A Graphics Hardware-based Vortex Detection and Visualization System

Simon Stegmaier* Thomas Ertl*

Institute of Visualization and Interactive Systems
University of Stuttgart

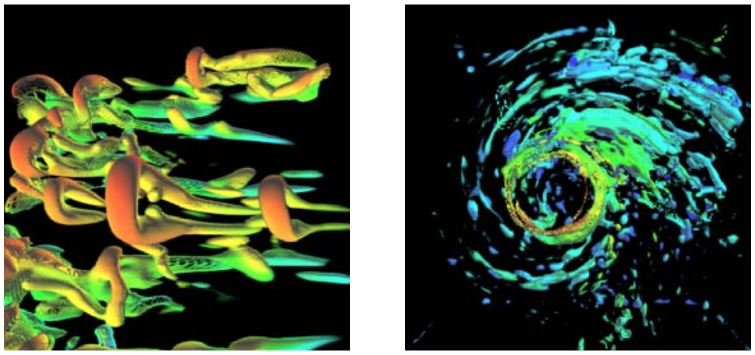


Figure 1: Vortex structures extracted by the GPU. Simulated flow transition (left), simulation of hurricane Isabel (right).

Abstract

Feature detection in flow fields is a well researched area, but practical application is often difficult due to the numerical complexity of the algorithms preventing interactive use and due to noise in experimental or high-resolution simulation data sets. We present an integrated system that provides interactive denoising, vortex detection, and visualization of vector data on Cartesian grids. All three major phases are implemented in such a way that the system runs completely on a modern GPU once the vector field is downloaded into graphics memory. The application aspect of our paper is twofold. First, we show how recently presented, prototypical GPU-based algorithms for filtering, numerical computation, and volume rendering can be combined into one productive system by handling all idiosyncrasies of a chosen graphics card. Second, we demonstrate that the significant speedup achieved compared to an optimized software implementation now allows interactive exploration of characteristic structures in turbulent flow fields.

CR Categories: I.3.3 [Computer Graphics]: Interactive Rendering—Flow Visualization;

Keywords: Features in Volume Data Sets, Flow Visualization, Hardware Acceleration, 3D Vector Field Visualization

1 Introduction

Physical and numerical flow simulations have become an important part of the research activities in both industry and academia. To gain understanding of the simulated flow, it is necessary to perform some kind of data analysis. This is usually done by scientific flow visualization. A good survey of state-of-the-art flow visualization techniques can be found in [17].

While there is probably no single visualization technique that can be regarded the best one, there is no doubt that feature detection methods are among the most effective tools for understanding flow field structures. Of these, in turn, the class of vortex detection algorithms has proven to be of special importance.

However, vortex detection is computationally much more expensive than drawing an isosurface of the velocity magnitude or a slice with any other scalar property mapped to a color. Accordingly, vortex detection currently cannot be done on an off-the-shelf PC at interactive frame rates—which may be desirable for tracking flow structures over several time-steps in unsteady flows or for analyzing flow data obtained by simulation or measuring techniques vulnerable to noise, e.g. direct numerical simulation (DNS) or particle image velocimetry (PIV).

In the latter cases, the vortex detection may be considerably affected by noise (Fig. 2). Accordingly, some form of denoising should be applied to the raw data before one tries to detect vortices. If the frequency of the noise is known in advance, denoising is most effectively accomplished by designing a bandpass filter capable of removing the relevant frequencies.

However, if the noise cannot be exactly located in the frequency domain, 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. Obviously, neither filtering nor visualization come for free so handling the complete cycle is even more difficult than handling the vortex detection alone. For engineers working in the field of fluid dynamics verifying results of experiments or simulations therefore presents a tedious and time-consuming task.

In this paper we present a system to alleviate this situation. We demonstrate that by shifting the entire cycle from the CPU to the GPU and 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 which assumes the in-

*Universität Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany, (stegmaier|ertl)@vis.uni-stuttgart.de

IEEE Visualization 2004
October 10-15, Austin, Texas, USA
0-7803-8788-0/04/$20.00 ©2004 IEEE

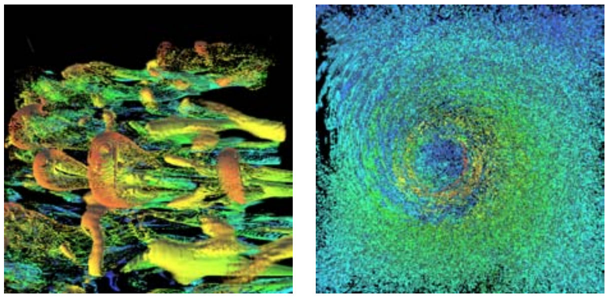


Figure 2: Vortex structures extracted from unfiltered vector fields. The data sets are the same as those used in Fig. 1.

put data is defined on a uniform Cartesian grid. However, there are efficient techniques for the conversion between grids without sacrificing much accuracy (see e.g. [23]), thus, this poses no limitation. Once the vortices have been detected an isosurface of the detected vortex regions is generated. If the user decides to adjust the filter support or the isovalue he will get an instantaneous update of the visualization. Since at no instant any intermediate results have to be passed back to the application, immediate feedback is obtained.

The remainder of the paper is organized as follows: In Section 2 we discuss work of other researchers related to this paper. Section 3 outlines the λ_2 vortex detection method and motivates why it was used in this work. The architecture of our system is presented in Section 4, followed by an evaluation in Section 5. The paper concludes in Section 6.

