# Time-Adaptive Lines for the Interactive Visualization of Unsteady Flow Data Sets

Nicolas Cuntz[1], Albert Pritzkau[1], Andreas Kolb[1]

[1]University of Siegen, Germany

**Abstract**

*The quest for the ideal flow visualization reveals two major challenges: interactivity and accuracy. Interactivity stands for explorative capabilities and real-time control. Accuracy is a prerequisite for every professional visualization in order to provide a reliable base for analysis of a data set. Geometric flow visualization has a long tradition and comes in very different flavors. Among these, stream, path and streak lines are known to be very useful for both 2D and 3D flows. Despite their importance in practice, appropriate algorithms suited for contemporary hardware are rare. In particular, the adaptive construction of the different line types is not sufficiently studied. This work provides a profound representation and discussion of stream, path and streak lines. Two algorithms are proposed for efficiently and accurately generating these lines using modern graphics hardware. Each includes a scheme for adaptive time-stepping. The adaptivity for stream and path lines is achieved through a new processing idea we call "selective transform feedback". The adaptivity for streak lines combines adaptive time-stepping and a geometric refinement of the curve itself. Our visualization is applied, amongst others, to a data set representing a simulated typhoon. The storage as a set of 3D textures requires special attention. Both algorithms explicitly support this storage, as well as the use of precomputed adaptivity information.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation, I.3.1 [Computer Graphics]: Parallel Processing

## 1. Introduction

Flow visualization has the potential to greatly enhance the understanding of complex flow data sets in a wide range of disciplines. Mostly, flow data is generated by numerical simulation (i.e. computational fluid dynamics) or by experimental techniques such as particle imaging velocimetry. Visual exploration is an important tool for analyzing the results. Typical application areas include climate research, where general circulation models are used to describe atmospheric and oceanic phenomena, as well as the aerospace, automotive industries, and chemical engineering.

State-of-the-art flow visualization can be classified in three main categories: Direct, texture-based and geometric visualization [Lar04]. Direct visualization typically uses glyphs, e.g. arrows, which cover an overall picture of the flow. Flow information can be coded as a color or shape attribute. Texture-based visualization offers a dense representation of the flow by advecting planar or volumetric grids. A widely used example is line integral convolution (LIC) [CL93], which applies convolution kernels to noise textures. Geometric visualization evolves geometric objects like particles, lines, surfaces and volumes by integration along the

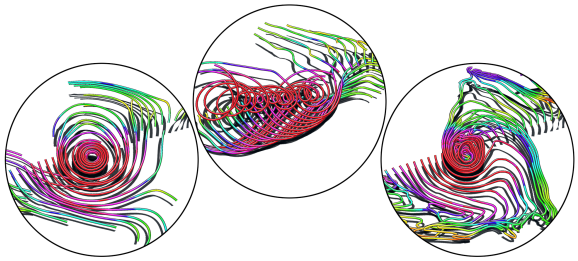

**Figure 1:** *Time-adaptive stream, path and streak lines (from left to right) in an unsteady typhoon flow (data set: courtesy of DKRZ Hamburg). All images are taken at same time. The rainbow colors stand for the velocity magnitude (red: high velocity).*

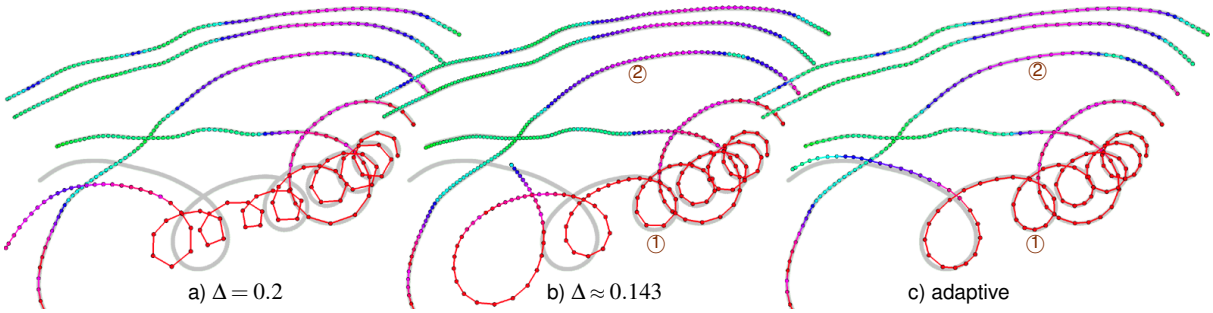


**Figure 2:** *Five path lines without adaptive time-stepping: time step* $\Delta = 0.2$*, 66.22 FPS, 385 segments (a),* $\Delta \approx 0.143$*, 63.96 FPS, 540 segments (b), and with adaptive time-stepping: 66.23 FPS, 355 segments (c). The gray ground-truth curve has been generated using time step* $\Delta = 0.02$*.*

flow field. Stream lines consist of the paths formed by particles traced while halting the flow, thus reflecting the flow at one fixed point in time. A path line shows how the flow is changing in time by tracing particles within an unsteady velocity field. Streak lines differ from the other line types in the fact that the whole line is moved instead of tracing one particle. They are natural in the sense that they frequently occur in the real world. For example, injecting a medium within a fluid (smoke in the wind) yields a streak line.

Typically, the flow line equations are solved by applying a high order integration like Runge-Kutta 2 or 4 (RK). This kind of numerical computation is based on discrete time steps. Simple strategies use a constant time step (see Fig. 2a & b), more sophisticated ones choose a time step that adapts to local flow characteristics (Fig. 2c). The objective of adaptive time-stepping is a reduction and adjustment of the truncation error of the integration. Ideally, this results in a finer sampling of intricate regions (Fig. 2: ①) and a coarser sampling in problem-free regions (Fig. 2: ②). Flow lines visualized by connecting linear segments eminently benefit from time-adaptivity, as sharp bends can be avoided.

Adaptive time-stepping can be applied directly to stream and path lines. Unfortunately, streak lines suffer from another problem: Discrete sampling is emphasized during evolution. This means that injected points will diverge according to flow divergence, resulting in sharp bends and overpopulated regions like in Fig. 3a.

