A Particle System for Interactive Visualization of 3D Flows

Jens Krüger, Peter Kipfer, Polina Kondratieva, Rüdiger Westermann

***Abstract*— We present a particle system for interactive visualization of steady 3D flow fields on uniform grids. For the amount of particles we target, particle integration needs to be accelerated and the transfer of these sets for rendering must be avoided. To fulfill these requirements, we exploit features of recent graphics accelerators to advect particles in the graphics processing unit (GPU), saving particle positions in graphics memory, and then sending these positions through the GPU again to obtain images in the frame buffer. This approach allows for interactive streaming and rendering of millions of particles, and it enables virtual exploration of high resolution fields in a way similar to real-world experiments. The ability to display the dynamics of large particle sets using visualization options like shaded points or oriented texture splats provides an effective means for visual flow analysis that is far beyond existing solutions. For each particle, flow quantities like vorticity magnitude and λ_2 are computed and displayed. Built upon a previously published GPU implementation of a sorting network, visibility sorting of transparent particles is implemented. To provide additional visual cues, the GPU constructs and displays visualization geometry like particle lines and stream ribbons.**

***Index Terms*— Flow visualization, particle tracing, programmable graphics hardware, visibility sorting, visualization geometry.**

I. Introduction

ADVANCES in experimental flow analysis and flow numerics are making available an unprecedented amount of data from physical phenomena. Relevant data can be captured and simulated with sufficient accuracy to permit reliable extraction of required information and to even disclose the world of unsteady flow mechanics. In flow research and industrial practice vector field data is one of the key sources for the analysis of flow field dynamics. Visual exploration of such fields imposes significant requirements on the visualization system and demands for approaches capable of dealing with large amounts of vector valued information at interactive rates. With increasing data and display resolution the number of graphical primitives required to comprehensively visualize such fields grows significantly.

In real-world fluid flow experiments [1], [2], external materials such as dye, water vapor, liquid or gas droplets are seeded into the flow. The advection of these materials creates flow lines or particle traces that show the flow patterns. Such tracer experiments have been simulated by scientific visualization researchers. Numerical methods and three-dimensional computer graphics techniques have been used to advect particles and to produce graphical primitives such as arrows, motion particles, particle lines, stream ribbons, and stream tubes. These primitives can emphasize flow properties and they act as depth cues to assist in the exploration of complex spatial fields.

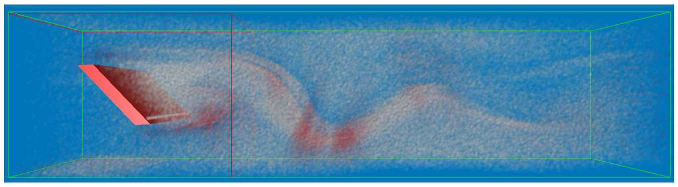

Fig. 1. Using a RK3(2) integration scheme, one million particles can be traced through the flow at 60 fps. The local integration error is color coded.

These techniques are effective in showing the flow fields local features, but they cannot produce *and* display the amount of primitives needed to visually convey large amounts of 3D directional information at interactive rates. Both numerical and memory bandwidth requirements imposed by particle integration schemes are too high as to allow for simultaneous advection of large particle sets. Even if particle positions can be updated at sufficient rates, particle rendering includes the transfer of data to the graphics system and therefore limits the performance.

In the following key experiment, the requirements on particle-based visualization techniques for flow fields are emphasized. Particle advection is used to visualize one time step of a numerical simulation of incompressible flow. The advection step is carried out on the CPU, and updated particle positions are sent to the GPU for rendering. The simulation is run on a uniform grid of size $128^2 \times 512$. Results are displayed on a screen resolution of 1600×1200.

One million particles are traced through the flow using an embedded Runge-Kutta integration scheme of 3rd order. Embedded schemes can be used to control numerical accuracy by computing and adaptively selecting the appropriate integration step size. In the experiment, however, the scheme is not used for this purpose. Instead, all particles move with the same step size and the error estimate that is computed by the integrator is visualized. The user can change this step size until for every particle the integration error is below a selected threshold. This allows one to trade between simulation speed and accuracy while the movement of particles is still bound to the simulation time step.

