Journal of Visualization, Vol. 8, No. 2 (2005)153-160

On a Graphics Hardware-Based Vortex Detection and Visualization System

Stegmaier, S.* and Ertl, T.*

* Institute for Visualization and Interactive Systems University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany. E-mail: stegmaier@vis.uni-stuttgart.de, ertl@vis.uni-stuttgart.de

Received 17 September 2004
Revised 18 November 2004

Abstract : We present a graphics hardware-based system for interactive denoising, vortex detection, and visualization of vector data. No intermediate results need to be read back by the application once the vector field has been loaded onto the graphics adapter. Due to modern GPUs' parallel processing capabilities, our system significantly outperforms software implementations and thus provides a valuable tool for the interactive exploration of vector fields and the understanding of flow structures.

Keywords : Flow visualization, Vortex detection, Graphics hardware.

1. Introduction

Numerical visualization is an important task when analyzing flow data obtained by physical or numerical flow simulations. Although there is nothing like the “best” visualization technique, vortex detection methods are undoubtedly among the most effective tools for understanding a flow field. However, sophisticated vortex detection is computationally expensive and cannot currently be done interactively on common PCs. However, this may be desirable if the input data is noisy and suitable bandpass filters for eliminating unwanted frequencies are required. In this case, an interactive cycle of filtering, vortex detection, visualization, and evaluation (based on the existing knowledge of the flow) must be entered and repeated until the optimal filter characteristics have been found and a visualization of acceptable quality is obtained. Since neither filtering nor visualization comes for free, this cycle is even more expensive than the vortex detection alone.

In this paper we show that by shifting the entire cycle from the CPU to the graphics processing unit (GPU) and thus by exploiting the modern GPUs' parallel processing capabilities interactive work *is* possible. Our solution expects the vector field data to be made available in a texture—on-board memory which is usually used to store images to be mapped to polygons to make them look more natural—and immediately generates isosurface visualizations of the detected vortices. Upon adjustments of filter support or isovalue, the visualization is instantaneously updated. At no instant is it necessary to pass any intermediate results back to the application. Since our texture-based approach assumes the input data to be defined on a uniform Cartesian grid, processing unstructured grids requires a resampling step which, however, can be done efficiently without a significant loss in accuracy (see Stegmaier et al., 2003) and must be done only once.

2. Related Work

There are several methods for detecting vortices, many of them tailored to specific applications. The papers by (Jiang et al., 2003) and (Post et al., 2002) give an overview and taxonomies of the most popular techniques. So-called *local* methods require only operations within the neighborhood of a cell. All algorithms based on the Jacobian matrix fall into this class. On the contrary, *global* methods examine many grid cells to detect a vortex. Typical representatives of this class are algorithms based on streamline tracing. Since global methods are obviously harder to parallelize and in general more complex than local methods, only local algorithms are suitable for a target platform as limited as a GPU.

Another desirable property of a vortex detection algorithm is Galilean invariance which allows the detection of vortices in time-varying flow fields, too. This further isolates the number of algorithms appropriate for an implementation on the GPU.

To our knowledge, no vortex detection algorithms have been implemented on a GPU yet. However, other techniques have been developed to visualize flow fields using a GPU. Especially texture-based techniques (Weiskopf et al., 2003) are a valuable foundation for this work since their basic procedures are very similar to that of GPU-based vortex detection.

3. Vortex Detection

The most simple vortex detection method that is both local and Galilean invariant is vorticity $\omega := \nabla \times u$ with the vector field $u(x) = (u_1, u_2, u_3)^T$. Since ω is a vector, vorticity is not directly suitable for visualization and its magnitude is, therefore, usually used instead.

A more sophisticated approach that also complies with the criteria defined for a GPU-based implementation is the λ_2 method (Jeong and Hussain, 1995; Müller et al., 1998). It first decomposes the velocity gradient tensor ∇u (the Jacobian) into a symmetric part S and an anti-symmetric part Ω:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \qquad \Omega_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i}\right). \tag{1}$$

From a physical point of view S is the strain-rate tensor and Ω the spin tensor. The λ_2 criterion defines a vortex as a connected region where the second largest of the three eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3$ of the matrix $S^2 + \Omega^2$ is negative. Vortex structures are then visualized by computing isosurfaces based on λ_2. Since both vorticity and the λ_2 criterion start from the Jacobian of the velocity field, vorticity-based vortex detection basically is a by-product of a λ_2-based vortex detection implementation. It will, therefore, serve as our starting point.

