Image-Based Streamsurfaces

Gustavo M. Machado, Filip Sadlo, Thomas Ertl
Visualization Research Center, University of Stuttgart, Germany

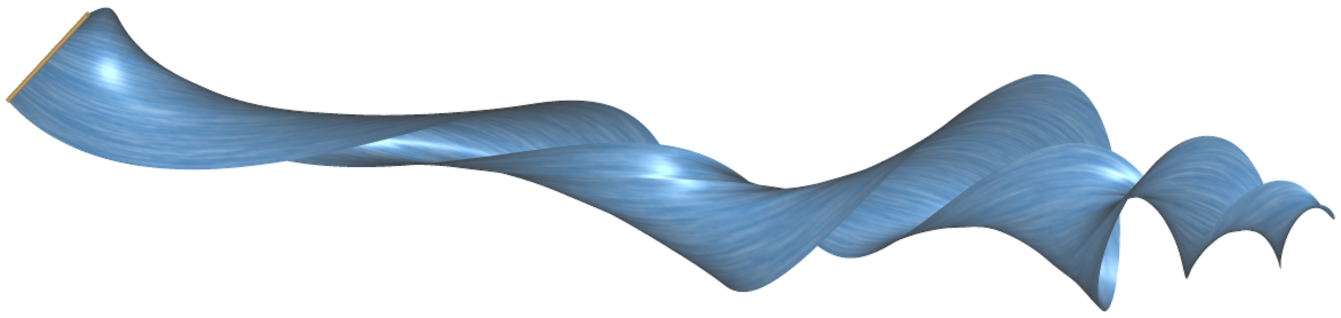

Fig. 1. Visualization of vortical flow behavior in Buoyant Flow dataset with our image-based streamsurface technique, rendered with dense streamlines. The seed curve (yellow) on the left defines the streamsurface (blue). Line integral convolution helps revealing intrinsic flow behavior. In our experiments, this example rendered at 7.5 FPS for a resolution of 900×900 pixels.

***Abstract*—Streamsurfaces are of fundamental importance to visualization of flows. Among other features, they offer strong capabilities in revealing flow behavior (e.g., in the vicinity of vortices), and are an essential tool for the computation of 2D separatrices in vector field topology. Computing streamsurfaces is, however, typically expensive due to the difficult triangulation involved, in particular when triangle sizes are kept in the order of the size of a pixel. We investigate image-based approaches for rendering streamsurfaces without triangulation, and propose a new technique that renders them by dense streamlines. Although our technique does not perform triangulation, it does not depend on user parametrization to avoid noticeable gaps. Our GPU-based implementation shows that our technique provides interactive frame rates and low memory usage in practical applications. We also show that previous texture-based flow visualization approaches can be integrated with our method, for example, for the visualization of flow direction with line integral convolution.**

***Keywords*-Streamsurfaces; Flow visualization; Image-based rendering.**

I. Introduction

Streamlines are integral curves everywhere tangent to a steady vector field or an isolated time step of time-dependent fields, and are defined by seed points, i.e., exactly one streamline passes through a seed point. A streamsurface is an extended concept and consists of an infinite set of streamlines that pass through a given seed curve. Beyond showing direction, streamsurfaces provide insights on flow behavior by exhibiting structures that reveal properties of, e.g., torsion or vorticity. In traditional vector field topology, streamsurfaces are an essential tool for computing two-dimensional separatrices that arise from saddle-type critical points or saddle-type periodic orbits. Moreover, streamsurfaces can be used as building blocks for other methods [1], [2].

Streamline computation is typically performed by explicit integration of a vector field, which represents a straightforward task. Streamsurface computation, on the other hand, may lead to expensive computational effort and possibly considerable memory usage due to the introduced connectivity construction. Most of the previous techniques for computing streamsurfaces focus on finding a triangle or quadrilateral mesh approximation to the surface [3]–[9]. Such a triangle mesh at hand can feature some advantages, like presenting fast rendering after extraction and supporting subsequent processing. The extraction of high quality meshes is, however, still challenging and computationally expensive. Moreover, extracting and storing such surfaces may be costly and application-dependent, and searching for specific streamsurfaces can turn out to be a tedious task. Therefore, for many situations, it is desirable to employ a fast method for computing streamsurfaces in direct visualization.

Aiming to avoid the triangulation cost, Schafhitzel et al. [10] presented a point-based approach that computes a set of particles along the streamsurfaces for rendering, which, however, demands user-assisted parametrization to prevent noticeable gaps. Their approach precomputes all particles for further rendering, and, therefore, may require high memory, in particular in complex flow configurations.

We present an image-based technique that extracts and renders streamsurfaces without triangulation. To this end, our approach renders dense streamlines directly during explicit integration of the surface. Its data structure keeps in memory only data referring to the streamsurface fronts of the last two time steps of integration. Our technique does not require user parametrization to prevent noticeable gaps, allowing for free navigation. Our GPU-based implementation shows that our method attains interactive frame rates for reasonable screen resolutions and reasonable flow complexity. We also incorporate line integral convolution (LIC) [11], which demonstrates the technique's suitability with existing texture-based flow visualization methods on surfaces.

II. Related Work

Existing streamsurface extraction techniques can be classified into *explicit* and *implicit*. In the former, streamsurfaces are computed by starting streamlines at seed curves and performing explicit integration from these curves along the vector field. Triangle-based approximation surfaces that cover the visited area are typically used for further rendering.