The experiment was run on a Pentium 4 3.0 GHz processor with 512 kB second level cache and 1 GB dual channel DDR2 main memory. The PC is equipped with an ATI X800 XT

All authors are with the Computer Graphics and Visualization Group at the Department of Computer Science, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany.
Email: {jens.krueger|kipfer|kondrati|westermann}@in.tum.de

graphics card. Particles are rendered as transparent points. The local integration error is color coded, ranging from low (white) to high (red) error (see Figure 1). The vector field data is stored as 16 Bit floating point data, and it is organized into blocks that fit into one cache line. Thus, for the advection of one particle cache misses are reduced. Because particle positions change in every frame, these positions can not be stored in server-side (i.e. GPU) video memory. In every frame the entire set of particle coordinates and associated color attributes has to be transferred to the graphics processing unit.

On the target architecture the maximum number of attributed points in client-side (CPU) memory that can be rendered per second is 15 millions. Consequently, in our experiment particles can be rendered at a frame rate of at most 15 fps independent of the performance of the integrator. The integrator itself updates 0.51 million positions per second, and we are seeing compute power becoming the major performance bottleneck. Overall, a frame rate of roughly 0.5 fps can be achieved. With the ability to do more integration steps per time interval, i.e. by using parallel architectures, SIMD optimization, lower order integration schemes or faster memory, the bandwidth required will grow substantially.

In this paper, we propose a method to overcome both computation and bandwidth limitations in particle tracing. Using this method, the experiment runs at 47 fps. This is achieved by leveraging functionality on recent graphics accelerators to carry out particle advection and rendering. Updated particle positions are saved in graphics memory, and they are then processed on the GPU again to obtain images in the frame buffer. Parallelism and memory bandwidth in the fragment units is exploited to accelerate numerical integration and to generate additional visualization geometry as shown in Figure 2. During particle tracing, bus transfer between the CPU and the GPU can be almost entirely avoided. Only a few API calls to execute the required GPU operations have to be issued.

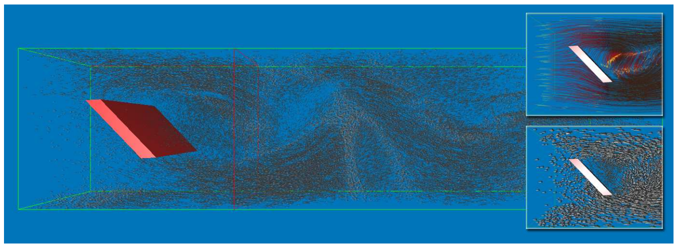
Fig. 2. Particles can be rendered using different visualization options like oriented splats or stream lines.

In contrast to topology or feature based techniques [3]–[5], which reduce the information to be displayed by extracting relevant flow structures, our method provides an interactive means for visualizing 3D flow dynamics. It allows the user to guide the visualization process at arbitrary resolution, and it can be used to virtually explore high resolution fields in a way similar to real-world experiments. In particular, since the proposed particle approach does not require a preprocessing step and enables direct visualization of flow dynamics as well as derived flow quantities, the system is well suited to serve as a back end for sources of unsteady flow.

The remainder of this paper is organized as follows. In Chapter 2 we summarize previous work, which is related to ours. We then describe particle integration schemes and we discuss their realization on recent graphics cards. The use of parallel fragment units for visibility sorting is subject of Chapter 4. In Chapter 5 we introduce various visualization options, and we show timing statistics for different scenarios. Chapter 6 is dedicated to the reconstruction of visualization geometry like stream lines and stream bands. We conclude the paper with a discussion of techniques to visualize unsteady flow and vector fields given on non-uniform grids.

A. *Related Work*

Over the last decade, particle tracing techniques for flow visualization have been studied intensively. At the core of these techniques, numerical integration schemes are employed to compute accurate particle trajectories in steady or unsteady vector fields. In the context of flow visualization, the analysis of such schemes with respect to accuracy, stability and performance has been done [6]–[9].

To integrate the velocity along particle traces in discrete meshes, techniques for particle location and vector field interpolation are required. In non-uniform grids, point location takes a significant fraction of one integration step. To overcome this burden, the physical domain is often transformed to a uniform computational domain, where point location is less expensive [6], [8], [10]. Computational space particle tracing, on the other hand, was shown to produce less accurate results because in general local transformations can only be approximated. To accelerate point location in physical space, cache coherence and efficient data structures have been utilized [11]. On-the-fly tetrahedral decomposition of curvilinear cells has been described by Kenwright and Lane [12] to efficiently predict the next cell that is entered by a particle. Then, barycentric interpolation yields the velocity field inside the elements. Since the trajectory of particles through tetrahedral cells can be computed exactly, for this kind of primitives approximate numerical integration can be entirely avoided [13], [14].