In the recent past, graphics hardware has evolved to a platform that provides powerful capabilities not only in game industry but also in scientific areas. The extended programmability and parallel architecture of modern graphics processing units (GPUs) is especially useful for computationally expensive applications. Today, the trend goes from a specialized graphics processor to a general parallel co-processor. This development is called general purpose GPU (GPGPU) [OLG*05, Buc05, Har05], and it makes the GPU the ideal hardware for visualization systems which require a heavy computational background.

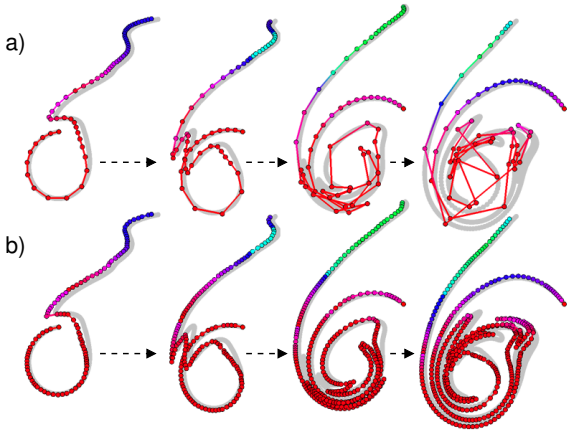


**Figure 3:** *Stop-motion of a streak line without (a) and with refinement (b). Frequency of the injection:* $\Delta = 0.2$*, both* $>$ *400 FPS. The gray ground-truth curve has been generated using time step* $\Delta = 0.02$*.*

This paper discusses the visualization of adaptive flow lines using up-to-date graphics hardware. The presented approach includes the following contributions:

- A discussion of stream, path and streak lines is provided.
- Two algorithms are proposed for efficiently generating lines using geometry shaders. In this context, we present a new processing idea we call *selective transform feedback*. Our approach includes direct support for time-adaptive stream and path lines.
- A refinement scheme for geometrically adaptive streak lines is proposed. It avoids the divergence problems shown in Fig. 3a by adding or removing points during evolution (Fig. 3b). The introduction of new points helps to maintain a balanced curve progression.
- The asynchronous flow data access required by the parallel time-adaptive particle tracing is explicitly addressed

in our algorithms. Consequently, our approach supports texture-based unsteady flow data sets.
- A qualitative evaluation of both a complex data set and analytical examples is provided.
- The approach is applied to flow data sets which do not fit entirely into GPU memory. We show that asynchronous texture transfer results in interactive frame rates even for those data sets.

## 2. Related Work

**Flow-vis** – Flow visualization is a well-studied topic (see [LHD*04] for one of several extensive overviews). Most of the literature is related to dense representations, particularly LIC and similar texture-based techniques. Often, the visual output is designed to reveal as many characteristics of the flow as possible, e.g. using multi-dimensional transfer functions (MDTF) [PBL*04]. Accordingly, feature-based visualization [PVH*03] extracts and visualizes flow features.

Geometric methods, on the other side, are typically based on particle traces, which involves a numerical integration of the flow field. In the last years, a lot of effort has been made to exploit graphics hardware for real-time visualization. GPU-based particle systems are designed to process a very large number of particles in real-time [KSW04, KLRS04]. This idea was extended to 3D flow visualization by Krüger et al. [KKKW05], including the generation of stream lines and ribbons. No adaptive time-stepping is provided, however a visual feedback of the truncation error marks difficult areas. GPU-based particle flow has been applied to climate visualization [CKL*07]. Here, the handling of large data is made possible by asynchronously transferring subsets of the flow data to graphics memory.

**Flow lines** – The work by Park et al. [PBL*05] discusses dense seeding of stream and path lines in steady and unsteady flows. Their technique supports MDTFs, and it is suited to graphics hardware. However, no streak lines are generated, and the algorithm requires one render pass per segment. Finally, adaptive time-stepping is not handled in this work. In contrast to pure geometric approaches, Theisel visualizes the curvature of stream, path and streak lines in unsteady flows without the need of any numerical integration [The98]. This is a nice method for avoiding the difficulties in the parametrization of streak lines. Liu et al. present a strategy for 2D stream line placement [LMG06] featuring loop detection and adaptive distance control. The results show clear images where flow patterns can be easily recognized by the viewer.

**Rendering** – The rendering of flow lines using graphics hardware is very efficient. Zöckler et al. present a technique for real-time illumination that uses texture mapping capabilities of the GPU [ZSH96]. Stoll et al. propose a hybrid CPU-GPU algorithm for a stylized line rendering, that combines piecewise-quadrilateral approximations with exact tube geometries [SGS05].

**Adaptive time-stepping** – The Runge-Kutta method is a widely used high order integration scheme that is known to produce numerically stable traces [Stö95]. An extensive survey on Runge-Kutta methods is given by Cartwright et al. [CP92], including a discussion of adaptive time-stepping. Advanced state-of-the-art techniques using the theory of automatic control for the design of adaptive numerical time-stepping are surveyed by Söderlind [Söd02]. An analysis of the truncation error (due to discrete time-stepping) and the interpolation error (due to discrete flow data) for particle tracing methods is provided by Teitzel et al. [TGE97].

A different interpretation of adaptivity is used by Sanna et al. [SMA00]. They create a dense texture-based representation by adaptively seeding streak lines into the problem domain.

**Graphics hardware** – Since the shader model 4, the graphics pipeline of the GPU supports a new stage called the geometry shader. This instance transforms each input primitive (i.e. a point, line or triangle) by adding or removing vertices. The output can be a variable number of reassembled primitives. This number, however, is limited by the hardware implementation. The geometry shader applies well to algorithms which dynamically alter or create a geometry, e.g. subdivision, marching cubes, etc.

The output primitives are then propagated to the rasterization stage, or they can be redirected to a specific buffer using the so-called transform feedback (TF) extension. This transform feedback records all output primitives without the possibility to select a subset.