The seminal technique for explicit computation of streamsurfaces is the one presented by Hultquist [3]. It constructs triangulated streamsurfaces by performing front advancement, which is held balanced to keep the front, as much as possible, orthogonal to the velocity field. To maintain uniform front density, Hultquist's approach inserts and removes streamlines at the front to adapt to flow divergence and convergence, respectively. This approach also detects internal flow deviations, which can cause very high divergence (e.g., when consecutive samples head in almost opposite directions), and splits the front at that point. Although this technique is straightforward, it faces some difficulties in generating high-quality meshes efficiently for complex flow scenarios.

Aiming to overcome the limitations of Hultquist's technique, some further explicit approaches were proposed [5], [7], [9]. Garth et al. [5] divide the method into two stages to obtain accurate streamsurfaces as well as other types of integral surfaces (such as pathsurfaces, streaksurfaces, and timesurfaces). First, they compute a surface approximation by a series of accurate fronts, i.e., time lines, and later they use these to generate the triangulation for rendering. Krishnan et al. [7] also decouple streamline integration from mesh generation to expoit systems with multiple processors like clusters for the generation of streaksurfaces and timesurfaces. McLoughlin et al. [6] compute quad-based streamsurfaces and pathsurfaces by adapting the speed of front advancement. They detect rotational flow behavior locally by identifying shear in the produced quadrilaterals at each front. Although, this approach achieves reasonable performance, it cannot be fully parallelized and is not suitable with higher-order integration methods. Recently, Schulze et al. [9] presented a technique that scales the vector field such that the fronts are kept as orthogonal as possible to the flow field, which provides a robust solution for quad-based streamsurfaces extraction. This approach generates well-spaced meshes by preventing small front advancements.

To avoid the expensive triangulation involved in most of the techniques, Schafhitzel et al. [10] presented a point-based approach, which is most closely related to ours. They sample a dense set of points along the streamsurfaces and apply splatting to avoid noticeable gaps on the visualization. Note that, different from our approach, their technique depends on user-parametrization to prevent gaps, while our approach guarantees visualization without gaps. Moreover, Schafhitzel et al. store the full set of particles in a two-dimensional data structure, which limits scalability, while our technique only keeps only the positions along the last two fronts during surface integration.

Introduced by van Wijk [4], implicit streamsurfaces are computed by generating a 3D scalar field that corresponds to the advection of a given seed curve. The streamsurfaces can hence be obtained from the scalar field with simple isosurface extraction techniques like marching cubes [12]. Stöter et al. [8] extended this approach by using a four-dimensional representation, which also allows for the computation of other types of integral surfaces.

Texture-based flow visualization on streamsurfaces is an important approach for the visualization of internal flow properties. Although we do not contribute novel texture advection approaches, we demonstrate that our technique is suitable for combination with some existing approaches [13]–[15]. Weiskopf and Ertl [15] presented a hybrid physical/device space representation that allows for rendering texture advection on surfaces and features frame-to-frame coherence. As their technique addresses not only streamsurfaces, we incorporated a simplified version of their approach in ours.

III. Physical-Space Streamsurfaces

Given a steady vector field $\mathbf{v}(\mathbf{x})$, we use the parametric representation of a streamsurface as the function of position $\mathbf{x}(\rho,\tau)$ of particles ρ of a streamline over time τ. Thus, streamsurfaces are computed by solving the ordinary differential equation:

$$\frac{\partial(\mathbf{x}(\rho,\tau))}{\partial\tau} = \mathbf{v}(\mathbf{x}(\rho,\tau)). \tag{1}$$

In a continuous representation, $\rho \in [\rho_0,\rho_1]$ and $\tau \in [\tau_0,\tau_N]$. The seed curve is hence represented by $\mathbf{x}(\rho,\tau_0)$ and streamlines by $\mathbf{x}(\rho_i,\tau)$, where $0 \le i \le 1$. Furthermore, a front, which according to this representation is a timeline, is hence $\mathbf{x}(\rho,\tau_c)$ for any constant value τ_c.

The basic algorithm for our technique is based on Hultquist's approach [3]. We start by placing particles along the seed curve $\mathbf{x}(\rho,\tau_0)$, which represents the front at time $\tau_c = \tau_0$. This sampling is performed under user constraints, where the distance between consecutive samples particles in physical space must not be larger than user-defined $\delta_{\max}$ and should not be smaller than user-defined $\delta_{\min}$. We keep $\delta_{\max}$ as a required constraint and $\delta_{\min}$ as just desirable to simplify the algorithm for resampling fronts. All sample particles are then integrated by a user-defined time step ω, which gives a new front at $\mathbf{x}(\rho,\tau_0+\omega)$. Notice that, in flow visualization, the value of ω is typically chosen as a fraction of the cell size at the position of the vector field's grid, but we assume a constant time step here to simplify the notation. The algorithm then resamples the particles at $\mathbf{x}(\rho,\tau_0+\omega)$ according to $\delta_{\min}$ and $\delta_{\max}$, i.e., it removes or adds particles accordingly. This algorithm performs iteratively, as long as $\tau_0 + s\cdot\omega \le \tau_N$, where s is the number of time steps already employed. If $\tau_0 + (s+1)\cdot\omega \ge \tau_N$, we need to employ the last time step as $\omega_{last} = \tau_N - (\tau_0 + s\cdot\omega)$.

IV. Image-Based Streamsurfaces

The basic idea of our approach is to avoid the common triangulation of the surface. Hence, our streamsurfaces are rendered with just lines. To do so, we need to adapt the approach described in Sec. III in an image-based manner, involving image-space parameters that are analogous to the physical-space $\delta_{\min}$, $\delta_{\max}$, and ω. In fact, at least the physical-space parameters $\delta_{\max}$ and ω must be kept for global consistency, i.e., the image-space parameters are used only if they do not violate these physical-space constraints, because otherwise this could affect the quality of the resulting surface, for example, by employing too long step sizes or introducing error when resampling the fronts based on existing pairs of consecutive samples that are comparably far apart.