Regardless the underlying grid structure, adaptive step size control in numerical integration can significantly reduce the number of numerical and memory access operations in particle tracing. The local truncation error is most commonly used to steer the refinement of particle traces [15], but also curvature based criteria [7], [12] and refinement schemes based on velocity magnitude [6], [16] have been considered.

Acceleration techniques for particle tracing also include implementations on parallel architectures [17], [18] as well as out-of-core strategies based on application controlled demand paging. In the work by Bruckschen et al. [19], at run-time pre-computed particle traces are loaded from disk, whereas Cox et al. and Ueng et al. [16], [20] discussed data partition and caching strategies for particle tracing.

To render particle trajectories, positional and directional information needs to be mapped to graphical icons. Particles can be rendered directly as point primitives or they can be connected by line segments. In either case, color provides a means for encoding flow properties like divergence, velocity magnitude or vorticity. Shaded primitives, transparency, depth cues and halos greatly enhance the perception of particle traces in 3D [21]–[23]. Oriented texture splats [24] have been utilized for image based rendering of graphical primitives, and they significantly reduce the load in the vertex unit of the graphics system. More complex visualization objects include stream ribbons, stream tubes or time surfaces [11], [16], [25]. Geometric properties of these objects can be modified according to the intrinsic flow properties to reveal local structures in the flow.

While particle based techniques can effectively visualize local features in the flow, global imaging techniques for visualizing 3D fields [26]–[29] can successfully illustrate the global behavior of such fields. However, it is difficult when using such methods to effectively control particle density in a way that depicts both the direction structure of the flow *and* the flow magnitude. Usually these techniques do not allow for flow visualization at interactive rates.

LIC-methods [30], [31], on the other hand, allow for interactive visual analysis of high resolution 2D vector fields. However, such techniques generally fail if utilized to globally visualize 3D flow. This is because of the tremendous information density they produce and their inherent occlusion effects. Only by selecting regions in the renderable representation can structures in 3D be emphasized [32]–[35]. These techniques, however, do not allow for selective and integral visualization of characteristic particle traces in general.

II. Particle Tracing

Particle tracing is a technique for computing the trajectory of massless particles in a flow field over time. In classical particle tracing the ordinary differential equation

$$\frac{\partial \tilde{x}}{\partial t} = \tilde{v}(\tilde{x}(t), t) \tag{1}$$

equipped with appropriate initial condition $\tilde{x}(0) = x_0$ is solved numerically. Here, $\tilde{x}(t)$ is the time-varying particle position, $\frac{\partial \tilde{x}}{\partial t}$ is the tangent to the particle trajectory, and $\tilde{v}$ is an approximation to the real vector field v. The exact particle trace is given by the solution of

$$\frac{\partial x}{\partial t} = v(x(t), t) \tag{2}$$

with the same initial condition. As v is sampled on a discrete lattice, interpolation must be performed to reconstruct particle velocities along their characteristic lines. The higher the approximation order of the integration scheme, the more often the interpolation function has to be evaluated. Consequently, the number of both numerical operations and memory access operations increases in a higher order setting. Additionally, in a single integration step the memory footprint is enlarged, letting higher order schemes become less efficient in terms of memory cache coherence.

Besides fixed step size integration schemes like classical Euler or Runge-Kutta, embedded schemes are known to yield superior results both with respect to accuracy and speed. In embedded schemes, the local integration error is used to refine or to enlarge the integration step size. In particular, RK3(2) computes a third and a second order solution and estimates a fourth order accurate local truncation error from them. Using the truncation error, the optimal step size can be computed. If it is larger than the current step size, the third order solution is taken and the next integration step will use the larger step size. If it is smaller, a third order solution using the reduced step size is recomputed.

For particle visualization, where one is interested in continuous animation of particle sets within constant time intervals, adaptive schemes are only of limited relevance. Consecutive particle positions have to be displayed at equally spaced simulation time steps. Therefore, adaptive schemes require varying numbers of operations to be carried out per particle. In a single animation frame, some particles perform many small integration steps while others have to repeat the integration step with reduced step size. As a consequence, the number of particles that can be processed within one animation frame varies non-deterministically.