4. Programming the GPU

Until a few years ago, graphics hardware included only a fixed-function pipeline that was programmed by setting states and generating geometry. Starting with NVIDIA's GeForce256 very limited programmability became possible with texture-combining modes. Modern graphics adapters include both programmable vertex and fragment processors (Fig. 1) which enable the programmer to operate on each vertex—coordinate vector—provided by the application and each fragment—pixel-like object with additional information—generated by rasterization. The processors are programmed by loading small programs, so-called *shaders* onto the graphics card which are automatically executed during the image generation process. Vertex or pixel shaders are usually used to create advanced visual effects like per pixel lighting, shadows, or refractions in real-time. However, modern GPUs combine several geometry engines and rendering pipelines with a very high

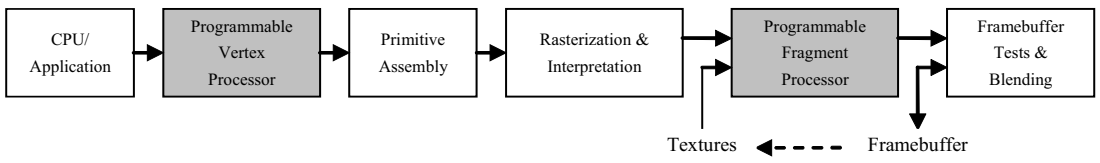

Fig. 1. Simplified programming model for modern GPUs.

memory throughput, so what has entered the desks are actually quite powerful parallel processors. And programmability has made these accessible. This fact has been exploited by many researchers in various fields to create applications that outperform software-based solutions by up to two orders magnitude. Examples can be found both in visualization as well as in general numerical computing (Krüger and Westermann, 2003).

However, GPUs are not as easy to program as CPUs. This is *not* because of the programming environment since the advent of high-level shader languages like NVIDIA's Cg (NVIDIA, 2002) or Microsoft's HLSL (virtually the same languages) has made it possible to write GPU programs with a C-like syntax. Rather, the reason is that the streaming model of GPUs does not support branching and loops with non-constant numbers of iterations. In addition, there are vendor-dependent limits regarding the maximum number of instruction slots[1] and variables that can be used by a program. E.g., on an NVIDIA GeForceFX the maximum number of instructions slots is 1,024 while there are only 64 arithmetic instruction slots and 32 texture lookups available on an ATI 9800.

For our implementation we have chosen the ATI 9800 graphics adapter and DirectX/D3D despite the lower instruction limit since on this platform multi-pass rendering (i.e. the use of previous computation results/pixel values in another rendering pass, see dashed path in Fig. 1) does involve only a very minor performance penalty. And as will be shown, multi-pass rendering is required if the number of redundant computations is to be reduced.

5. System Architecture

The system is divided into an initialization part executed once per dataset and the actual cycle that is entered each time the filter characteristics are adjusted by the user.

Before entering the cycle a one-cell border is added to the volume with the border cell values chosen such that the gradients at the original cells can all be determined using central differences. This relieves us from having to handle border cells differently from inner cells during gradient estimation. Assuming original grid dimensions of $I \times J \times K$ this results in a new volume of the dimensions $(I+2) \times (J+2) \times (K+2)$ subsequently cut into $K+2$ slices of constant Z-coordinate and stored as 2D floating point RGBA-textures.

The cycle in turn consists of three major parts: denoising the raw vector data, computing vorticity magnitude, and rendering the detected vortices.

5.1 Filtering

All measurements are subject to noise. To oppress this noise, a lowpass filter can be employed. Usually, a lowpass filter's ability to oppress noise greatly depends on its support, i.e. the number of neighbors incorporated into the calculation of the filtered value.

In a hardware implementation the neighbors need to be determined by texture lookups; thus, a filter of support N requires N^3 texture lookups to obtain the neighbor information. Since our target

[1] There is a one-to-many relation between *instructions* and *instruction slots* so differentiation between the two terms is inescapable.

platform does only support 32 texture lookups per pass, a single-pass implementation of a non-separable filter is restricted to a filter support of three. This is obviously of limited value for denoising. We therefore concentrate on separable filters requiring $3 \times N$ texture lookups and a multi-pass implementation. Since the Gaussian lowpass filter is separable, isotropic (i.e. it well preserves oriented features) and easy to implement, this filter presents a suitable choice for our application.