A. Image-Based Front Resampling

Resampling of the streamsurface fronts, as aforementioned, requires the two parameters $\delta_{\min}$ and $\delta_{\max}$. $\delta_{\max}$ is typically chosen such that it limits the error introduced by the insertion of particles during front resampling. As this error potentially grows during line integration, its estimation is usually difficult. Therefore, a relatively small $\delta_{\max}$ should be chosen. On the other hand, excessive sampling, caused by flow convergence, may negatively impact computational performance. Thus, $\delta_{\min}$ has to be selected sufficiently large, causing sufficient particle removal.

Our image-based resampling of the streamsurface fronts consists of finding a set of particle samples along each front, such that the distance between them is maintained according to the image-space parameters $\delta'_{\min}$ and $\delta'_{\max}$. These parameters represent distances in image space, i.e., relative to the pixel size, which we assume throughout this paper to equal to one. Assume that $\mathbf{x}'(\rho_i, \tau_t)$ is the position of $\mathbf{x}(\rho_i, \tau_t)$ in image space. The physical-space parameter $\delta_{\max}$ is used also in our image-space approach to limit error growth. Thus, we insert N evenly-spaced particle samples to the front $\mathbf{x}(\rho, \tau_t)$ between the consecutive samples $\mathbf{x}(\rho_i, \tau_t)$ and $\mathbf{x}(\rho_j, \tau_t)$, where

$$N = \left\lfloor \max\left(\frac{\Delta(\rho_i, \rho_j, \tau_t)}{\delta_{\max}}, \frac{\Delta'(\rho_i, \rho_j, \tau_t)}{\delta'_{\max}}\right)\right\rfloor, \tag{2}$$

and $\Delta(\rho_i, \rho_j, \tau_t)$ is the distance between $\mathbf{x}(\rho_i, \tau_t)$ and $\mathbf{x}(\rho_j, \tau_t)$ in physical space, and $\Delta'(\rho_i, \rho_j, \tau_t)$ is their distance in image space. Particle $\mathbf{x}(\rho_i, \tau_t)$ is removed if its image-space distance to its previous and next neighbor is smaller than $\delta'_{\min}$, if and only if this removal does not conflict with (2).

B. Image-Based Line Integration

The time step ω for line integration is typically selected such that it is small enough with respect to the integration error. The precision of higher-order integration schemes, like the fourth-order Runge-Kutta method (RK4) or the fifth-order Dormmand-Price (DOPRI5), is usually satisfactory when the time step corresponds to about $\frac{1}{5}$ of a cell size or smaller.

In an optimal situation, image-based line integration would be parametrized such that an image-based step size ω' limits the physical-space time step ω, i.e., an integration step is employed with ω, unless ω' would require a smaller step. However, mapping ω' to an optimal step size in physical space is not straightforward and in practice ω' can only be determined after the step was employed. Thus, the most straightforward approach would be to employ the step in physical space, as usual, with ω. Afterwards, one could check the step length in image space and crop it, if necessary. This approach, however, would only be suitable for explicit Euler integration, and could not be applied for higher-order schemes like RK4 or DOPRI5. For such cases, one could employ the bisection method to find the optimal step size. Even with cache coherence of currently available graphics hardware, this approach would, however, strongly impact on the technique's performance by demanding many reads from the vector field due to the iterative character of the bisection method. Moreover, it could also in worst cases not be able to find a satisfactory step size at reasonable iteration count, i.e., a step size in physical space that leads to a step that is smaller than ω' in image space. Therefore, we adopt a conservative approach that maps the length of ω' from image space directly to physical space, as if it was the length of a vector parallel to the image plane, located at the position (depth) of the integration. The actual step length employed is hence chosen as the smallest between the mapped length and ω. Thus, this approach limits the step size in image space and in physical space to a maximum of ω' and ω, respectively.

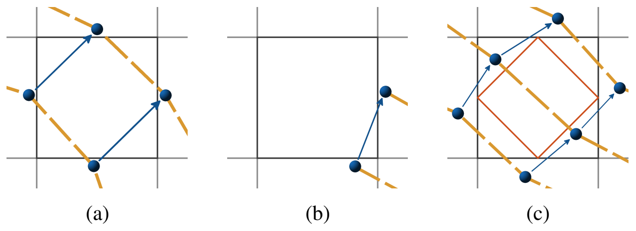

Fig. 2. Illustration of image fragments (black square). In image space, the particles (blue spheres) are sampled along fronts (yellow dashed curves) and propagated along the flow field with an integration method (blue arrows). (a) Choosing parameters $\delta'_{\max}$ and ω' greater than $\frac{1}{\sqrt{2}}$, even if smaller than the fragment's size, makes the approach for rendering particles susceptible to visible gaps. (b) Border cases, which happen when the fragment is at a border between surface and non-surface regions on the image. These cases can be neglected in our approach without impact on the quality of the rendered surfaces. (c) The diamond-exit rule, which demands that a parametrization of $\delta'_{\max}$ (or ω') also equals to $\frac{1}{\sqrt{2}}$ if rendering lines.

C. Method

With the image-based front resampling and image-based line integration at hand (Secs. IV-A and IV-B), we analyse three different approaches for image-based streamsurfaces, that differ from each other basically by the rendering method: *particle-based*, *streamline-based*, and *front-based* rendering. The techniques described in the following sections assume the OpenGL specifications for rasterization.