In this work, we use an embedded RK3(2) scheme for numerical particle integration, and we use the local integration error as an additional visual cue. The proposed particle system provides a visualization mode that enables the user to visualize the local per particle truncation error using the third and second order solution to the current particle position, respectively. By reducing the global integration time step this error can be reduced accordingly.

III. GPU Particle Tracing

Our GPU particle system for interactive exploration of 3D flow fields exploits functionality on recent graphics cards. On such cards, it is now possible to access texture maps in the vertex units and to allocate memory objects that can be interpreted as texture maps and vertex arrays alternatively. Figure 3 is a pictorial representation of the improvements that have recently been made to the rendering pipeline. In combination with programmable fragment shaders, this functionality enables construction, manipulation and rendering of geometric data on the GPU. By using this functionality, particle tracing can be entirely performed on the GPU without any read back to application memory.

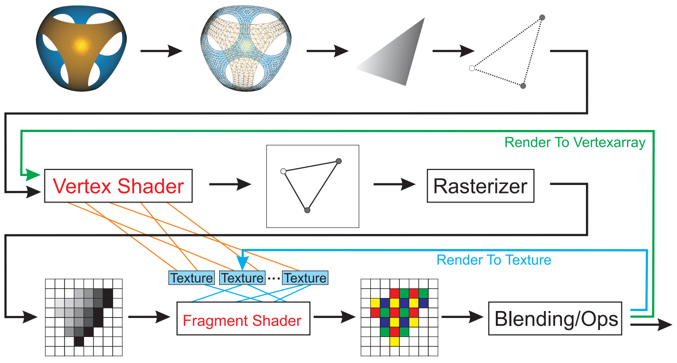

Fig. 3. On recent GPUs, textures can be accessed in the vertex units, and rendering can be directed into textures and vertex arrays.

Our method computes intermediate results on the GPU, saves these results in graphics memory, and uses them again as input to the geometry units to render images in the frame buffer. This process requires application control over the allocation and use of graphics memory; intermediate results are "drawn" into invisible buffers, and these buffers are subsequently used to present vertex data or textures to the GPU.

Initial particle positions are stored in the RGB color components of a floating point texture of size $M \times N$. These positions are distributed regularly or randomly in the unit cube. The visual effect of different distributions is demonstrated in Figure 4. In the alpha component, each particle carries a random floating point value that is uniformly distributed in the range of $(0.75, 1.25)$. This value is multiplied by a user defined global lifetime to give each particle an individual lifetime. By letting particles die – and thus reincarnate – after different numbers of time steps, particle distributions very similar to those generated in real-world experiments can be simulated.

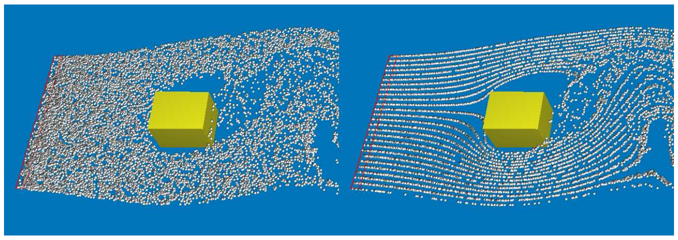

Fig. 4. Random and regular distribution of starting positions within the particle probe.

The user specifies the number of particles continuously released into the flow. Once this number is changed, an appropriately sized initial particle texture is generated on the CPU and it is uploaded to the GPU. Particle integration now consists of two steps: incarnation and advection.

A. Particle Incarnation

The user can interactively position and resize a 3D probe that injects particles into the flow. Both the position and the size of this probe are specified with respect to the local object coordinate system in which the flow field is initially defined. These parameters are encoded in a transformation matrix A.

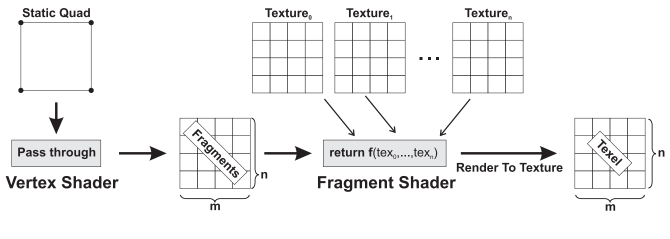

Fig. 5. A fragment stream is generated by rendering a quad that covers as many pixels as there should be items in the stream.