2 Related Work

A large body of research has been conducted on methods that are able to reliably detect vortices. The papers by Jiang et al. [9] and Post et al. [17] provide good overviews of the most popular techniques and give taxonomies of vortex detection algorithms.

So-called *local* methods require only operations within the neighborhood of a cell; thus, all algorithms based on the Jacobian matrix (or the velocity gradient tensor) 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.

Furthermore, vortex detection algorithms can be classified according to Galilean invariance. A vortex detection algorithm that is Galilean invariant is valid in all inertial frames of reference and thus is able to detect vortices not only in steady but also in time-varying flow fields.

To our knowledge, none of the vortex detection algorithms found in the literature have ever been implemented on a GPU. However, many techniques have been developed to effectively visualize flow fields using a GPU. Especially texture-based techniques—though rather dated [2, 14]—are enjoying great popularity and are still a topic of research [24]. Since the basic procedure of GPU-based vortex detection can be expected to be the same as that of GPU-based flow visualization, these research results are nevertheless a valuable foundation for this work.

GPU-based feature detection is the most prominent part of this work. Two other aspects are GPU-based filtering and volume rendering. Both fields have been studied in detail by several researchers. Hardware-accelerated filters—both linear and non-linear—implementing a wide variety of kernels have been described in [7, 6, 21]. Volume rendering techniques for Cartesian grids are studied in [25, 18, 11]. Since this work concentrates on feature detection and extraction and on integrating various techniques into a productive system, we exclusively relied on these standard techniques for filtering our data and visualizing the vortex structures.

3 The λ_2 Method

Most of the graphics adapters nowadays shipped with off-the-shelf PCs include both programmable vertex and fragment processors. Thus, the programmer is able to operate on each vertex provided by the application and each fragment generated by the rasterization. Since arbitrary data can be stored in texture memory with 32 bit accuracy, these features have been used not only to create real-time advanced visual effects but also to implement numerous numerical algorithms. In fact, the modern GPUs' high parallelism on several levels (memory bandwidth, redundant rendering pipelines) has even led to the development of general numerical libraries that allow the user to transparently use the GPU as a high-performance floating-point coprocessor [12].

Nevertheless, a GPU implementation will not be advantageous for all algorithms. For a GPU implementation to be beneficial, it is of paramount importance that the algorithm exhibits local behavior or—put another way—that it can be efficiently implemented on a multi-processor computer. Thus, referring to the taxonomies given in Sec. 2 global vortex detection methods can immediately be discarded. Galilean invariance was said to be another desirable property of a vortex detection algorithm. This further isolates the number of vortex detection algorithms appropriate for an implementation on the GPU.

Of the algorithms proposed in the literature, we have found the λ_2 method proposed by Jeong and Hussain [8, 15] to be the most suitable algorithm with these properties. This method is generally regarded to produce good results and only shows some shortcomings when applied to turbomachinery flows [20]; thus, the λ_2 method presents an adequate choice.

Given a vector field

$$u(x) = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

the λ_2 method first decomposes the velocity gradient tensor ∇u (the Jacobian matrix of the vector field) into a symmetric part S and an antisymmetric part Ω:

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

From a physical point of view S is the strain-rate tensor and Ω the spin tensor. Next, the eigenvalues of the matrix $S^2 + \Omega^2$ need to be determined. Since this matrix is real and symmetric, there will be three eigenvalues—roots of the characteristic polynomial—denoted by $\lambda_1 \geq \lambda_2 \geq \lambda_3$. A vortex is then defined as a connected region where two of the eigenvalues are negative. The eigenvalue relevant for visualization is λ_2—hence the name of the method.

4 System Architecture

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

Prior to entering the cycle, the vector field (which is assumed to be defined on a Cartesian grid) must be loaded and preprocessed. The preprocessing consists of adding a one-cell border around the volume. The values of the border cells are chosen such that the gradients at the original cells can all be determined using central differences. This work-around relieves us from the burden of having to handle border cells differently from inner cells during gradient estimation. Thus, assuming original grid dimensions of $I \times J \times K$, a volume of the dimensions $(I+2) \times (J+2) \times (K+2)$ is obtained. This volume is then cut into $K+2$ slices each holding all the values of a constant Z-coordinate and each slice copied to a 2D RGBA-texture. We have chosen a 32 bit floating point texture for this purpose since this allows us to store vector components of type `float` without introducing truncation errors on common PC platforms.

The cycle in turn can be further divided into three major parts: denoising of the raw vector data, computation of λ_2 values, and rendering of the output of the vortex detection algorithm. Since the λ_2 values comprise scalar data, any volume rendering technique can be employed for visualization. Thus, part three can be titled the volume visualization phase. The following subsections describe these parts in detail.

For our implementation we have chosen the ATI 9800 graphics adapter and DirectX/D3D despite the low instruction limit of $64 + 32 = 96$ instruction slots. The reason for choosing this platform is that—as will be shown—the system requires multiple passes if the number of redundant computations is to be reduced and multi-pass rendering with the ATI/DX9 combination does involve only a very minor performance penalty.

4.1 Filtering

All measurements—and even some simulation data—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 lookups into textures filled by the application with the appropriate information. Thus, a filter of support N requires N^3 texture lookups to obtain the neighbor information. Our target platform only supports 32 texture lookups per pass; thus, the maximum filter support that can be implemented in a single pass is three. If more advanced filters are required, multi-pass implementations must be resorted to.