1) Particle-Based Rendering: This approach represents the most brute force among the ones that we analyze. As such, it is assumed that the surface rendering will be performed in a point-based fashion and for visual quality we assume points with size one, i.e., one particle sample will render at most one pixel on the screen. The aforementioned image-based methods for particle sampling along fronts and line integration, must be combined with each other in such a manner that every image fragment in the output that is covered by the streamsurface during integration must retain at least one particle sample for rendering. Hence, a robust parametrization ($\delta'_{\max}$, $\delta'_{\max}$, and ω') must be defined to fulfil this requirement.

According to the graphics API, due to a given input particle $\mathbf{x}(\rho_i, \tau_t)$, if $\mathbf{x}'(\rho_i, \tau_t)$ is inside the viewport, the fragments

$$(x,y) = (\lfloor \mathbf{x}'_x(\rho_i, \tau_t) \rfloor + 0.5, \lfloor \mathbf{x}'_y(\rho_i, \tau_t) \rfloor + 0.5) \tag{3}$$

must be rasterized. Therefore, a fragment consists in image space of a square with side length equal to one, whose area includes the left and top edges and excludes the bottom and right edges. If a particle's position in image space is inside this area, then this fragment is selected for shading. Hence, to define the values of $\delta'_{\max}$ and ω', we neglect border cases, which consist of fragments at borders between surfaces and non-surface regions in the image, due to the current integration step. Figure 2(b) illustrates such a border case. In this example, the particle (blue spheres) at open end points of the fronts (yellow dashed lines) employs one integration step (blue arrow) that crosses the fragment area without providing a sample inside it. Defining a parametrization that handles these cases demands very small values, and incorporating methods to detect these cases would represent an unnecessary effort, because the border cases can be neglected without impacting the quality of the rendered surface. For all other cases, and to avoid perceivable gaps in the rendered image, we select the side length of the inscribed square of the incircle of one fragment, i.e., $\delta'_{\max} = \omega' = \frac{1}{\sqrt{2}}$. Any greater value for any of the two parameters would make the approach susceptible to perceivable gaps as illustrated in Figure 2(a), where greater values for parameters were chosen and the fragment at the center would be missed by the approach.

2) Streamline-Based Rendering: Aiming at a method that is more efficient, we choose to render lines instead of points. By rendering streamlines we relax the integration constraints, i.e., employ line integration with just the physical-space parameter ω and apply the image-based method only for the front resampling. Now we have to ensure that whenever we render two consecutive streamlines, there are no gaps between them.

The rasterization of lines adopts the *diamond-exit* rule, which consists of a kind of Bresenham's algorithm. According to the diamond-exit rule, if the line intersects a diamond-shaped area (red square in Figure 2(c)) defined by $D = \{(x,y) \mid |x-cx| + |y-cy| < 0.5\}$, where (cx, cy) is the center of the fragment, then the fragment is selected for shading, except for line end points residing inside this area. Thus, to avoid gaps, we set $\delta'_{\max}$ to the length of one side of this diamond-shaped area, which is $\delta'_{\max} = \frac{1}{\sqrt{2}}$, as well.

3) Front-Based Rendering: Another possible approach for rendering is to render streamsurface fronts instead of streamlines. In this case, we employ image-based line integration

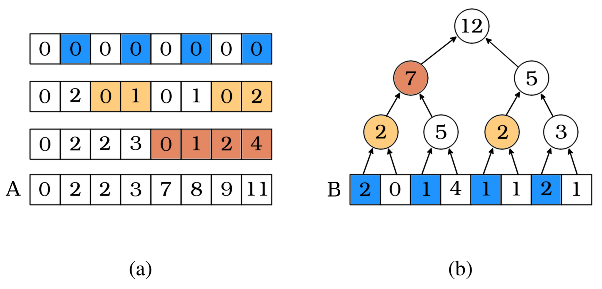

Fig. 3. Illustration of the fast resampling algorithm (Sec. V). The array B is positioned at the leaves of the tree (b). Each node of this tree is then assigned with the sum of its two child nodes. (a) The values of A during the iterations of our algorithm (top to bottom). This algorithm can run in parallel by assigning a separate thread for each one of the elements of A, with a synchronization stop between iterations.

together with physical-space front resampling. By adopting an analogous logic as for rendering streamlines (e.g., switching fronts by streamlines in Figure 2(c)) we set $\omega' = \frac{1}{\sqrt{2}}$.

D. Discussion

Typically, the parameters $\delta'_{\max}$ and ω' restrain the physical-space parameters $\delta_{\max}$ and ω. Therefore, the particle-based rendering approach is potentially very accurate. It performs very fine sampling of the surface and its accuracy is, in fact, limited by the output image's resolution. Its computation is, however, more expensive than the other two rendering approaches. It usually performs as many integration steps as the front-based rendering approach and consequently as many front resamples. The front-based rendering allows for front resampling after more than one integration step without producing visual gaps. This is, however, not recommended, because in our experiments we observed that it can affect surface quality. Moreover, the particle-based approach performs over more elements than the front-based approach along the front at each front resampling. The data structure for both approaches demands to keep the last front during integration and rendering, which is performed simultaneously in our technique. The streamline-based rendering approach, on the other hand, usually employs less steps than the other two approaches and, although each front resampling is performed over as many elements as for the particle-based rendering, the streamline-based rendering is the approach that usually computes faster. Its data structure keeps the last two fronts during integration and drawing, which is not critical, because our technique has low overall memory usage. While performing front resampling, one must consider the constraints over the values from the previous front, as well.

The value of $\delta'_{\min}$ should be selected to balance memory capacity and computation time. In our experiments, we have satisfactorily set $\delta'_{\min} = \frac{2}{3} \cdot \delta'_{\max}$.