The filtering is accomplished by rendering $K + 2$ filled quadrilaterals of $(I + 2) \times (J + 2)$ pixels. In a pixel shader, the current pixel's N neighbor values (including itself) are looked up and multiplied by weights obtained by evaluating the Gauss function at equidistantly spaced points. The data is filtered first in Z-direction to minimize the number of temporary textures.

The passes consume N texture lookups each and 32 and 12 instructions slots for the filtering in X-/Y- and Z-direction, respectively. Since in each pass at most N texture lookups are required, filter supports of up to 31 could theoretically be implemented with the number of texture lookups restricted to 32. However, on our platform the number of texture samplers is currently limited to 16 so the maximum filter support is 15.

5.2 Vortex Dtection

Implementing vorticity-based vortex detection on a GPU is straightforward when using HLSL. All that needs to be done is calculating the Jacobian using central differences from the six surrounding neighbors of the volume element currently being processed:

```
// Determine first column of the Jacobian (x-direction)
forwardNeighbor = tex2D(slice1Sampler, IN.rightTexCoords);
backwardNeighbor = tex2D(slice1Sampler, IN.leftTexCoords);
gradientX = (forwardNeighbor - backwardNeighbor) * DOUBLE_DISTANCES_INV.x;

// Determine second and third column of the Jacobian (y- and Z-direction)
…
vorticity.x = gradientY.z - gradientZ.y;
vorticity.y = gradientZ.x - gradientX.z;
vorticity.z = gradientX.y - gradientY.x;

OUT.color = float4(length(vorticity), 0.0, 0.0, 0.0);
```

The resulting pixel shader code is—with six texture lookups and 16 arithmetic instructions—well within the limits of our hardware platform.

5.3 Volume Visualization

The input vector data is processed slice per slice and the resulting vorticity magnitudes again written to a stack of 2D textures since at the time of this writing rendering to 3D textures was impossible. To generate high-quality visualizations of this stack of 2D textures we adopted a solution proposed in (Rezk-Salama et al., 2000). The basic idea is to determine the intersection polygons of view-aligned slices with the given stack of quadrilaterals and to interpolate color values on these polygons on-the-fly using the two neighboring textures. Unfortunately, this approach suffers from the large number of intersection polygons that have to be calculated each time the volume is rotated. Instead of determining view-aligned intersection polygons on-the-fly we, therefore, pre-compute sets of intersection polygons for the X- and Y-direction (both positive and negative) and switch between them and the original stack depending on the orientation of the volume's bounding box to the viewer. This not only enables us to pre-calculate the intersection polygons but also to send the geometry data of the slices only once to the graphics adapter as a vertex buffer. In addition, assuming

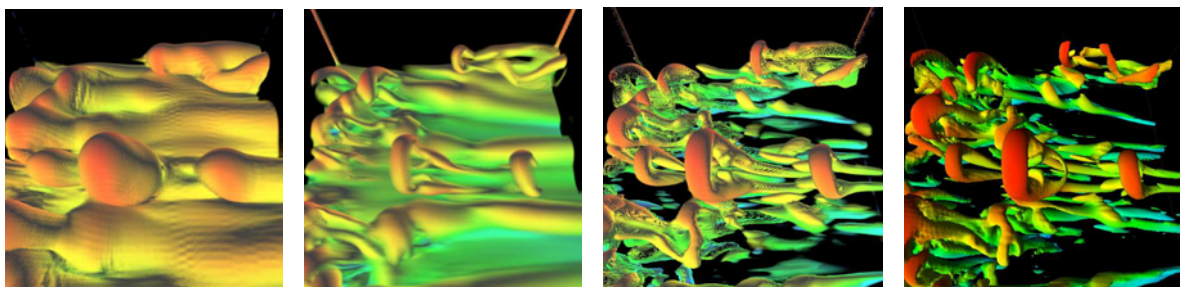

Fig. 2. Isosurfaces of vortex structures in K-type transition simulation data (229 × 116 × 250 voxels). From left to right: a) Vorticity magnitude of original data b) Vorticity magnitude of filtered data c) λ_2 values of filtered data d) Reference image generated by a commercial flow visualization software.

equidistantly-spaced slices of the original stack the amount of geometry stored in the vertex buffers can be further significantly reduced (by factors of $I+1$ and $J+1$, respectively) by storing only a single stack of stripes and rendering the remaining ones with a suitable translation applied. The approach is hence able to accelerate the visualization without consuming a noteworthy amount of memory.