## 3. Theoretical Background

Assume an unsteady flow given by a 3-dimensional velocity field $\vec{\mathbf{v}} : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3$ which maps a spatial location $\mathbf{x}$ and a point in time $t$ to a velocity vector $\vec{\mathbf{v}}(\mathbf{x},t)$. We first introduce the notions of stream, path and streak lines. Then, the step-doubling approach for the adaptive time-stepping in the Runge-Kutta 4 method is discussed briefly.

### 3.1. Stream, Path and Streak Lines

The **stream line**, starting at $\mathbf{x}_0$, is defined as the trace of a particle moved in the flow at a fixed time $t$:

$$\mathbf{x}_t^{sm}(s) = \mathbf{x}_0 + \int_0^s \vec{\mathbf{v}}\left(\mathbf{x}_t^{sm}(\sigma),\, t\right)\, d\sigma \qquad (1)$$

The **path line**, starting at $\mathbf{x}_0$ and $t$, is defined as the trace of a particle moved in the flow which is changing in time:

$$\mathbf{x}_t^{p}(s) = \mathbf{x}_0 + \int_0^s \vec{\mathbf{v}}\left(\mathbf{x}_t^{p}(\sigma),\, t+\sigma\right)\, d\sigma \qquad (2)$$

The **streak line** in $\mathbf{x}_0$ and $t$ is defined as the line built by

individual traces of particles which were injected at successive points in time. These individual traces can be described by using the path line representation:

$$\begin{aligned}\mathbf{x}_t^{sk}(s) &= \mathbf{x}_{t-s}^{p}(s) \\ &= \mathbf{x}_0 + \int_0^s \vec{\mathbf{v}}\big(\mathbf{x}_{t-s}^{p}(\sigma),\, t-s+\sigma\big)\, d\sigma \end{aligned} \tag{3}$$

Fig. 4 provides a schematic illustration of this relation between a streak and a path line. According to [WTS*07], the term *generalized streak line* applies to a streak line that is built using a starting point $\mathbf{x}_0$ that changes in time.

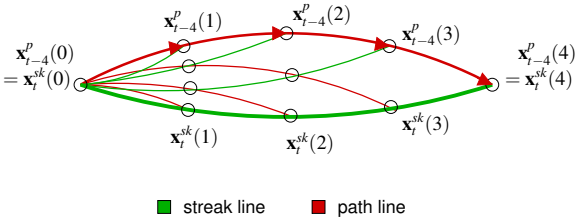


**Figure 4:** *Visual comparison of path lines and streak lines. Note that a streak line consists of the ending points of different path lines and vice versa. Flow:* $\vec{\mathbf{v}}(\mathbf{x},t) = (\cos t, -\sin t, 0)$.

Both stream and path lines (Eq. 1 and 2) can be generated in one pass. Consider a Runge-Kutta operator $\mathrm{RK}^4\big(\mathbf{x},\, t,\, \Delta\big)$ which calculates a new location according to a given starting point $\mathbf{x}$, a point in time $t$ and a time step $\Delta$. Then the following recursion provides an approximation of $\mathbf{x}_t^{sm}$:

$$\begin{aligned}\mathbf{x}_t^{sm}(0) &= \mathbf{x}_0 \\ \mathbf{x}_t^{sm}(s+\Delta) &= \mathbf{x}_t^{sm}(s) + \int_s^{s+\Delta} \vec{\mathbf{v}}\big(\mathbf{x}_t^{sm}(\sigma),\, t\big)\, d\sigma \\ &\approx \mathbf{x}_t^{sm}(s) + \mathrm{RK}^4\big(\mathbf{x}_t^{sm}(s),\, t,\, \Delta\big) \end{aligned} \tag{4}$$

(For $\mathbf{x}_t^{p}$, modify the time parameter according to Eq. 2.) Unfortunately, streak lines cannot be constructed this way due to the dependence on the integral bound $s$ in Eq. 3. However, it is possible to deduce a location $\mathbf{x}_{t+\Delta}^{sk}(s+\Delta)$ from the previous line location $\mathbf{x}_t^{sk}(s)$:

$$\begin{aligned}\mathbf{x}_t^{sk}(0) &= \mathbf{x}_0 \\ \mathbf{x}_{t+\Delta}^{sk}(s+\Delta) &= \mathbf{x}_{(t+\Delta)-(s+\Delta)}^{p}(s+\Delta) \;=\; \mathbf{x}_{t-s}^{p}(s+\Delta) \\ &= \mathbf{x}_{t-s}^{p}(s) + \int_s^{s+\Delta} \vec{\mathbf{v}}\big(\mathbf{x}_{t-s}^{p}(\sigma),\, t-s+\sigma\big)\, d\sigma \\ &= \mathbf{x}_t^{sk}(s) + \int_s^{s+\Delta} \vec{\mathbf{v}}\big(\mathbf{x}_{t-s}^{p}(\sigma),\, t-s+\sigma\big)\, d\sigma \\ &\approx \mathbf{x}_t^{sk}(s) + \mathrm{RK}^4\big(\mathbf{x}_t^{sk}(s),\, t,\, \Delta\big) \end{aligned} \tag{5}$$

### 3.2. Adaptive Time-Stepping

Adaptive time-stepping aims for reducing the truncation error and the round-off error. The first occurs when discretizing time using a fixed step-size, the latter is due to finite floating-point arithmetic and increases proportionally to the number of integration steps. Thus, a good adaptive time step is small enough to avoid truncation and large enough to avoid round-off. For the Runge-Kutta 4 method, one of the most common schemes is step-doubling. Step-doubling determines a numerically sound time step $\Delta'$ as follows:

$$\begin{aligned}\mathbf{x}_1 &= \mathrm{RK}^4\big(\mathbf{x},\, t,\, \Delta\big) \\ \mathbf{x}_2 &= \mathrm{RK}^4\big(\mathrm{RK}^4\big(\mathbf{x},\, t,\, \tfrac{\Delta}{2}\big),\, t+\tfrac{\Delta}{2},\, \tfrac{\Delta}{2}\big) \\ \Delta' &= \Delta \cdot \left(\frac{d}{\|\mathbf{x}_2 - \mathbf{x}_1\|}\right)^{\frac{1}{5}} \end{aligned}$$

The distance $\|\mathbf{x}_2 - \mathbf{x}_1\|$ is an error estimation which is asymptotically bounded by $\Delta^5$. The target distance $d$ must be chosen to produce smooth results (i.e. no sharp bends). It is advisable to avoid a denominator near zero. Also, in our implementation, $\Delta'$ is clamped to be within a reasonable interval: $\Delta' \in [10^{-3}, 10^3]$.