In every simulation pass, a stream of $M \times N$ fragments corresponding to a contiguous block of pixels in screen space is generated. Figure 5 illustrates this procedure. The fragment output is rendered to an equally sized texture render target, which becomes the particle container in the next pass. The initial particle container is bound to the first texture unit thus enabling access to initial particle attributes in a fragment shader. In the first simulation pass, the shader performs the following operations:

- *Transformation*: Unit particle coordinates are read from the texture unit and transformed with respect to A. The matrix is issued by the application program as a uniform shader parameter.
- *Birth*: Each particle is born by initializing its lifetime as described. The maximum lifetime is presented to the shader via a uniform parameter.
- *Update*: Updated particle coordinates and the initial lifetime are output to the RGB and alpha components of the render target, respectively.

The 2D render target now contains for every particle both its current local object space coordinate and lifetime. In the advection step these values are changed subsequently.

B. Particle Advection

Particle advection is performed in the fragment units. The above procedure is employed to generate $M \times N$ fragments, which read current particle positions, advect these positions and store the results in an additional texture. Besides the initial particle container, the current container and a 3D texture storing the velocity field are bound to a second and third texture unit. Vector valued information is stored in the RGB color components of the 3D texture. Since tri-linear interpolation in 32 Bit floating point textures is not supported on current GPUs, vector components are stored as 16 Bit floating point values internally represented in the OpenEXR formt (6 bits exponent and 9 bits mantissa). In the fragment shader, the following operations are carried out:

- *Texture Access*: Current particle positions and lifetimes are read from the second texture unit.
- *Death Test*: The shader checks for positions outside the domain or lifetimes equal to zero. If one of these conditions is true, the particle is reincarnated. Otherwise, it is advected through the flow.
  - *Advection*: Particles are advected using the RK3(2) integration scheme, which involves multiple fetches into the third texture unit. In addition, each particles' lifetime is decremented by one. Updated positions are written to a 2D texture render target, which becomes the particle container in the next pass.
  - *Reincarnation*: The same operations as described above for particle incarnation are carried out.

Our proposed particle engine for flow visualization on the GPU is implemented in Cg [36]. Cg allows one to use high level constructs and to abstract from the underlying hardware architecture. In the following, a Cg code fragment implementing particle advection is shown.

```
float4 advection(app2Vertex IN) : COLOR0 {
  float4 pos   = tex2D(sPositions,IN.Coords);
  float3 field = tex3D(sVolume,pos.xyz).xyz;

  // pos.w stores livetime
  pos += float4(field,-1);

  if (pos.w <= 0 || outOfGrid(pos.xyz))
    pos = tex2D(sStartPositions,IN.Coords);
  return pos;
}
```

Upon finishing the advection step, the current particle container can directly be used to render particle primitives at respective positions in the flow domain. Some rendering options, however, require additional particle attributes like the local integration error, velocity or derived flow quantities. In the fragment shader used to advect particles, these quantities are directly computed. This procedure will be discussed in detail later in this paper. Scalar values are stored in the alpha component of the particle container, whereas vector valued information is written to an additional render target using the `render_target` extension [37]. This extension enables the fragment shader to simultaneously write to multiple texture render targets, which can be accessed in upcoming rendering passes.

Whenever the user changes the number of particles, incarnation is repeated using the initial particle container. Changes to the global lifetime or to the position and size of the probe do not affect the flow of operations.

TABLE I
INTEGRATOR PERFORMANCE IN MILLION PARTICLES PER SECOND ON VARYING TEXTURE SIZES FOR 32 BIT FLOATING POINT (8 BIT UNSIGNED CHAR) DATA ON ATI X800 XT.

integrator 32Bit *(integrator 8Bit)*	Euler	RK2	RK3	RK3(2)
CPU all sizes	2.4 (2.4)	1.1 (1.1)	0.73 (0.73)	0.51 (0.51)
GPU 256^2	41 (82)	37 (82)	31 (82)	31 (78)
GPU 512^2	55 (174)	48 (169)	39 (160)	39 (139)
GPU 1024^2	59 (209)	51 (201)	41 (189)	41 (160)

In table I we give timings for the advection of particles using different integration schemes. These timings include all operations that are carried out until updated positions are available in the current particle container. Because on the ATI X800, there is no hardware support for tri-linear interpolation in 32 Bit floating point textures, we give timings for tri-linear 32 Bit floating point interpolation hand–coded in the fragment shader compared to 8 Bit hardware supported tri-linear interpolation. Internally, 32 Bit floating point texture values are represented by 1, 8, and 23 bits for sign, exponent and mantissa, respectively. On the ATI, after reading these values from a texture, they are converted to an ATI specific 24 Bit floating point format used by the shader program for internal computations. This restriction will be dropped on upcoming ATI cards, and it is nonexistent on current *n*VIDIA hardware.

