternative triangulation methods in the near future.

As the proposed techniques have only been validated for flow fields on cartesian grids, we will investigate their performance on unstructured grids in the near future. Since none of our techniques inherently depend on a uniform grid structure, we expect this implementation to be straightforward. Yet, as particle tracing in unstructured grids comes at an additional expense due to more complex operations for point location and interpolation, it will most likely make the mesh-based approach the favorable technique not only in terms of quality of the resulting image but also performance wise.

Acknowledgments

The authors wish to thank Simone Camarri and co-workers for providing the square cylinder data set as well as Tino Weinkauf for providing the downsampled version. Furthermore we wish to thank Octavian Frederich et al. for providing the second time-dependent data set.

References

[1] D. Blythe. The Direct3D 10 system. *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, 2006.

[2] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, 0:17–141, 2005.

[3] R. W. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-Time Out-of-Core Visualization of Particle Traces. In *IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics (PVG2001)*, pages 45–50, 2001.

[4] K. Bürger, P. Kondratieva, J. Krüger, and R. Westermann. Importance-Driven Particle Techniques for Flow Visualization. In *Proceedings of IEEE VGTC Pacific Visualization Symposium 2008*, 2008.

[5] K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, and R. Westermann. Interactive Visual Exploration of Instationary 3D-Flows. In *Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, pages 251–258, 2007.

[6] O. Frederich, E. Wassen, and F. Thiele. Flow Simulation around a Finite Cylinder on Massively Parallel Computer Architecture. In *International Conference on Parallel Computational Fluid Dynamics*, pages 85–93, 2005.

[7] C. Garth, H. Krishnan, X. Tricoche, T. Bobach, and K. I. Joy. Generation of Accurate Integral Surfaces in Time-Dependent Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1404–1411, 2008.

[8] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann. Surface Techniques for Vortex Visualization. In *Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 155–164, 2004.

[9] J. P. M. Hultquist. Constructing stream surfaces in steady 3D vector fields. In *VIS '92: Proceedings of the 3rd Conference on Visualization '92*, pages 171–178, 1992.

[10] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[11] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. Feature Extraction and Visualisation of Flow Fields. In *Eurographics 2002 State of the Art Reports*, pages 69–100. 2002.

[12] S.Camarri, M. Salvetti, M. Buffoni, and A.Iollo. Simulation of the three-dimensional flow around a square cylinder between parallel walls at moderate Reynolds numbers. In *Proceedings of XVII Congresso di Meccanica Teorica ed Applicata*, 2005.

[13] T. Schafhitzel, E. Tejada, D. Weiskopf, and T. Ertl. Point-based Stream Surfaces and Path Surfaces. In *Proceedings of Graphics Interface 2007*, pages 289–296, 2007.

[14] G. Scheuermann, T. Bobach, H. H. K. Mahrous, B. Hamann, K. Joy, and W. Kollmann. A Tetrahedra-based Stream Surface Algorithm. In *VIS '01: Proceedings of the Conference on Visualization '01*, pages 151–158, 2001.

[15] M. Schirski, C. Bischof, and T. Kuhlen. Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 153–160, 2006.

[16] D. Schneider, A. Wiebel, and G. Scheuermann. Smooth Stream Surfaces of Fourth Order Precision. In *Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, pages 871–878, 2009.

[17] H.-W. Shen, G.-S. Li, and U. D. Bordoloi. Interactive Visualization of Three-Dimensional Vector Fields with Flexible Appearance Control. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):434–445, 2004.

[18] D. Stalling. *Fast Texture-based Algorithms for Vector Field Visualization*. PhD thesis, FU Berlin, Department of Mathematics and Computer Science, 1998.

[19] J. J. van Wijk. Implicit Stream Surfaces. In *VIS '93: Proceedings of the 4th Conference on Visualization '93*, pages 245–252, 1993.

[20] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke Surfaces: An Interactive Flow Visualization Technique Inspired by Real-World Flow Experiments. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1396–1403, 2008.