## 4. Time-Adaptive Lines

This section discusses the construction of stream, path and streak lines. The according algorithms are presented in GPU specific terms, because the applicability to parallel graphics hardware is one major aspect of our approach.

### 4.1. Line Construction

Our line generator relies on the geometry shader (see Sec. 2). This shader unit is able to process a stream by adding and removing elements. The stream is variable in size, thus it is ideal for representing adaptive line data.

#### 4.1.1. Stream and Path Lines

From a theoretical point of view, Eq. 4 can be used to generate stream and path lines in one pass. The input of this pass consists of a set of line seeds which is processed in parallel.

Unfortunately, the number of elements that can be generated in the geometry shader is limited to a constant number. Thus, in order to get a line of arbitrary length, we have to break the creation into several passes. This idea is sketched in Fig. 5. The selective transform feedback shown in the diagram is explained in the next paragraph.

**Selective transform feedback** – Breaking the geometry pass implies a splitting of 1. the line data used for rendering and 2. the seed information (i.e. the last point generated for each line) used as input for consecutive sub-passes. Note that the seed information is not located at the end of the overall line data, because parts of different lines are generated and

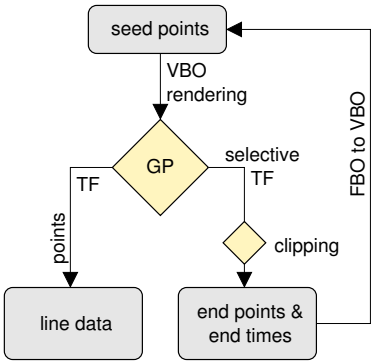


**Figure 5:** *Generating a stream (or path) line. GP: geometry program, TF: transform feedback, VBO/FBO: vertex/fragment buffer object.*

stored in an interlocked way. The geometry shader does not directly support the output into two different streams. However, it is possible to generate a transform feedback stream while simultaneously rendering primitives into the current frame buffer. The ID of the current input seed (given by `gl_PrimitveID`) determines the line that is generated in the current shader instance. Using this ID, it is possible to write the last point on a per-line basis into the frame buffer. The frame buffer content is then interpreted as new seed input for the next sub-pass. Superfluous points rendered when pushing elements into the TF are discarded by moving them outside the clipping area.

Alg. 1 lists the complete algorithm in pseudo code. In each frame, the whole line is generated. The iteration stops when no more elements are generated in the geometry shader, i.e. if the desired line length has been reached in the previous pass. Querying the number of generated elements is a subfeature of OpenGL's transform feedback.

**Algorithm 1** (stream/path line algorithm)

```
1  for each frame {
2    emit seeds p_1, ..., p_n            // chosen by user interaction
3    while query ≠ 0 {
4      query = 0
5      process each seed p ∈ {p_1, ..., p_n}:
6      geometry shader + TF:
7        s = p.s                         // using time stamp p.s
8        while s < line_length {
9          emit line point p
10         p = RK4(p, t, Δ)              // with adaptive Δ
11         s += Δ
12         query++
13       }
14       emit new seeds p_1, ..., p_n    // using selective TF
15   }
16 }
```

For path lines, replace the time parameter of the $\mathrm{RK}^4$ operator by $t+s$. Note that the number of iterations of the inner *while* loop is bounded by a constant limit specific to the hardware. The adaptive time step $\Delta$ is determined using the step-doubling approach (see Sec. 3.2) or it is read from the data set. See Sec. 4.2 for a discussion of data set specific aspects, including texture synchronization.

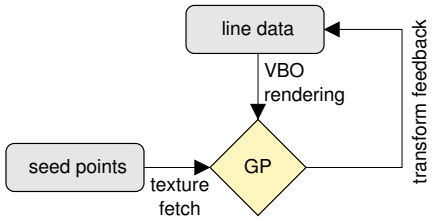


**Figure 6:** *Generating a streak line.*

### 4.1.2. Streak Lines

Streak lines are generated according to Eq. 5. The line data of the previous frame forms the input of the next frame. As the line data is used for processing, a time stamp is added to each point so that the length of the line can be controlled. The streak line approach is sketched in Fig. 6.

The main idea of the algorithm is to add new points (fetched from a seed texture) at the beginning of each line and to remove points at the end according to the desired line length. Thus, the geometry processor must distinguish between starting, inner and end points of the line. Our implementation uses special markers in the line data, which are also motivated by the rendering process. The exact structure of the line data is discussed in Sec. 5.1.

The streak line algorithm scans the stream for markers and writes the new line data directly through transform feedback:

**Algorithm 2** (streak line algorithm)

```
1  initialize line data
2  for each frame {
3    process each line data segment (p_1, p_2):
4    geometry shader + TF:
5      if  p_2 is a starting point (p_1 is marker):
6        emit new seed by texture fetching
7      if  p_1 is an end point (p_2 is marker):
8        if  t − p_1.s > line_length:  remove p_1
9        emit p_1 = RK4(p_1, t, Δ)          // possibly adaptive
10     if  p_1 and p_2 are inner points:
11       emit p_1 = RK4(p_1, t, Δ)          // possibly adaptive
12       perform refinement
13 }
```

Alg. 2 has to fulfill a fixed time step $\Delta$ in order to ensure the correct line output (see Eq. 5). In other words, when computing a point $\mathbf{x}^{sk}_{t+\Delta}(s+\Delta)$, the time step $\Delta$ is not only a time step for numerical integration but also the time step

controlling the injection of new seeds. Consequently, adaptive time-stepping cannot be applied directly to our streak line approach. Our interpretation of adaptive streak lines involves two strategies:

1. Time adaptivity: The time step $\Delta$ can be divided into several adaptive sub-steps in order to achieve a more accurate result for individual points.
2. Geometric adaptivity: In order to avoid sharp bends and oversampling (see Fig. 3a), a refinement scheme is supported in our algorithm (line 12 in Alg. 2).