V. Fast Front Resampling

The front resampling is a bottleneck of our algorithm. We typically resample the fronts once for each integration step, except for the front-based rendering that can perform front resampling after more integration steps. Our data structure is represented as arrays of particles to achieve faster primitive rendering. However, inserting and removing particles from arrays requires resizing the arrays in our technique. A straightforward implementation of the front resampling would basically consist of iterating over each element of the array, copying the particle data to a second array while inserting new ones when required. After each iteration, the arrays would be switched for rendering. Notice that this algorithm has complexity $O(n)$. Schafhitzel et al. [10] achieve this particle connectivity with linked lists on the GPU, which is not the case for our array representation as it impacts on our techniques' performance.

We propose an implementation that, by adopting a divide and conquer strategy for multiprocessors (the GPU in our implementation), exhibits complexity $O(\log(n))$. This algorithm splits the resampling into two steps. First, we map the output array indices for storing the new and remaining particles from a given input front sampled in the array $P_{in} = \{p_0, p_1, \ldots, p_{N-1}\}$ of size N, where p_i represent the particle samples. The data structure for this mapping is hence stored in two further arrays with same size $A = \{a_0, a_2, \ldots, a_{N-1}\}$ and $B = \{b_0, b_1, \ldots, b_{N-1}\}$. Then, we process P_{in} to store in A the indices j at which particles descendant from P_{in} will be copied to the output array P_{out}. In B, we store the number of particles descendant from each one from P_{in}. For example, if we need to insert m particles between p_i and p_{i+1} (m includes p_i), then $b_i = m$ and a_i has the index j at which the first from these particles will be written to. Hence, if $m = 0$ then the particle sample p_i is removed from the system. Afterwards, once we have P_{in}, A, and B properly set, we can build the appropriate output array P_{out} in parallel in almost constant time, due to the number of available processors, by assigning one processor for each element in P_{in}.

Finding the values of A and B starts by computing B, which is straightforward and can be computed in parallel. However, because we are performing front resampling in parallel, it is not trivial to decide which elements from P_{in} are optimal candidates for removal (e.g., in a sequence of particles that are all very close to each other in physical and/or image space). Moreover, removing all particles with closer distances than the minimum constraints ($\delta_{\min}$ and/or $\delta'_{\max}$) only with a local test can lead to large distances between consecutive particles which may conflict with the maximum constraints ($\delta_{\max}$ and/or $\delta'_{\max}$). Thus, we only remove a given particle p_i if it is at an odd location in the array, i.e., if i is odd.

For computing the array A, we take B as input. We use a binary tree T, where each node of T stores the sum of its two child nodes. The tree is built by assigning the values at the leaves with the content of B. Figure 3 illustrates an example for this algorithm where (b) illustrates T and (a) shows the values of A after each iteration from top to bottom. A is hence initialized with zeros. The algorithm starts at the leaves and iterates upwards on T. At each iteration, for each node f of T at the ascendant level with left child node l and right child node r, the elements of A that reference children of l retain their values, while the elements of A that reference children of r increment their values by the value stored at l. In Figure 3, the elements of A that should increment their values are highlighted in (a) with the same color as the values used to increment them in (b). In the end, A constains the array locations used to store particles to P_{out}. The root node of T has the total number of elements that P_{out} should have to write those values. Note that we can assign one separate thread for each element of A in this algorithm, with a synchronization step between iterations.

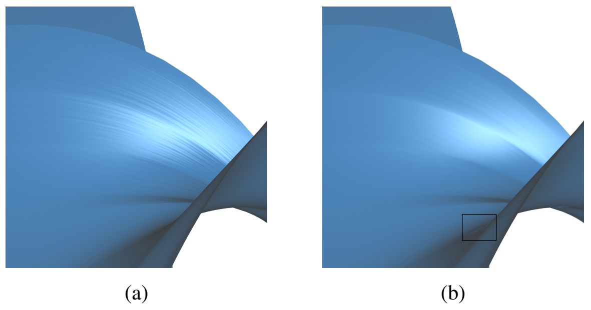
Fig. 4. Zoomed view showing part of the streamsurface from Fig.1, rendered without LIC. Results of our approach while rendering without (a) and with (b) tangent vector advection. Note that adopting the tangent vector advection makes the shading insusceptible to front deformation and thus produces better shading. However, this approach does not offer full control over the tangents computed, therefore some small artifacts may become visible, for example, light blue artifacts near torsion regions on (b) (box).

VI. Shading by Tangent Vector Advection

Although there exist many techniques for computing normal vectors from point-based surfaces [10], [16], we keep in our data structure the current and the previous front during integration and rendering. Therefore, the normal vectors can be locally computed by the cross product of the surface tangent vector at each particle position and the velocity vector at that point. For simplicity, the tangent vectors could be locally extracted as the tangent of the front at each particle's position. This method, however, would be susceptible to front deformation (Figure 4(a)).