Figure 2 shows two visualizations of vortices detected with our system using a vorticity-based approach. Image a) shows the isosurface obtained from the original dataset, image b) the result after applying a strong Gaussian filter of kernel size 11. The dataset used for these screenshots was obtained by DNS of K-type transition experiments (Kachanov, 1994; Rist and Kachanov, 1995). This kind of data typically exhibits striking Λ- and Ω-vortices. Obviously, these vortex structures are only found in the filtered data while the original data seem to cast a shroud over the vortex structures. However, the individual vortices are not even neatly separated when filtering the data—which comes at no surprise regarding the primitive vortex detection approach. On the other hand, this means that for a productive system a more sophisticated approach like the λ_2-method is required.

6. λ_2 Vortex Detection

A careful implementation of the λ_2 criterion requires almost twice as many instruction slots as are available on the ATI 9800; thus, the implementation must be split into several passes. And since only four 32-bit floating point values can be passed from one pass to the next and expensive multiple render targets (MRTs) are to be avoided, this split-up must be chosen carefully.

It would be natural to set up the matrix $S^2 + \Omega^2$ in a first pass and to compute its eigenvalues in a second pass. However, since the matrix is 3×3 this approach require nine values to be exchanged between the passes. Writing down the expression for $S^2 + \Omega^2$ however reveals that the matrix is symmetric and, therefore, can be exchanged by sending only six values (J denotes the Jacobian):

$$J = \left(a_{ij}\right); \quad J^T = \left(a_{ji}\right) \tag{2}$$

$$S^2 + \Omega^2 = \left(\frac{J + J^T}{2}\right)^2 + \left(\frac{J - J^T}{2}\right)^2 = \left(\frac{1}{2}\sum_{k=1}^{3} a_{ik}a_{kj} + a_{ki}a_{jk}\right) \tag{3}$$

Since this still does not fit into a four-component RGBA value our solution is to calculate the four coefficients of the characteristic cubic polynomial in a first pass and to solve the characteristic equation in a second pass using an adaptation of *Cardan's Solution* (Nickalls, 1993). With this solution only four floating point numbers need to be passed between the passes which tightly fit into an RGBA value and, accordingly, MRTs can be abandoned. Eventually, the first rendering pass in this approach consumes six texture fetches and 59 instruction slots.

The calculation of eigenvalues in the second pass can be implemented very efficiently when taking advantage of the fact that $S^2 + \Omega^2$ is real and symmetric and, therefore, always has three real

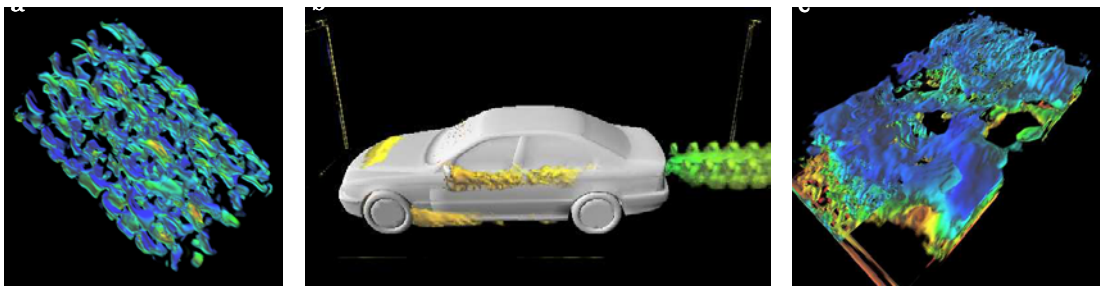

Fig. 3. Vortices extracted on the GPU. From left to right: a) DNS dataset of a flow through a cylinder filled with spheres (201 × 152 × 152 grid); b) flow around car body (RANS simulation, 301 × 131 × 100 grid); c) experimental data of laminar water channel (200 × 40 × 110 grid, split view for two filter kernels). Datasets courtesy LSTM, University of Erlangen, BMW AG, and IAG, University of Stuttgart.

roots and by replacing the required cosine and arccosine calculations by a single texture lookup. This increases the number of texture lookups to two but—with only 55 instruction slots—allows us to stay within the tight hardware limits of our platform.