The refinement step compares the distance between two adjacent points $\mathbf{p}_1, \mathbf{p}_2$ as processed in Alg. 2. If the distance is larger than a user specified threshold, then a new point is added in-between by linear interpolation of $\mathbf{p}_1$ and $\mathbf{p}_2$. If the distance is smaller than half the threshold, the point $\mathbf{p}_1$ is removed. This simple scheme has shown to be very efficient and essential for generating reasonably smooth streak lines.

### 4.2. Flow Data Set Storage & Synchronization

The data set is partitioned into several time records. Each of them is stored in a 3D texture holding the velocity data. In addition, an adaptive time step is stored at each location of each time record. This time step is computed according to the step-doubling approach (see Sec. 3.2) for every texture location within a separate pre-processing.

Binding several textures in a shader program can have a large impact on the performance: Conditional texture selection (using `if`) is significantly more expensive than fetching one single texture. This has been verified on a GeForce 8 graphics card: In an experiment, a single conditional fetch among 4 textures is about 3 times faster than among 16. The *texture array* extension supports a presumably faster solution, however it does not support arrays of 3D textures. Thus, it is essential to bind only a few 3D textures in each render pass. This restriction affects the generation of path lines.

The restriction to a small set of flow textures yields a synchronization problem: Only a limited time interval can be covered by a path line part generated in one render pass. It turns out that this restriction can be fulfilled by just adding an additional stop criterion in line 8 of Alg. 1 which ensures the right time interval. When `query` signals that no more vertices have been output, it is safe to increment the data set time interval to different textures. This adds a surrounding loop incrementing the interval until `query` remains 0 in two subsequent render passes. Note that this approach is efficient because a render pass with $\texttt{query} = 0$ is inexpensive.

### 4.3. Flow Data Sets Exceeding GPU Memory

The handling of data sets which are too large to be entirely stored in GPU memory is a task that requires special interest. Only those parts of the flow data which are momentarily necessary should be present in graphics memory. While a spatial division into bricks is rather hard to achieve for arbitrary lines, the idea of lazily transferring time records maps well to our line generator: The synchronization method presented in Sec. 4.2 ensures that the generator breaks whenever a necessary time record is missing. Before the next iteration, the framework can upload appropriate time records so that line generation is continued in the next pass.

In order to speed-up this scheme, the pixel buffer object (PBO) extension is used for asynchronously transferring the texture data using intermediate system memory. Additionally, multiple textures can be used to cache time records which are likely to be used in subsequent iterations. For path line generation involving several time records during the generation of one line, this can be of great benefit as one can see in the results (Sec. 6.3).

## 5. Line Storage and Rendering

The line renderer requires a specific storage of the line data stream. More precisely, each rendered segment $(\mathbf{p}_1, \mathbf{p}_2)$ is provided with its adjacent points, such that normals can be computed for both $\mathbf{p}_1$ and $\mathbf{p}_2$. These normals can be used for lighting and for a sophisticated rendering of tube-like geometries. Additionally, our storage layout is designed to support the splitting of lines into distinct parts (Alg. 1) and it supports end point markers (Alg. 2).

### 5.1. Line Storage

A special `BREAK` symbol identifies line parts generated by Alg. 1. This break element also contains an adjacent point used for rendering. The `SEP` symbol marks the end points of each line. It tells the renderer to terminate the geometry appropriately. The following list is an example of how a stream or path line can be represented. The $+$ operator specifies a combination of vertex and marker information.

$$\begin{aligned}
&\texttt{SEP}, \mathbf{x}_1, \ldots, \mathbf{x}_5 + \texttt{BREAK},\\
&\texttt{SEP}, \mathbf{y}_1, \ldots, \mathbf{y}_7 + \texttt{BREAK},\\
&\mathbf{x}_3 + \texttt{BREAK}, \mathbf{x}_4, \ldots, \mathbf{x}_6, \texttt{SEP},\\
&\mathbf{y}_5 + \texttt{BREAK}, \mathbf{y}_6, \ldots, \mathbf{y}_9, \texttt{SEP}
\end{aligned}$$

Fig. 7 shows how the line parts $\texttt{SEP}, \mathbf{x}_1, \ldots, \mathbf{x}_5 + \texttt{BREAK}$ and $\mathbf{x}_3 + \texttt{BREAK}, \mathbf{x}_4, \ldots, \mathbf{x}_6, \texttt{SEP}$ are combined to one geometry during rendering.

Separator elements (`BREAK` and `SEP`) are marked by setting the fourth component of the according vertex to a predefined large constant number. Streak lines require an additional identifier for distinguishing lines (see Alg. 2, line 6). This ID is stored in the starting and ending separator of each line. Streak line data also includes time stamps for all points, which are stored in a separate (texture coordinate) buffer.

Note: The exact generation of the line data requires minor modifications to Alg. 1 and Alg. 2 in order to produce a correct sequence of points, `BREAK`s and `SEP`s. This aspect has been omitted due to its implementation specific nature.

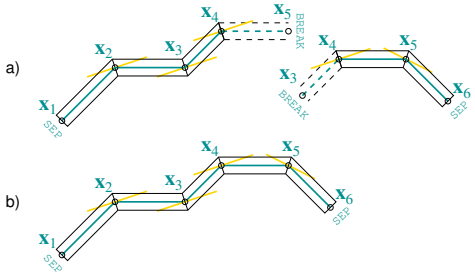


**Figure 7:** *How the renderer combines two line parts (a) to one single line (b). Due to adjacent points stored in the* BREAK *elements, the renderer can compute proper normals.*

In total, the memory consumption for the line data is asymptotically bounded by the number of segments necessary for visualizing all lines. In the implementation, a fixed limit must be given to transform feedback, which is 262,144 in our configuration. An overflow can be detected by using an OpenGL query. So the algorithm could be designed to dynamically reinstantiate the buffer if more segments are produced.