We compute the tangent vectors $E = \{e_0, e_1, \ldots, e_{N-1}\}$ at each particle sample position p_i in P by advecting the tangent vector from the seed curve and keeping it perpendicular to the velocity field at the particle's position. We start by extracting the tangents of the seed curve, which is in our implementation the vector with direction equal to the difference between the next and the previous particles' samples position $p_{i+1} - p_{i-1}$. This vector is then adjusted, such that it remains perpendicular to the velocity field and with length equal to ε. ε is in practice an offset parameter and is set to half $\delta_{\min}$ or $\delta'_{\min}$ in physical space, accordingly. At each integration step of our technique, we advect the tangent vector by also integrating a tangent particle at position $h_i = p_i + e_i$. The tangent vector at the next front is hence selected by adjusting $h_i - p_i$ such that it again remains perpendicular to the flow field and with length equal to

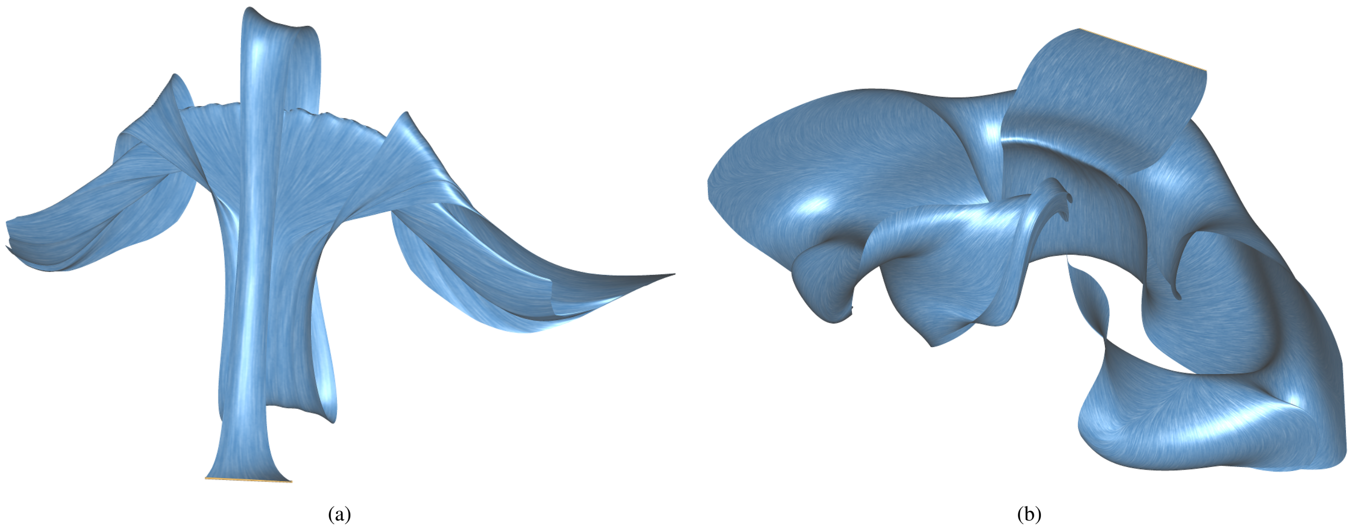

Fig. 5. Visualization of the two Buoyant Flow datasets by our image-based streamsurfaces. The streamsurfaces (blue) are defined by the seed curves (yellow). Note that in complex flow fields, a short seed curve can lead to large streamsurfaces (a) due to strong flow divergence. In (b) the seed curve at the top leads to a wide streamsurface that covers a large region of the whole dataset.

ε. This procedure is performed for each integration step and the normal direction is hence obtained as the cross product $e_i \times \mathbf{v}(p_i)$. Figure 4 compares the result of shading the surface without and with tangent vector advection in (a) and (b), respectively. This example shows that applying tangent vector advection provides more consistent normals along the surface, leading to a better shading. However, some small artifacts may become visible as this approach does, due to discretization, not offer full control over the relation between tangent vectors and the actual tangent of the computed streamsurface.

VII. Incorporating Texture Advection

Texture-based flow visualization, especially LIC, are important techniques to reveal the intrinsic structure of streamsurfaces. Many existing methods for texture-based visualization on surfaces are suitable candidates to be combined with our approach [13]–[15]. We applied LIC with a simplified version of the method by Weiskopf and Ertl [15] to demonstrate this feature with our method. For this, we attach two geometry buffers that, for each image fragment store vertex positions and normals, both represented in physical space. A 3D noise texture is then replicated in physical space for sampling. The geometry buffers and the noise texture are then used for convolution along the flow field and rendering the final pixel colors, which is further combined with the back buffer.

VIII. Results

We have implemented our technique with CUDA 5.5 and OpenGL/GLSL interoperability. We used CUDA for line integration, front resampling, LIC computation, and shading. The geometry buffer is built with GLSL by rendering the OpenGL primitives *GL_POINTS*, *GL_LINES*, and *GL_LINE_STRIP* for rendering, with the particle-based, streamline-based, and front-based approaches, respectively. Our experiments were performed on an Intel Core i5 CPU with 4×3.40 GHz and 16 GB of Memory, and a NVidia GeForce GTX 770 GPU with 4 GB.

We demonstrate our technique in the visualization of two buoyant flow computational fluid dynamics (CFD) datasets. Both datasets are discretized on structured grids with resolution of $61 \times 31 \times 61$ nodes. For the streamline integration, we employed the RK4 method with step size of $\frac{1}{5}$ of a cell size. Figures 1 and 5 show the respective results. The streamline-based visualization approach was used to generate these images and, for the visualization of intrinsic flow structure, we incorporated LIC. In Figure 1, the streamsurface reveals the flow behavior in the vicinity of a region with vortical flow, 150 time steps after the seed curve (yellow). Note that our approach provides consistent visualization of the surface with coherent shading, though it renders only lines instead of triangles. We show results for more complex streamsurfaces in Figure 5 to demonstrate our technique's robustness. The streamsurface in Figure 5(a) starts with a short seed curve that turns out to result in very long fronts after 375 time steps due to strong flow divergence. Figure 5(b) shows a very wide streamsurface that after 412 time steps covers an area about half of the whole dataset.

Table I shows the timings of our technique to the visualization of the streamsurfaces of the shown figures for an output image with resolution of 900×900 pixels. Note that the time taken by each stage of our approach is balanced among each other. Due to our GPU-based implementation, the total computation time is hence highly dependent on the number of integration steps employed (Sec. VIII-A).