We will not go into further details on how the two passes are actually mapped to the graphics hardware since the basic approach is similar to what has already been described in Sec. 5 but fairly subtle and tedious details have to be coped with to stay within the instruction limit and to obtain an efficient implementation. Nevertheless, a GPU-based λ_2 implementation *is* possible and a comparison with vorticity-based vortex detection (Figs. 2 b) and c)) well justifies the introduction of a more advanced—while also more time-consuming—detection algorithm.

7. Results

7.1 Visual Evaluation

The presented system combines filtering and vortex detection capabilities. Thus, the system is particularly well suited to noisy data like, e.g., the datasets obtained by experiments. However, the system is also applicable to special types of simulation data.

Both Reynolds Averaged Navier-Stokes (RANS) and Large Eddy simulations (LES) result in an increased computational viscosity which means that small-scale structures will effectively be eliminated through diffusion and dissipation. Neither RANS simulations nor LES will, therefore, be very vulnerable to noise. On the contrary, DNS (Direct Numerical Simulation) should capture all scales that are relevant in the flow without any turbulence modeling. Large Reynolds number flows typically mean small viscosity, and—particularly for turbulent flows—very small length scales in comparison to a relevant geometric length scale. Very small length scales, in turn, means high grid resolution. If insufficient resolution is available, the smallest scales will be determined by the numerics, and not by the physics, which will result in noisy data. We have, therefore, included both experimental and DNS data for the evaluation of our system. Figure 3 shows vortices extracted from DNS data (a), RANS data (b), and experimental data (c). Since these visualizations do not definitely attest the quality of the visualized data, we also created reference images using the commercial flow visualization software *PowerVIZ* (Exa Corp., 2001). Figure 2 c) and d) contrast two λ_2 visualizations of the K-type transition dataset. PowerVIZ implements geometry-based isosurface extraction, does not reveal all its internal settings used for producing the visualizations (perspective transformation, color tables, etc.) and uses higher-precision arithmetic than is currently available on the ATI 9800 (24 bit). Exactly matching the images is, therefore, virtually impossible. However, the visualizations are nevertheless almost identical and give evidence of the high quality that can be obtained with hardware-accelerated implementations.

7.2 Storage Requirements

Our GPU-based implementation stores the input data in textures. With only 256 MB texture memory on our platform this means that memory efficiency is a crucial topic in the evaluation of graphics-hardware–accelerated implementations.

The raw input data is stored in 128 bit RGBA textures which must not be modified since they are needed each time the filter characteristics are adjusted. To store the filtered results, another stack of 128 bit floating point textures is required. These textures are reused for storing the gradients of the λ_2 scalar field required for lighting the isosurface. The remaining textures are independent of the size of the input data and of negligible size. Thus, if the input data comprises N nodes, about 32 Nbyte memory are consumed by the system. Assuming 256 MB texture memory, this means that the system is applicable to datasets of at most eight million grid points or equivalently a cube of the dimensions $200 \times 200 \times 200$. However, since both the filtering and the λ_2 vortex detection are local, a bricking approach can be easily used to accommodate the system to larger datasets.

7.3 Performance Evaluation

The system performance was evaluated with a $135 \times 225 \times 129$ voxel dataset and a viewport size of 512×512 pixels on an ATI 9800 XT graphics adapter (price at time of this writing: about $400). In this configuration, the filtering time was found to be 165 ms for a Gauss filter of support 11 and the λ_2 computation time to be 117 ms. The gradient calculation required for lighting the isosurface took 16 ms. These times are independent of the number of intermediate slices and the direction from which the volume is looked at. On the other hand, the rendering time depends strongly on the direction: For the Z-direction we measured a visualization time of 41 ms (24.4 fps), for the Y-direction a time of 229 ms (4.4 fps). As expected, the frame rates scale linearly with the amount of pixels generated by the application.

A hand-optimized software implementation of the λ_2 vortex detection algorithm using SSE2 vector instructions required 1,150 ms for the vortex extraction on an Intel Pentium 4 processor—almost an order of magnitude slower compared to the hardware-based approach. And PowerVIZ, a tool well-known for its generally high performance, is even slower by a factor of almost 70 (~8 s) which, however, must probably be partially ascribed to its more complex internal data structures (which are hierarchies of Cartesian grids).