### 5.2. Rendering

The renderer directly visualizes the line data as described in Sec. 5.1. A geometry shader functionality called line adjacency is used to compute normal information. This can be used for lighting and for a non-trivial rendering of the lines. We have chosen the approach by [SGS05] which produces quadrilateral tube approximations. These quads are directly generated in the geometry shader. The shader is even able to generate explicit tube geometries in intricate situations where the view vector coincides with the tangent of a segment (see Fig. 12).

## 6. Results

In the following, our line-based flow visualization system is evaluated by presenting performance and accuracy results for different examples. The hardware used for testing is a PC with a GeForce 8800 GTX graphics card, except for the tests in Sec. 6.3. The latter have been performed on a PC with 2 GB RAM and a GeForce 9600 GT graphics card. This computer has been chosen because of its very stable multi-disk setup which was crucial for memory swapping purposes.

Our main example, the typhoon, is an unsteady flow data set (courtesy of DKRZ Hamburg) consisting of 32 time records, each stored in a $106 \times 53 \times 39$ texture. Since 2 different adaptive time steps, namely one for stream and one for path and streak lines, are stored, the total size of the data set adds up to $106 \cdot 53 \cdot 39 \cdot 5 \cdot 4$ bytes $= 134$ MB (5 components and 4 byte large floats). Slices are arranged in a non-uniform way. We solve this problem by performing a binary search

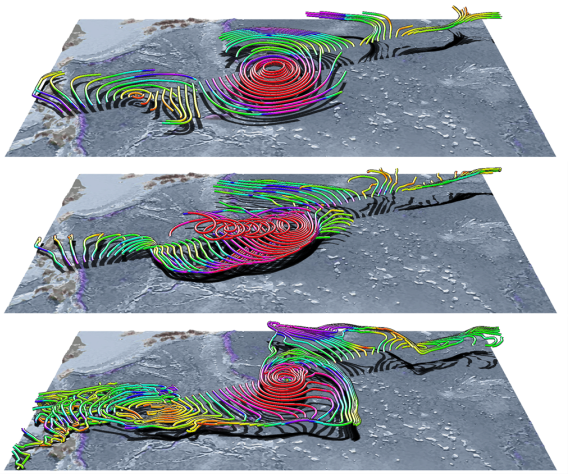

**Figure 8:** *The typhoon flow visualized using adaptive stream, path and streak lines and stylized rendering, number of lines: 64, frame rates (in FPS): 74.89, 37.38, 96.22, segments (approx.): 3780, 5900, 9500.*

in the shader in order to find the correct slice when sampling the data. Fig. 8 shows the entire visualization including an underground height field and line shadows. These Shadows have shown to improve 3D perception in practice.

Performance evaluation (see figure captions) reveals that the streak line generator is faster than the stream/path line generator when comparing the number of segments produced. This happens because the streak approach processes several segments in parallel while the stream/path approach processes several *lines* in parallel. Obviously, the number of segments is much higher than the number of lines. In order to get an impression of the quality of our approach, the typhoon example has been evaluated both visually and statistically (Sec. 6.1).

### 6.1. The Typhoon Data Set

Fig. 9 visualizes the local error provided by the step-doubling approach (compare with Sec. 3.2), i.e. a measure for the truncation error during a single integration step. It shows an interesting relation between speed and accuracy: the non-adaptive path lines are slower than the adaptive counterpart (with texture-fetched time step), yet exhibiting a much larger error in the difficult twirl region. This advantage of the time-adaptive version in both speed and accuracy is due to the number of segments, which is balanced by the time-adaptivity.

Fig. 9 also shows that in comparison to real-time step-doubling, texture-fetching the time step produces a small but visible error due to trilinear interpolation of the time step. The pre-processing storing the adaptive time-step into the

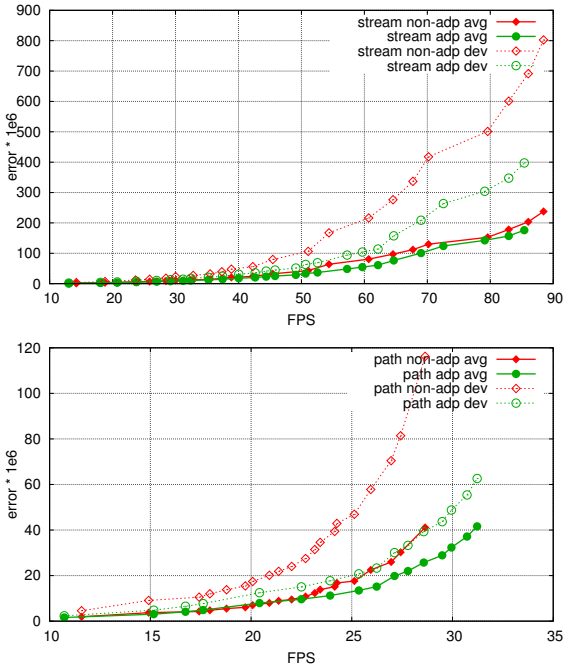


**Figure 10:** *Frame rate versus local error for the example shown in Fig. 2, with 256 stream/path lines instead of five path lines. "adp": adaptive, "avg": average, "dev": standard deviation. The error has been multiplied by* $10^6$.

textures takes 30 minutes on the CPU and approx. 10 seconds in our GPU implementation. By using our *long double*-based CPU pre-processor, a slight improvement in accuracy can be obtained.

In Fig. 10, the local step-doubling error is plotted for different frame rates and line types. For the non-adaptive lines, different values have been measured by varying the integration time step. For the adaptive lines, the step-doubling accuracy has been varied using our GPU-based pre-processor before each measurement. Both diagrams show in a very clear way that adaptive stream and path lines produce both a smaller average error as well as a smaller standard deviation of the error compared to non-adaptive counterparts at same frame rate. This result coincides with the visual analysis in Fig. 9. For streak lines, a similar analysis is hard to provide because of the different nature of adaptivity. Pure time-adaptivity (i.e. without geometric refinement) is insufficient for increasing both speed and accuracy. This is due to the fact that the performance of the streak line generator is less sensitive to a high number of line segments than for stream or path lines.