For this work, a Gaussian lowpass filter has been employed. This type of filter has two benefits that make it suitable for our application:

- The Gauss filter is isotropic, i.e. it is rotation invariant. In contrary to non-isotropic filters, isotropic filters are good at preserving oriented features which is essential for vortex detection.
- The Gauss filter is separable, i.e. small filter kernels can be implemented in a single pass while large filter kernels can be implemented with several passes with little additional overhead.

For the presented system, both single- and multi-pass Gaussian lowpass filters have been implemented.

Single-Pass Filtering To filter the (border-extended) vector data, the application renders $K+2$ filled quadrilaterals of $(I+2) \times (J+2)$ pixels each. For each quadrilateral, the current slice, the one to the back, and the one to the front are passed as textures. A pixel shader then is executed for each generated fragment. In the shader the neighbors are looked up and multiplied by a weighting factor. The intermediate results are then added and written to the RGB-components of the output pixel. The filtered data is directly rendered to a second stack of $K+2$ floating point textures, each texture again of size $(I+2) \times (J+2)$ texels. Overall, 27 texture lookups plus 44 arithmetic instruction slots are required.

To obtain the weights of the neighbors we evaluate the Gauss function for the arguments 0, x, $\sqrt{2}x$, and $\sqrt{3}x$ and normalize the function values so that the sum over all 27 weights equals one. The value of x is predefined to some constant value and denotes the minimum distance between voxel centers. The remaining arguments are calculated by considering the other distances (match in at most one coordinate) relative to the minimum distance. The arguments for evaluating the Gauss function remain constant. However, the user can adjust the variance σ^2 of the Gauss function and thus can adjust the neighbor weights which are then passed to the pixel shader as program parameters.

Multi-Pass Filtering A single-pass Gauss filter is restricted to support three. If multiple passes are acceptable, the 3D Gauss filter can be separated into three 1D filters, each of them applied in a single rendering pass and reusing the output of the preceding filter pass. The order of the passes is arbitrary and does not effect the result. However, the complexity of the implementation and the amount of temporary texture memory are very well affected.

If the data is filtered first in X- and Y-direction, only a single input and output texture are required for the first two passes since the input data is stored as Z-aligned slices. However, the final Z-direction rendering pass requires N slices to have already been filtered in X- and Y-direction; thus, $N+1$ temporary floating point textures are required.

On the other hand, if the data is filtered first in Z-direction, N slices of the unfiltered vector data are used as input and the result is written to a single floating point texture. For the remaining passes, another floating point texture is required; thus, only two temporary textures are required (independent of the filter support) and no program logic is needed that caches the results of the XY-filtering for a final Z-filtering pass.

As before, the filtering is then done by rendering $K+2$ filled quadrilaterals of $(I+2) \times (J+2)$ pixels. In the pixel shader, the current pixel's N neighbor values (including itself) are looked up. Since all neighbors are equidistantly spaced, the Gauss function is now evaluated at equidistantly spaced points to obtain the weights.

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 either case, 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.

4.2 Vortex Detection

A careful implementation of λ_2 vortex detection requires almost twice as many instruction slots as are available on the ATI 9800. Hence it is impossible to implement the vortex detection in a single pass and the question arises as to where to cut the pixel shader into passes.

This split-up must not be arbitrary. When adjusting an algorithm to multi-pass rendering, intermediate results are written to a texture which in turn is made available to the subsequent pass. This means that the amount of intermediate data per pixel has to fit into a RBGA texel of at most 128 bit. If this does not suffice it is possible on some architectures to use multiple render targets (MRTs). Fortunately, the λ_2 method can very nicely be split into two passes.

The idea is to calculate the coefficients of the characteristic polynomial in the first pass and to solve the characteristic equation in the second pass. Since the characteristic polynomial is a cubic, only four floating point numbers need to be passed between the passes. This data tightly fits into an RGBA value and thus allows us to abandon MRTs.

The following paragraphs outline the shader implementations.

Pass 1: Finding the Characteristic Polynomial The HLSL instruction set includes instructions for manipulating matrices and vectors. Formulating the λ_2 method in HLSL is therefore a trivial task:

```
// Construct the Jacobian and its transposed
jacobianT = float3x3(gradientX, gradientY, gradientZ);
jacobian = transpose(jacobianT);

// Determine the symmetric and antisymmetric parts
symmetric = (jacobian + jacobianT)/2.0;
antisymmetric = (jacobian - jacobianT)/2.0;

// ... and the sum of their squares
sum = mul(symmetric, symmetric) +
      mul(antisymmetric, antisymmetric);

...
```

Unfortunately, this code compiles to too many instruction slots. However, by considering that the matrix to be used for calculating eigenvalues is symmetric, a more efficient (and somewhat more unreadable) formulation can be found that requires only six texture lookups and 59 instruction slots and, therefore, is accepted as a single-pass shader:

```
sum00 =  gradientX.x * gradientX.x +
         gradientY.x * gradientX.y +
         gradientZ.x * gradientX.z;

sum01 =  gradientX.x * (gradientY.x + gradientX.y)/2.0 +
        (gradientY.x * gradientY.y +
         gradientZ.x * gradientY.z +
         gradientX.y * gradientY.y +
         gradientZ.y * gradientX.z)/2.0;
```

The remaining matrix entries can be derived analogously by permutations of indices. Once the sum matrix has been derived, the coefficients of the characteristic polynomial are calculated in a straightforward way and assigned to the current pixel's color to make them available to the solver pass.