Table II compares the performance of our algorithm of front resampling (Sec. V) with a single-threaded implementation. The timings for the single-threaded approach consist of com-

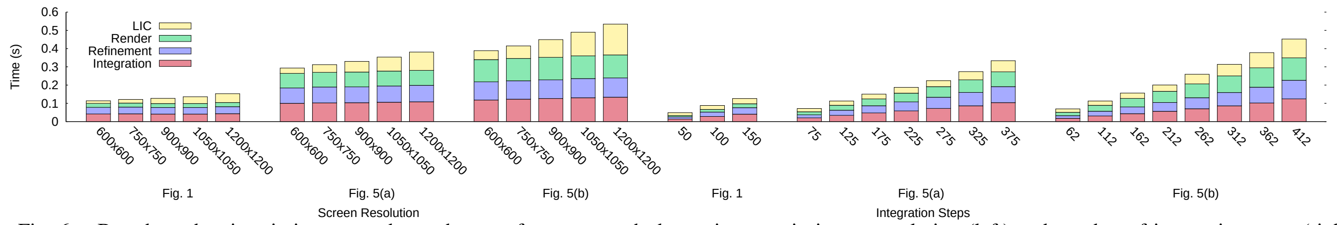

Fig. 6. Bar chart showing timings spent by each step of our approach due to increase in image resolution (left) and number of integration steps (right) for the streamsurfaces shown in Figs. 1, 5(a), and 5(b). We can observe that the image resolution does not impact the overall performance of our algorithm, while increasing the number of integration steps directly affects performance.

TABLE I
OUR STREAMLINE-BASED TECHNIQUE'S TIMINGS FOR LINE INTEGRATION, FRONT RESAMPLING, SURFACE RENDERING, TEXTURE ADVECTION, AND TOTAL TIME.

Surface	**Time[s]**				
	Integ.	**Resam.**	**Render**	**LIC**	**Total**
Fig. 1	0.0409	0.0360	0.0217	0.0284	0.1316
Fig. 5(a)	0.1034	0.0868	0.0812	0.0582	0.3475
Fig. 5(b)	0.1264	0.1022	0.1235	0.0970	0.4633

puting the arrays regarding the first step of our fast algorithm. The timings for the second step of our front resample approach are in the last column. Our approach reduces the complexity from the order of $O(n)$ in single threaded systems to $O(\log(n))$ on parallel systems. Note that, due to the performance gain in the overall front resample time, the fast front line refinement might be required to achieve interactive frame rates.

TABLE II
TIMES TO COMPUTE THE INPUT ARRAYS WITH A SINGLE THREAD AND ON THE GPU, AND THE TIME TO PERFORM FAST RESAMPLING ON THE GPU AFTER THE COMPUTATION OF THE INPUT DATA STRUCTURE.

Surface	**Time[s]**		
	Sing.	**GPU**	**Resample**
Fig. 1	0.0415	0.0134	0.0225
Fig. 5(a)	0.1200	0.0324	0.0544
Fig. 5(b)	0.2172	0.0412	0.0610

Table III compares the timings of the three image-based approaches presented in Sec. IV-C. Note that the streamline-based approach computes much faster than the other two, and is hence the technique that we recommend for practical use. This technique is fast, and as such, it is, to the best of our knowledge, the first interactive streamsurface computation technique that offers full image-space resolution without user parametrization.

TABLE III
PERFORMANCE COMPARISON FOR OUR IMAGE-BASED STREAMSURFACES WITH PARTICLE-BASED, FRONT-BASED, AND STREAMLINE-BASED RENDERING.

Surface	**Time[s]**		
	Particles	**Fronts**	**Streamlines**
Fig. 1	0.8946	0.8461	0.1316
Fig. 5(a)	1.9919	1.8905	0.3475
Fig. 5(b)	1.3932	1.2670	0.4633

All timings in Tables I, II, and III were measured as the mean time of rendering 100 frames with output image resolution of 900×900.

A. Discussion

Comparative studies on accuracy and performance of existing streamsurface extraction techniques are, for the best of our knowledge, missing in the literature nowadays. Such study would be very difficult due to the diversity and unexpectable behavior of flows and streamsurfaces. Therefore, previous works typically provide the isolated evaluation of their own results. We also remain on the systematic analysis of our algorithm compared to insights from previous works.

Let us discuss the performance of our technique by analysing the impact of each step separately. Assume a full parallelizable machine. Employing one integration step to the front in parallel (e.g., on the GPU) for each vertex sample computes in constant time. Resampling the front can be performed in logarithmic time due to our fast front resampling algorithm (Sec. V). Thus, the overall performance of our technique should be impacted linearly by the number of integration steps employed, because our method, unlike previous works, does not interfere on the front advection's speed. The performance is also impacted logarithmically by the number of front samples, which is carried by the screen resolution. The exact relation between screen resolution and front sampling is empiric as it consists of sampling curves based on the screen resolution and the physical space parametrization. Beyond that, our algorithm renders lines after each integration step, which is optimized by graphics cards, and, as long as the other steps of our method directly output the vertices and normals as arrays on the GPU's memory and yet in the correct order for rendering, the rendering step should also tend to constant time. More details on the performance of rendering should be found on the specifications of the graphics library and varies among machines' models.

The stacked bar chart in Fig. 6 shows the performance behavior of our technique with the increase to screen resolution (left) and number of integration steps (right). We can observe in Fig. 6 (left) that, even though the screen resolution increases from 600×600 to four times more pixels in 1200×1200, the impact to performance is minimal. In fact, the overall performance mainly reacts to texture advection (yellow) as to any of the steps of our algorithm (red, blue, and green).