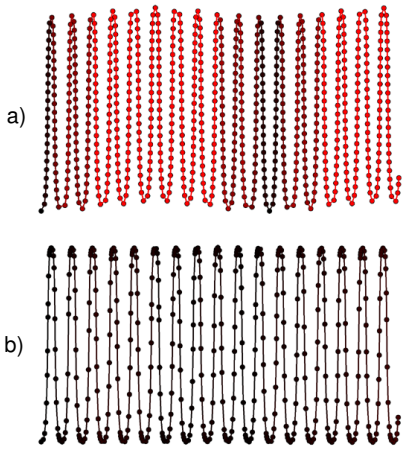


**Figure 11:** *The global error for a cubic B-spline function (marked in red). a) non-adaptive, 47.38 FPS, 776 segments, b) time-adaptive 62.16 FPS, 445 segments.*

### 6.2. Analytical Flows

In order to provide further results, two analytical flows have been evaluated. Sampling an analytical function avoids the bottle neck of texture fetching, resulting in much higher frame rates. We first consider a flow defined by the function $\vec{\mathbf{v}}(\mathbf{x},t) = (1, \sin(t) \cdot \cos(\mathbf{x}_x), 0)$. While the adaptive line generator produces approx. 10,000 texture-fetched segments in real-time (stream: 10,076 @ 22.64 FPS, path: 10,023 @ 18.66 FPS, streak: 10,079 @ 91.15 FPS), it can produce more than 10 times more segments when evaluating the analytical function (stream: 115,840 @ 30.99 FPS, path: 103,637 @ 29 FPS, streak: 100,057 @ 126.81 FPS).

In order to get a better idea of the benefit of time-adaptivity, we have evaluated another analytical flow produced by a repetitive cubic B-spline (see Fig. 11), which can be integrated analytically in order to provide an exact path line representation. Velocities are normalized in order to sharpen the difficulty near the curved extrema. A global error has been evaluated by taking the distance to exact line locations according to the time parameter of the computed line. The top (non-adaptive) line shows a large error despite the fact that more segments are rendered than in the bottom line, which adapts well to the characteristics of the flow. Note that the periodic nature of the function causes interferences in the error, thus canceling the error in certain regions.

### 6.3. Larger Flow Data Sets

In practice, one might be interested in visualizing larger flow data sets than the one presented in Sec. 6.1. In order to test the applicability of our approach to this task, various data sets of different size have been generated, all representing the same flow defined by the function $\vec{\mathbf{v}}(\mathbf{x},t) = (1, \sin(t) \cdot \cos(\mathbf{x}^x), 0)$.

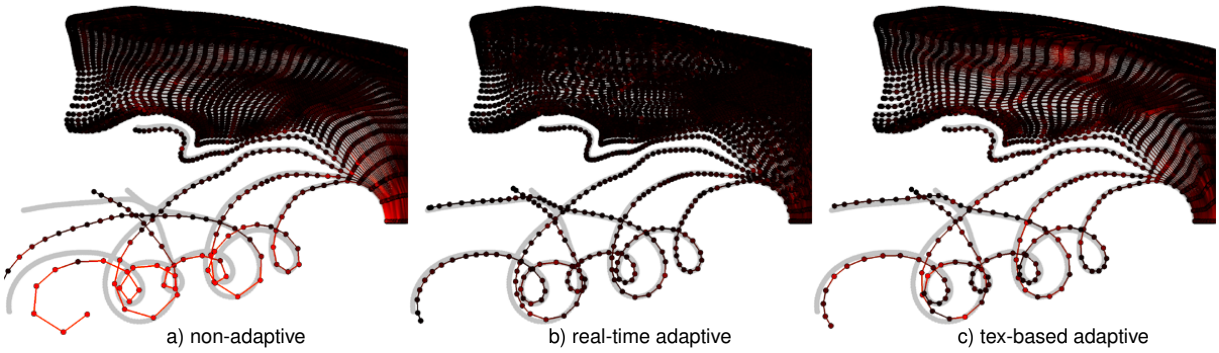


**Figure 9:** *The local step-doubling error (marked in red) for the typhoon example. Frame rates (in FPS, without error computation): 37.28, 11.94, 37.83, number of segments: 11776, 13886, 10505*

**Figure 12:** *The stylized rendering combines quadrilateral segments with explicit tube geometries (approx. 20 FPS).*

**Table 1:** *Runtime for stream/path/streak lines in a flow data set of varying resolution. In all tests, 512 lines cover nearly the whole flow volume and at least 4 time records.*

| res. | steps | size | FPS (stream/path/streak) | | |
|---|---|---|---|---|---|
| | | | normal | async | async & cached |
| $50^3$ | 100 | 238M | 40/26/44 | 39/38/44 | 39/93/44 |
| $50^3$ | 250 | 596M | 40/26/44 | 40/39/45 | 39/83/45 |
| $50^3$ | 500 | 1.2G | 40/27/44 | 39/37/45 | 39/92/44 |
| $100^3$ | 50 | 954M | 55/5/44 | 59/8/48 | 53/79/48 |
| $100^3$ | 100 | 1.9G | 58/3/44 | 74/8/47 | 69/42/47 |
| $100^3$ | 200 | 3.7G | 6/5/13 | 18/5/15 | 10/11/11 |
| $150^3$ | 10 | 644M | 37/1/34 | 48/3/40 | 23/55/31 |
| $150^3$ | 50 | 3.1G | 5/1/7 | 4/3/8 | 6/5/11 |

Notation: *res.*: resolution of one time record, *steps*: number of time records, *size*: file size of the data set (containing 2 different adaptive time-steps for stream and path/streak lines), *normal*: no use of PBOs, *async*: asynchronous texture transfer using PBOs, *async & cached*: 1 PBO and 8 instead of 3 textures in GPU memory.