Pass 2: Solving the Characteristic Equation To solve the polynomial equation involving the characteristic polynomial, we adopted the modification of *Cardan's solution* proposed by Nickalls [16]. Assuming a cubic polynomial equation

$$ax^3 + bx^2 + cx + d = 0$$

the method gives the three roots (eigenvalues in our case) α, β, and γ by

$$\begin{aligned} \alpha &= x_N + 2\delta\cos(\theta) && (1)\\ \beta &= x_N + 2\delta\cos(2\pi/3 + \theta) && (2)\\ \gamma &= x_N + 2\delta\cos(4\pi/3 + \theta) && (3) \end{aligned}$$

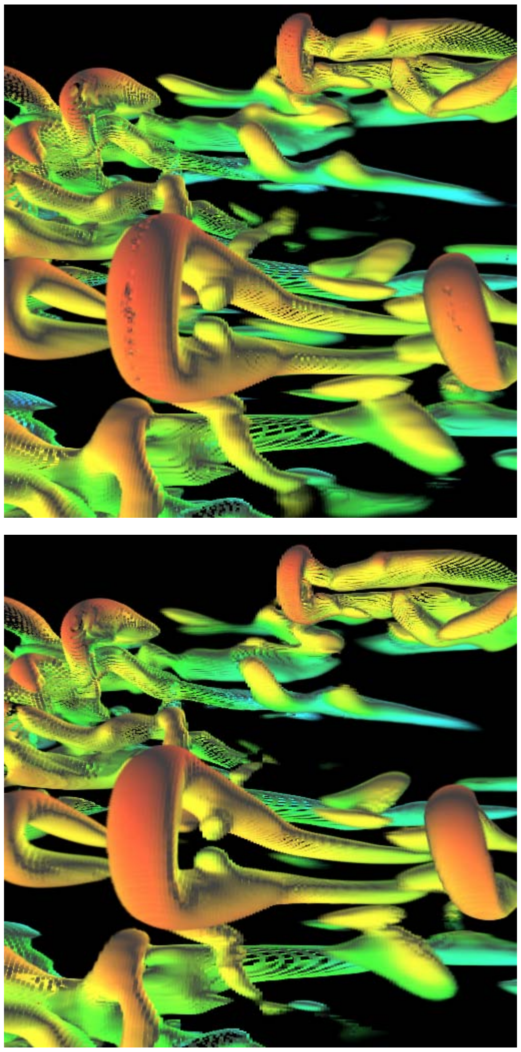


Figure 3: Comparison of the two root finding approaches. As can be seen at the large arch near the image center, the handling of two cases (bottom) removes some artifacts present when handling only one case (top). The colors denote the velocity magnitude.

where

$$\begin{aligned} x_N &= -b/(3a)\\ \cos(3\theta) &= -y_N/(2a\delta^3)\\ \delta^2 &= (b^2 - 3ac)/(9a^2) \end{aligned}$$

and y_N is the value of the polynomial evaluated at x_N.

Expressions (1) – (3) can be easily mapped to graphics hardware since the HLSL instruction set includes a both a `cos` and an `acos` instruction. However, the PS2.0 `sincos` instruction (which the HLSL `cos` is mapped to) consumes eight instruction slots; thus, a total of 24 instruction slots are required for the three cosine calculations alone. As a result, when using the native instructions of the GPU, the maximum number of instructions is surpassed and the program is again rejected.

One often used technique to solve this problem is to replace trigonometric operations with lookups into 1D luminance textures of precalculated function values. Thus, by creating two textures with precomputed cosine and inverse cosine values, respectively, only four

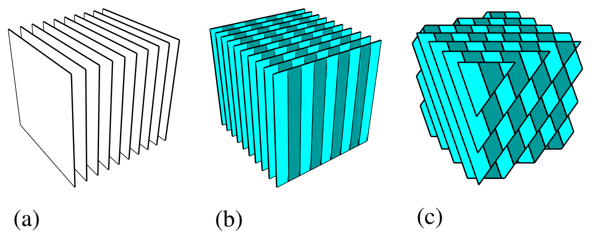


Figure 4: Comparison of slab filling methods. **(a)** Original slices, **(b)** pre-calculated axis-aligned slices, **(c)** viewport-aligned slices calculated on-the-fly.

texture lookups are necessary for obtaining θ and the three cosines.

However, this approach is too general for our application. What is actually needed are cosine values for the arguments $\arccos(-y_N/(2a\delta^3))/3+2k/3\pi$ where $k \in \{0,1,2\}$. It is, therefore, much more efficient to create a single 1D RGBA texture[1] storing precomputed cosines for these arguments and to access the texture with the inverse cosine argument. The three cosines can then be obtained simultaneously with a single texture lookup and no inverse cosine calculations are necessary at all. The resulting program thus requires only two texture lookups and 31 arithmetic instruction slots.

Obviously, compared to the four texture lookups of the more general approach, this is another significant performance gain. This also becomes evident when comparing the λ_2 calculation times: The overall computation time reduces by another 15%.