8. Conclusions

We have described a system for filtering, vortex detection, and visualization of flow data. By employing modern graphics hardware for performing the calculations instead of the CPU, we were able to improve the system performance by almost an order of magnitude. For the first time, the cycle of filtering, vortex detection, and visualization can be handled interactively using low-cost off-the-shelf hardware readily available at the desks of many researchers.

All the optimizations described in this article were dictated by the particular platform we chose for our implementation. While it may be true that these restrictions will become obsolete with the next generation of graphics cards, it were exactly these limitations which led to highly optimized algorithms and, finally, interactive frame rates. This work can, therefore, be considered to be of great value for interactive vortex detection even for future generations of graphics hardware.

Acknowledgements

We are grateful to the German Research Council (DFG) for financing this work as part of SPP 1147 and to M. Walter and M. Schulz of science+computing AG for making available a prototype of PowerVIZ capable of calculating and visualizing λ_2 values. Furthermore, we would like to thank all the people that provided us with experimental and simulated flow datasets, namely U. Rist and M.

Linnick (IAG, University of Stuttgart), O. Theissen (BMW AG), and T. Zeiser (LSTM, University of Erlangen).

References

Exa Corporation, PowerVIZ specifications (2001), http://www.exa.com/pdf/PowerVIZscreen.pdf.
Jeong, J. and Hussain, F., On the identification of a vortex, J. of Fluid Mech., 285 (1995), 69-94.
Jiang, M., Machiraju, R. and Thompson, D., Detection and visualization of vortices, Visualization Handbook, (2004), Academic Press.
Kachanov, Y. S., Physical mechanisms of laminar-boundary-layer transition. Annu. Rev. Fluid Mech., 26 (1994), 411-482.
Krüger, J. and Westermann, R., Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics (TOG), 22-3 (2003), 908-916.
Max, N. and Becker, B., Flow visualization using moving textures, Proc. of the ICASW/LaRC Symposium on Visualizing Time-Varying Data (1995).
Müller, K., Rist, U. and Wagner, S., Enhanced visualization of late-stage transitional structures using vortex identification and automatic feature extraction, Computational Fluid Dynamics, (1998), 786-791, John Wiley & Sons.
NVIDIA Corp. Cg Language Specification (2002). Available at http://developer.nvidia.com.
Nickalls, R. W. D., A new approach to solving the cubic: Cardan's solution revealed, Mathematical Gazette 77 (1993), 354-359.
Post, F. H., Vrolijk, B., Hauser, H., Laramee, R. S. and Doleisch, H., Eurographics 2002 STAR, State of The Art Report Feature Extraction and Visualization of Flow Fields (2002).
Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G. and Ertl, T., Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization, Eurographics / SIGGRAPH Workshop on Graphics Hardware '00, (2000), 109-118,147, Addison-Wesley.
Rist, U. and Kachanov, Y. S., Numerical and experimental investigation of the K-regime of boundary-layer transition, Laminar-Turbulent Transition, (1995), 405-412.
Stegmaier, S., Schulz, M. and Ertl, T., Resampling of Large Datasets for Industrial Flow Visualization, Workshop on Vision, Modelling, and Visualization VMV '03, (2003), 375-382, infix.
Weiskopf, D., Erlebacher, G. and Ertl, T., A Texture-Based Framework for Spacetime-Coherent Visualization of Time-Dependent Vector Fields, Proc. IEEE Visualization, (2003), 107-114.

Author Profile

Simon Stegmaier: He received his diploma degree in Computer Science in 2001 from the University of Stuttgart and is currently a Ph.D. candidate at the Institute for Visualization and Interactive Sytems at the niversity of Stuttgart. His research interests include remote visualization, flow visualization, and graphics hardware-based methods for accelerating algorithms.

Thomas Ertl: He received the master's degree in computer science from the University of Colorado at Boulder and the Ph.D. degree in theoretical astrophysics from the University of Tübingen. Currently, Dr. Ertl is a full professor of computer science at the University of Stuttgart, Germany, and the head of the Visualization and Interactive Systems Institute (VIS). His research interests include visualization, computer graphics, and human computer interaction in general, with a focus on volume rendering, flow visualization, multiresolution analysis, and parallel and hardware accelerated graphics for large data sets. Dr. Ertl is coauthor of more than 150 scientific publications and he has served on several program and paper committees as well as being a reviewer for numerous conferences and journals in the field. He is a member of the IEEE Computer Society.