Table 1 shows the timings for all tested data sets. During visualization, one time record covers ten frames. All three types of lines have been tested in different memory configurations: Besides three textures representing the current time records needed for one sinlge Runge-Kutta integration step, texture transfer has been accelerated by dedicating a PBO for asynchronous transfer (second column of the FPS results). A third configuration has been tested using a stack of 8 cached textures. This number has been chosen because eight $150^3$ textures plus PBO and the line data (approx. 32 MB in this configuration) amounts to nearly 512 MB of memory, which is the limit of the graphics card used for the tests.

The results show two interesting aspects: First, texture transfer is fairly fast for stream and streak lines. Path lines require multiple switches of the textures during one frame, as lines are covering several time records. Thus, path lines highly benefit from asynchronous transfer and caching of subsequent records. This yields interactive frame rates even for data sets which do not fit entirely in GPU memory. However, one can see that system memory is a bottleneck as soon as disk swapping is involved. While the larger data sets ($>$ 3 GB) can still be visualized with few frames per second in average, fetching necessary time records from hard disk results in unpleasant delays disturbing the animation. For handling such data sets, a more elaborate memory management or a special hardware for fast memory access would be necessary so that a smooth visualization is ensured.

## 7. Conclusion and Future Work

A method for efficiently generating and visualizing adaptive stream, path and streak lines using modern graphics hard-

ware has been presented. The adaptive time-stepping controls the truncation error, and thus a reliable visualization is obtained. The use of graphics hardware results in an interactive visualization. Due to the generic nature of the presented line generators, it should be possible to incorporate more sophisticated adaptive time-stepping methods that replace step-doubling. Similarly, streak line refinement could be improved without changing the algorithmic idea. Together with support of new graphics hardware for double precision arithmetic, this could expand the applicability of our method to areas where an even higher accuracy is necessary.

**Acknowledgements** – We would like to acknowledge the German Climate Computing Centre Hamburg (DKRZ) for providing the typhoon data set and useful advices.

## References

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

[CKL*07] CUNTZ N., KOLB A., LEIDL M., REZK-SALAMA C., BÖTTINGER M.: GPU-based dynamic flow visualization for climate research applications. In *Proc. SimVis* (2007), pp. 371–384.

[CL93] CABRAL B., LEEDOM L. C.: Imaging vector fields using line integral convolution. In *ACM Proc. SIGGRAPH* (1993), pp. 263–270.

[CP92] CARTWRIGHT J. H. E., PIRO O.: The dynamics of runge-kutta methods. *Int. J. of Bifurcation and Chaos 2*, 3 (1992), 427–449.

[Har05] HARRIS M.: Mapping computational concepts to GPUs. In *GPU Gems 2*. Addison Wesley, 2005, ch. 31, pp. 493–508.

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

[KLRS04] KOLB A., LATTA L., REZK-SALAMA C.: Hardware-based simulation and collision detection for large particle systems. In *Proc. Graphics Hardware* (2004), ACM/Eurographics, pp. 123–131.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: A GPU-based particle engine. In *Proc. Graphics Hardware* (2004), ACM/Eurographics, pp. 115–122.

[Lar04] LARAMEE R. S.: *Interactive 3D Flow Visualization Based on Textures and Geometric Primitives*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2004.

[LHD*04] LARAMEE R. S., HAUSER H., DOLEISCH H., VROLIJK B., POST F. H., WEISKOPF D.: The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum 23* (2004), 203–222.

[LMG06] LIU Z., MOORHEAD R. J., GRONER J.: An advanced evenly-spaced streamline placement algorithm. *IEEE Trans. on Visualization and Computer Graphics 12*, 5 (2006), 965–972.

[OLG*05] OWENS J., LUEBKE D., GOVINDARAJU N., HARRIS M., KRUEGER J., LEFOHN A., PURCELL T.: A survey of general-purpose computation on graphics hardware. In *Proc. Eurographics (State of the Art Report)* (2005), pp. 21–51.

[PBL*04] PARK S. W., BUDGE B., LINSEN L., HAMANN B., JOY K. I.: Multi-dimensional transfer functions for interactive 3D flow visualization. In *Proc. Pacific Graphics* (2004), pp. 1–8.

[PBL*05] PARK S. W., BUDGE B., LINSEN L., HAMANN B., JOY K. I.: Dense geometric flow visualization. In *Proc. EG/IEEE VGTC Symp. on Visualization* (2005), pp. 177–185.

[PVH*03] POST F. H., VROLIJK B., HAUSER H., LARAMEE R. S., DOLEISCH H.: The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum 22*, 4 (2003), 775–792.

[SGS05] STOLL C., GUMHOLD S., SEIDEL H.-P.: Visualization with stylized line primitives. In *Proc. IEEE Conf. on Visualization* (2005), pp. 659–702.

[SMA00] SANNA A., MONTRUCCHIO B., ARINAZ R.: Visualizing unsteady flows by adaptive streaklines. In *Proc. WSCG* (2000).

[Söd02] SÖDERLIND G.: Automatic control and adaptive time-stepping. *Numerical Algorithms 31*, 1–4 (2002), 281–310.

[Stö95] STÖCKER H.: *Taschenbuch mathematischer Formeln und moderner Verfahren*. Verlag Harri Deutsch, 1995, ch. 18.12 Numerische Integration von Differentialgleichungen, pp. 635–ff.

[TGE97] TEITZEL C., GROSSO R., ERTL T.: Efficient and reliable integration methods for particle tracing in unsteady flows on discrete meshes. In *8th Eurographics Workshop on Visualization in Scientific Computing* (1997), Lefer W., Grave M., (Eds.), pp. 49–56.

[The98] THEISEL H.: Visualizing the curvature of unsteady 2d flow fields. In *Proc. 9. Eurographics Workshop on Visualization in Scientific Computing* (1998), pp. 47–56.

[WTS*07] WIEBEL A., TRICOCHE X., SCHNEIDER D., JAENICKE H., SCHEUERMANN G.: Generalized streak lines: Analysis and visualization of boundary induced vortices. *IEEE Trans. on Visualization and Computer Graphics13*, 6 (2007), 1735–1742.

[ZSH96] ZÖCKLER M., STALLING D., HEGE H.-C.: Interactive visualization of 3d-vector fields using illuminated stream lines. In *Proc. IEEE Conf. on Visualization* (1996), pp. 107–ff.