Actually, the described method for solving the characteristic polynomial does only produce correct eigenvalues if there are three distinct real roots. This is the case in about 99.5 percent for the data set shown in Fig. 5, thus, the results obtained by the given approach are a good approximation. Setting $h = 2a\delta^3$ three real roots are obtained if $y_N^2 < h^2$. For $y_N > h^2$ (one real root) or $y_N^2 = h^2$ (three real roots, two or three equal roots) the calculation must be modified. For the latter case, δ is adjusted to $\delta = \sqrt[3]{y_N/(2a)}$ and the roots are then $\alpha = \beta = x_N + \delta$ and $\gamma = x_N - 2\delta$.

Handling all three cases is costly on a streaming architecture and even exceeds the maximum number of instruction slots of our platform. The matrix $S^2 + \Omega^2$, however, is real and symmetric and, therefore, will always have three real roots. The case $y_N > h^2$ will thus occur only very rarely (if at all) due to numerical inaccuracies. Accordingly, we have decided to neglect this case and to handle only the remaining two cases. The number of required instruction slots in our fragment program then raises from 31 to 55 which is well within the limits of available instruction slots. Fig. 3 contrasts the approach handling two cases with the simplified root finding approach handling only a single case.

4.3 Volume Visualization

The input vector data is processed slice per slice. Rendering the resulting λ_2 values directly into a 3D texture was not possible at the time of this writing, thus, the results are again written to a stack of 2D. Obviously, simply blending quadrilaterals with the λ_2 values mapped as textures will not produce appealing visualizations since virtually nothing at all is seen when looking along the X- or the Y-axis[2].

[1]The ATI 9800 does not support RGB floating point textures so RGBA textures had to be used, leaving the alpha component unused.

Many techniques have been developed to cope with this problem. Rendering view-aligned slices is generally regarded as the method to produce the least artifacts. However, for doing this efficiently, 3D textures are usually required [3] which would require transferring the scalar field from the graphics card memory back to the application. This introduces significant latencies into the vortex detection system and, thus, is not appropriate for our application.

One work-around to this problem was proposed by Rezk-Salama et. al [18] based on the work of Eckel [4]. The basic idea is to determine the intersection polygons of viewport-aligned slices with the given stack of quadrilaterals and to interpolate color values on these polygons using the two neighboring textures. The original stack of quadrilaterals and the resulting intermediate slices are depicted in Fig. 4 (a) and (c), respectively.

A drawback of this approach is the large number of intersection polygons that have to be calculated each time the volume is rotated. Our implementation, therefore, employs a slightly different approach.

Instead of determining viewport-aligned intersection polygons on-the-fly, we pre-compute sets of intersection polygons from the X- and Y-direction (both positive and negative), respectively, and switch between them (and the original stack) depending on the orientation of the volume's bounding box to the viewer. Fig. 4 (b) shows the resulting slices.

This approach 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. If the original stack's slices are assumed to be equidistant, the amount of geometry stored in the vertex buffers can further be 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. Thus, the approach is able to accelerate the visualization without taking a noteworthy amount of memory.

An isosurface is most efficiently extracted from the stack of slices by rendering the slices with all pixels within a user-defined interval around the isovalue set to opaque and the remaining pixels set to transparent. However, the resulting "surface" will have a uniform color which conceals its 3D structure. We have, therefore, integrated a lighting model into our system incorporating both ambient

[2]This statement assumes orthogonal projection; with perspective projection the situation is better but still unacceptable.

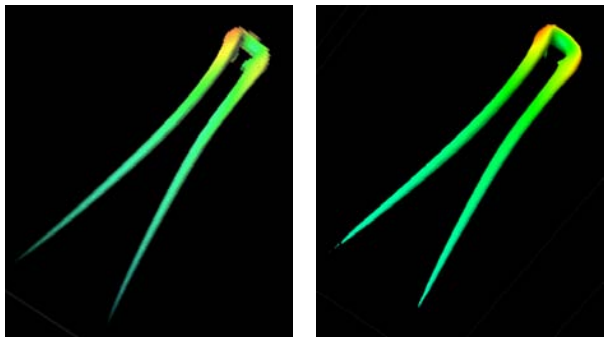


Figure 5: Two visualizations of the same vortex tubes. The left image was produced with the hardware-based approach using direct volume rendering, the right image using a commercial flow visualization tool using a *Marching Cubes* approach.