The number of integration steps, on the other hand, impacts directly on the performance of our technique. We can note in the three clusters on Fig. 6 (right) that the regular increase of 50 steps increases alsmost constantly the time spent by every step of our technique.

Evaluating the accuracy of streamsurfaces is not trivial. In continuous representation, streamsurfaces are sets of infinite streamlines that are seeded at curves. In computational representation, on the other hand, these curves are typically represented as finite sets of connected vertices, which are therefore not continuous. Moreover, due to flow divergence, a streamsurface can rarely be completely mapped by seeding streamlines directly at the seeding curves, as one can easily reach the numerical representation's limits at the seeds and still find streamlines that diverge from each other. Thus, although the method of front resample is known to insert small errors that might lead to meaningful discrepancies, it is still an acceptable method. It is hence trivial to assume that maintaining high sampling frequencies reduces this error. In our image-based approach, these distances are managed by physical-space parameters, which assure that the resulting surfaces are qualitatively as good as most of previous approaches. In fact, our image-based parameters tend to tighten even more these constraints while the image resolution increases, which might provide better approximations.

IX. Conclusion

We presented an image-based technique for the computation of streamsurfaces. Our technique renders only lines (or points) in order to avoid the expensive triangulation involved in most of the previous techniques. Our technique is very simple to implement because it does not adopt complex criteria for dealing with flow divergence and convergence. It is also independent of extra user-defined parametrization to avoid perceivable discretization artifacts (gaps) on the visualizations. A GPU optimization was demonstrated to handle front resampling during streamsurface integration, that reduces the complexity of the front resample from $O(n)$ to $O(\log(n))$ by adopting a divide and conquer strategy for the GPU. We analysed three different approaches for image-based streamsurfaces (i.e., particle-based, streamline-based, and front-based) and concluded that due to the performance gains over the other two, the streamline-based approach is most useful for practical visualization.

As future work we propose to compare the accuracy of our image-based streamsurfaces with those from previous work. Our approach's accuracy is mainly limited by the resolution of the output image. Hence, we expect visualization on ultra-high resolution displays to provide extremely accurate results. We also propose to further investigate our image-based integration to study the behaviour of streamlines in the vicinity of very small structures like critical points, as our approach automatically performs streamline adaptation during navigation.

Acknowledgment

This work was supported by Capes-Brazil (BEX 4097/10-6) and the Collaborative Research Center SFB-TRR 75 at the University of Stuttgart.

References

[1] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann, "Surface techniques for vortex visualization," in *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization*, ser. VISSYM'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 155–164.

[2] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel, "Saddle connectors - an approach to visualizing the topological skeleton of complex 3d vector fields," in *Proc. IEEE Visualization 2003*, G. Turk, J. J. van Wijk, and R. Moorhead, Eds., Seattle, U.S.A., October 2003, pp. 225–232.

[3] J. P. M. Hultquist, "Constructing stream surfaces in steady 3d vector fields," in *Proceedings of the 3rd conference on Visualization '92*, ser. VIS '92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 171–178.

[4] J. J. van Wijk, "Implicit stream surfaces." in *IEEE Visualization*, G. M. Nielson and R. D. Bergeron, Eds. IEEE Computer Society, 1993, pp. 245–252.

[5] C. Garth, H. Krishnan, X. Tricoche, T. Tricoche, and K. I. Joy, "Generation of accurate integral surfaces in time-dependent vector fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1404–1411, 2008.

[6] T. McLoughlin, R. S. Laramee, and E. Zhang, "Easy integral surfaces: A fast, quad-based stream and path surface algorithm," in *Proceedings of the 2009 Computer Graphics International Conference*, ser. CGI '09. New York, NY, USA: ACM, 2009, pp. 73–82.

[7] H. Krishnan, C. Garth, and K. Joy, "Time and streak surfaces for flow visualization in large time-varying data sets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1267–1274, Nov. 2009.

[8] T. Stöter, T. Weinkauf, H.-P. Seidel, and H. Theisel, "Implicit integral surfaces," in *Proc. Vision, Modeling and Visualization*, Magdeburg, Germany, November 2012, pp. 127–134.

[9] M. Schulze, T. Germer, C. Rössl, and H. Theisel, "Stream surface parametrization by flow-orthogonal front lines," *Computer Graphics Forum*, vol. 31, no. 5, pp. 1725–1734, 2012.

[10] T. Schafhitzel, E. Tejada, D. Weiskopf, and T. Ertl, "Point-based stream surfaces and path surfaces," in *In Proceedings of Graphics Interface 2007*. ACM Press, 2007, pp. 289–296.

[11] B. Cabral and L. C. Leedom, "Imaging vector fields using line integral convolution," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 263–270.

[12] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169.

[13] J. J. v. Wijk, "Image based flow visualization for curved surfaces," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, ser. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 17–.

[14] R. S. Laramee, B. Jobard, and H. Hauser, "Image space based visualization of unsteady flow on surfaces," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, ser. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 18–.

[15] D. Weiskopf and T. Ertl, "A hybrid physical/device-space approach for spatio-temporally coherent interactive texture advection on curved surfaces," in *Proceedings of Graphics Interface 2004*, ser. GI '04. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2004, pp. 263–270.

[16] H. Hoppe, T. DeRose, T. Duchamp, J. Mcdonald, and W. Stuetzle, "Surface reconstruction from unorganized points," in *COMPUTER GRAPHICS (SIGGRAPH '92 PROCEEDINGS)*, 1992, pp. 71–78.