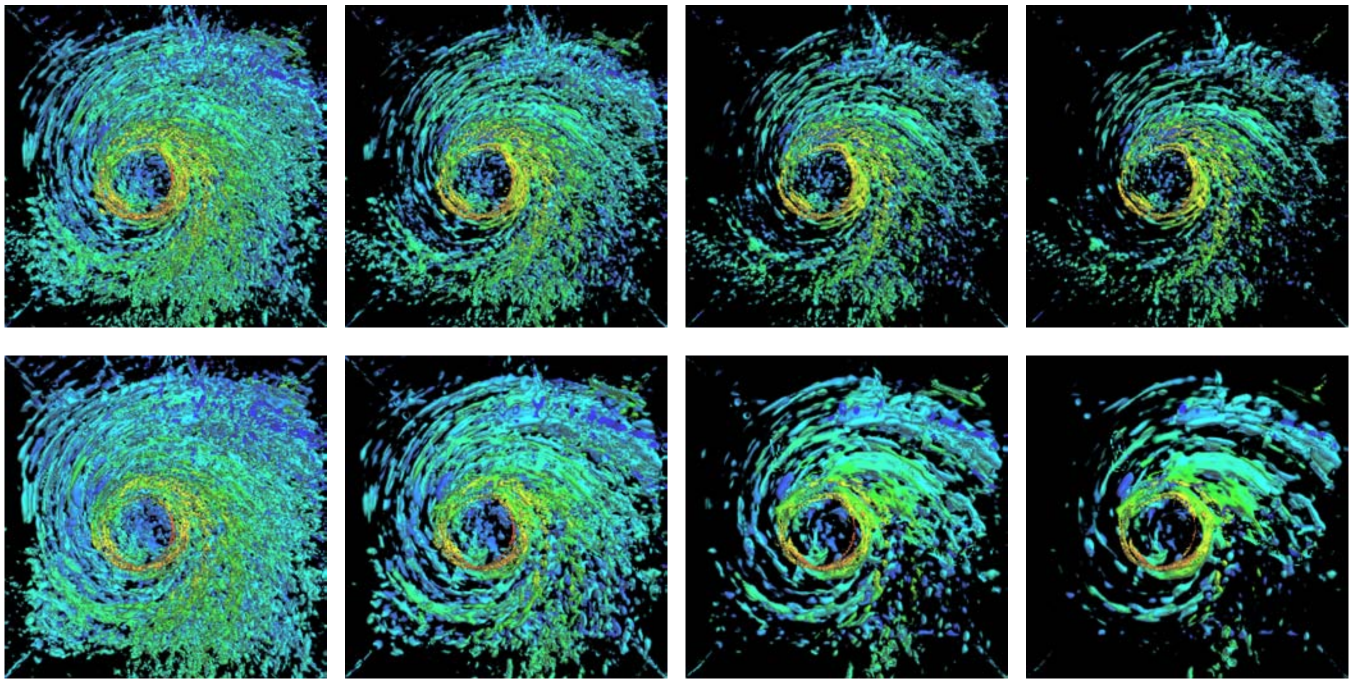


Figure 6: λ_2 isosurfaces of the hurricane Isabel data set. The images in the upper row show the effect of successively decreasing the isovalue. As can be seen, the vortex structures gets narrower. The lower row images illustrate the effect of modifying the filter characteristics starting with the same base image. While both adjustments tend to remove small-scale vortex structures, it can be seen that filtering better preserves large vortex structures.

and diffuse. A full-featured Blinn-Phong model including specular lighting was rejected due to the computational costs (we found that the specular component is about as costly as the ambient and diffuse terms taken together) and the little additional insight that is obtained from highlights. This way, the rendering in Z-direction can be accomplished with one texture lookup and 15 arithmetic instruction slots, the rendering in X- and Y-direction with two texture lookups and 18 instruction slots.

The quality of the extracted isosurface strongly depends on the number of slices used for the visualization. This is inherent in all slice-based volume rendering techniques and no drawback of the approach chosen for this system. We have added the possibility to add intermediate slices interpolated between grid points to be able to trade more pleasing visualizations for reduced frame rates. Fig. 7 shows the effect of adding intermediate slices for the data set depicted in Fig. 3. Although not all artifacts are removed, there is nevertheless a notable improvement in image quality.

Fig. 5 shows screenshots taken with our system (without adding intermediate slices) and screenshots taken with *PowerVIZ*, a commercial flow visualization tool [5]. The data set used for these screenshots has $135 \times 225 \times 129$ voxels and was obtained by DNS of K-type transition experiments [10, 19]. PowerVIZ does not use volume rendering techniques for extracting the isosurface but instead extracts a polygonal representation using a *Marching Cubes* approach. As can be seen, the results are nevertheless very similar in general and only differ (visually) where the vortex tubes narrow and the number of slices over the profile is reduced.

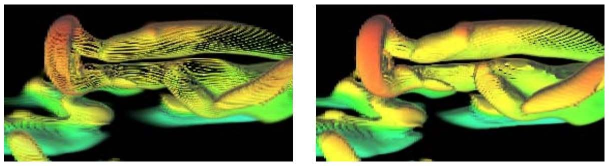


Figure 7: Upper right section of the data set shown in Fig. 3 rendered with no intermediate slices (left) and five intermediate slices (right).

The effects of modifying the filter characteristics and isovalues are depicted in Fig. 6 at the example of a $251 \times 251 \times 100$ data set of hurricane Isabel. As can be seen, the Gauss filtering (lower row) clearly highlights large-scale vortex structures while suppressing small vortices—a behavior that cannot be obtained by isovalue adjustments (upper row).

Fig. 1, left, demonstrates the filtering at the example of a larger K-type transition data set of $229 \times 116 \times 250$ voxels extracted from a more turbulent zone. As can be seen, the characteristic Λ-vortices that are of particular interest to fluid dynamics researchers are much easier to spot in the filtered vector field than in the unfiltered field (Fig. 2, left).

5 Results

5.1 Storage Requirements

Most GPU-based algorithms using fragments programs or pixel shaders for performing the calculations store the input data as textures. However, the amount of texture memory is quite limited and 256 MB must be considered very large at the moment. Memory efficiency is, therefore, a crucial topic in the evaluation of graphics-hardware–accelerated implementations since it eventually defines an upper limit for the size of the input data sets.

Our system stores the raw input data in a 128 bit RGBA texture. This texture must not be modified since it is needed each time the filter characteristics are adjusted. To store the filtered results, another 128 bit floating point texture is required. This texture is 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 $32N$ byte memory are consumed by the system. Assuming 256 MB texture memory, this means that the system is applicable to data sets of at most eight million grid points or equivalently a cube of the dimensions $200 \times 200 \times 200$. We consider this to be sufficient for data sets obtained experimentally.

However, both the filtering and the λ_2 vortex detection are local. Thus, if the system limits are surpassed, a bricking approach can be easily used to accommodate the system to larger data sets.

5.2 Performance Evaluation

For evaluating the performance of the system the data set of Fig. 5 of the dimensions $135 \times 225 \times 129$ was visualized at a viewport size of 512×512 pixels on an ATI 9800 XT graphics adapter. Since the system is almost completely GPU-based, the configuration of the underlying PC/workstation is irrelevant for the benchmark.

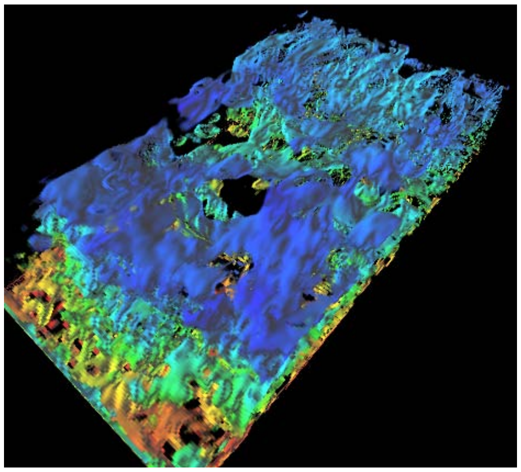
We have found the filtering time to be about 147 ms for both the single-pass Gauss filter of support three and the multi-pass Gauss filter of support 11. Obviously, the reduced number of texture lookups required for the separable Gauss filter more than outweighs the overhead of two additional rendering passes. The λ_2 computation took 108 ms for the simplified root finding approach and 130 ms for the correct approach, the gradient calculation time was 14 ms. These times are independent of the number of intermediate slices and the direction from which the volume is looked at.

One the other hand, the rendering time depends strongly on the direction and the number of intermediate slices, therefore, we have measured rendering times for both the cheapest direction (Z-direction, original stack of slices) and the most expensive direction (Y-direction, stacks of pre-calculated stripes, largest of the three dimensions).

For the Z-direction we measured a visualization time of 38 ms (26.3 fps), for the Y-direction a time of 212 ms (4.7 fps). When adding one intermediate slice, the visualization times increase by a factor of two. As expected, the frame rates scale linearly with the amount of pixels generated by the application. No color mapping of velocity magnitudes was done for these measurements.

To judge the performance of a GPU-based algorithm, a comparison with an equally optimized software implementation is required. For this reason, we have also implemented the vortex detection algorithm proposed above on an Intel Pentium 4 processor. All the proposed optimizations like e.g. the usage of lookup tables for determining cosine values were included. Since the P4's SSE2 extension is capable of processing four-component vectors with 32 bit per components, macros for performing vector operations were implemented and used where possible. Unfortunately, there is no SSE2 instruction a dot product can be mapped to; thus, an instruction sequence had to be used instead[3]. We have chosen the sequence proposed by Breternitz et al. [1]. Using this implementation, we measured 1,150 ms for the λ_2 vortex detection on the $135 \times 225 \times 129$ data set—almost an order of magnitude slower compared to the hardware-based approach.

[3]MMX allows the dot product to be mapped to two instructions using the `PMADDWD` instruction. However, each vector component then is confined to 16 bit which is unacceptable for this application.



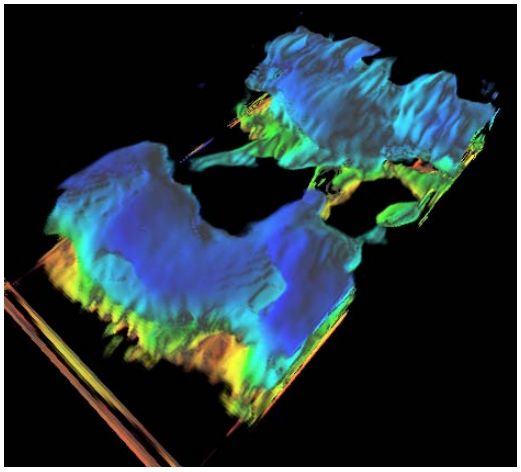


Figure 8: Visualizations of vortex structures in an experimentally obtained water channel data set with little filtering (upper image) and strong filtering (lower image).

As expected, PowerVIZ, a tool well-known for its generally good performance is even significantly slower (about 8 s) than the optimized software implementation. This must probably be ascribed to more complex internal data structures since PowerVIZ works on a hierarchy of Cartesian grids while our system requires a single Cartesian grid. Anyway, using an own optimized implementation for comparison is well justified.

5.3 Application to CFD

An interactive cycle of filtering, vortex detection, and visualization is particularly useful for noisy data. For experimentally obtained datasets this is obviously the case. Fig. 8 shows a data set obtained in an experiment studying laminar-turbulent boundary layer transitions [13].

For data sets obtained by computational fluid dynamics (CFD) the situation depends on the simulation technique. Turbulence models vary in how much modelling they contribute to the fluid flow simulation. At one end of the spectrum there is RANS (Reynolds Averaged Navier-Stokes) which models turbulence by essentially increasing the fluid viscosity. Increased viscosity leads to the elimination of small-scale structures through diffusion and dissipation. Thus, RANS simulations usually are only modestly vulnerable to noise. Turbulence modelling in LES (Large Eddy Simulation) also results in an increased computational viscosity in comparison with the molecular viscosity and accordingly also results in data sets with moderate noise.

On the contrary, DNS should capture all scales that are relevant in the flow without any modelling. Large Reynolds number flows typically mean small viscosity, and, particularly for turbulent flows, very small length scales. Thus, high grid resolutions are required for DNS. If insufficient resolution is available, the smallest scales will be determined by the numerics and not by the physics, which eventually results in noisy data.

Our system is, therefore, particularly suitable for environments where experimental (Fig. 8) or DNS data (Fig. 1 and 3) are subject of analysis.

6 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. The presented system has already been approved by engineers working in the field of fluid dynamics. Despite the fact that basically only several well-known techniques are combined into a single tool our collaborators have decided that the performance and effectiveness of our system well justify an integration into their workflow.

In the future, we plan to evaluate further local vortex detection algorithms and more advanced filters in terms of portability to graphics hardware. Furthermore, we intend to improve the volume rendering to be able to produce more pleasing visualizations for data sets with a low number of slices. Another attractive feature would be the tracking of features over several time-steps as it was presented in [22].

Acknowledgements

We are grateful to the German Research Council (DFG) for financing this work as part of SPP 1147, to Milosz Walter and Martin Schulz of science+computing AG for making available a prototype of PowerVIZ capable of calculating and visualizing λ_2 values, to the organizers of the IEEE Visualization 2004 contest for providing the hurricane dataset, and to Ulrich Rist and Mark Linnick of the Institute of Aerodynamics and Gasdynamics (IAG) of the University of Stuttgart for providing experimental and simulated flow data sets and for sharing their insights regarding the analysis of flow field structures.

References

[1] M. Breternitz, Hum. H., and S. Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, pages 135–145, 2003.

[2] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH '93*, pages 263–270. ACM Press, 1993.

[3] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR93-027, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1993.

[4] G. Eckel. *OpenGL Volumizer Programmer's Guide*. Silicon Graphics, Inc., Mountain View, CA, USA, 1998.

[5] Exa Corporation. PowerVIZ specifications, 2001. http://www.exa.com/pdf/PowerVIZscreen.pdf.

[6] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering on PC hardware. In *Workshop on Vision, Modelling, and Visualization VMV '01*, pages 105–112, 2001.

[7] M. Hopf and T. Ertl. Accelerating 3D Convolution using Graphics Hardware. In *Proceedings of IEEE Visualization '99*, pages 471–474. IEEE, 1999.

[8] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, pages 69–94, 285 1995.

[9] M. Jiang, R. Machiraju, and D. Thompson. Detection and visualization of vortices. In C. Johnson and C. Hansen, editors, *Visualization Handbook*. Academic Press, 2004.

[10] Y. S. Kachanov. Physical mechanisms of laminar-boundary-layer transition. *Annu. Rev. Fluid Mech.*, pages 411–482, 26 1994.

[11] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization '03*, pages 287–292. IEEE, 2003.

[12] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[13] M. Lang, U. Rist, and S. Wagner. Investigations on Disturbance Amplification in a Laminar Separation Bubble by Means of LDA and PIV. In *Proceedings of the 11th International Symposium on Laser Techniques to Fluid Mechanics*, 2002.

[14] N. Max and B. Becker. Flow visualization using moving textures. In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data*, 1995.

[15] K. Müller, U. Rist, and S. Wagner. Enhanced visualization of late-stage transitional structures using vortex identification and automatic feature extraction. In *Computational Fluid Dynamics*, pages 786–791. John Wiley & Sons, 1998.

[16] R. W. D. Nickalls. A new approach to solving the cubic: Cardan's solution revealed. In *Mathematical Gazette*, volume 77, pages 354–359. The Mathematical Association, 1993.

[17] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. Eurographics 2002 STAR – State of The Art Report Feature Extraction and Visualisation of Flow Fields, 2002.

[18] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Graphics Hardware Workshop*, pages 109–118,147. Addison-Wesley Publishing Company, Inc., 2000.

[19] U. Rist and Y. S. Kachanov. Numerical and experimental investigation of the K-regime of boundary-layer transition. *Laminar-Turbulent Transition*, pages 405–412, 1995.

[20] M. Roth and R. Peikert. A higher-order method for finding vortex core lines. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *Proceedings of IEEE Visualization '98*, pages 143–150. IEEE, October 1998.

[21] M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84. Springer, 2001.

[22] D. Silver and X. Wang. Tracking Scalar Features in Unstructured Datasets. In *Proceedings of IEEE Visualization '98*, pages 79–86. ACM Press, 1998.

[23] S. Stegmaier, M. Schulz, and T. Ertl. Resampling of Large Datasets for Industrial Flow Visualization. In *Workshop on Vision, Modelling, and Visualization VMV '03*, pages 375–382. infix, 2003.

[24] D. Weiskopf, G. Erlebacher, and T. Ertl. A Texture-Based Framework for Spacetime-Coherent Visualization of Time-Dependent Vector Fields. In *Proceedings of IEEE Visualization '03*, pages 107–114. IEEE, 2003.

[25] R. Westermann and T. Ertl. Efficiently using Graphics Hardware in Volume Rendering Applications. In *Proceedings of SIGGRAPH '98*, pages 169–177, 1998.