Our timings clearly show the advantages of GPUs for particle integration. In particular, the efficient realization of the memory subsystem including memory bandwidth, caches and parallelization is demonstrated. Even though the number of texture access operations increases from one (Euler) to four (RK3(2)), we do not see a corresponding decrease in performance. Compared to its CPU counterpart, the RK3(2) integrator is about 110 times faster.

C. Particle Rendering

In the particle advection step, new particle positions are computed in the fragment shader and written as RGB colors into the current render target (see Figure 6). To render these positions, different possibilities are available on recent GPUs – OpenGL SuperBuffers and vertex texture fetches using Shader 3.0 or GLSL.

- OpenGL SuperBuffers
  To provide the application program with better control of the GPUs local video memory, the OpenGL SuperBuffer extension [38] has been introduced. It defines a *memory object* that holds a piece of raw video memory. In this paper, we use ATI's UberBuffer which is a preliminary implementation of the OpenGL SuperBuffer. SuperBuffers are currently under consideration for standardization in OpenGL.
  The memory object interface allows the application to allocate graphics memory directly, and to specify how this memory is to be used. This information, in turn, is used by the driver to allocate memory in a format suitable for the requested uses. When the allocated memory is bound to an *attachment point* (a render target, texture, a vertex or color array), no copying takes place. The net effect for the application program therefore is a separation of raw GPU memory from OpenGLs semantic meaning of the data. In our current implementation, a memory object is subsequently bound as the current texture render target and as a vertex array used to draw particle primitives.
- Vertex Texture Fetch
  On traditional graphics architectures, textures could only be accessed in a fragment shader program. The Shader 3.0 and the GLSL specification, finally, also enable texture access in the vertex units hence providing an effective means for displacing geometry on the GPU. This functionality is supported on recent *n*VIDIA graphics hardware.
  To render displaced particles, we render a static vertex array stored in GPU memory. In a vertex shader program the particle position is fetched from the current container, and this position then replaces the position initially stored in the vertex array.

For particle advection, ATIs memory objects and vertex texture fetches on *n*VIDIA cards under Shader 3.0 or GLSL offer similar functionality. The key concept is to let the fragment units generate textures and to use these textures as displacement maps for geometric primitives in subsequent rendering passes. Although the render module of our particle engine can render particles through either of both interfaces, all timings in this paper are given with respect to the implementation using memory objects on the ATI X800 XT graphics card. The reason for this is that currently the ATI card can render

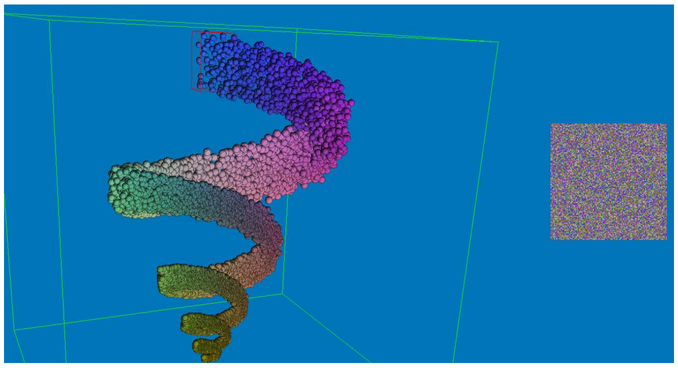

Fig. 6. Particle positions are randomly stored in a 2D texture map, which is then interpreted as a vertex array. Because vertices are rendered in the order of their occurrence in the array, they are displayed in wrong visibility order. The visualization of particles in correct visibility order is shown in Figure 11.

displaced point primitives faster than the *n*VIDIA 6800 Ultra using Shader 3.0. The comparison was done using a 2D texture of size $1k \times 1k$, which was initialized with random positions in normalized device coordinates. Points were rendered on a $1k \times 1k$ view port, and colors were encoded in an additional RGB texture of equal size.

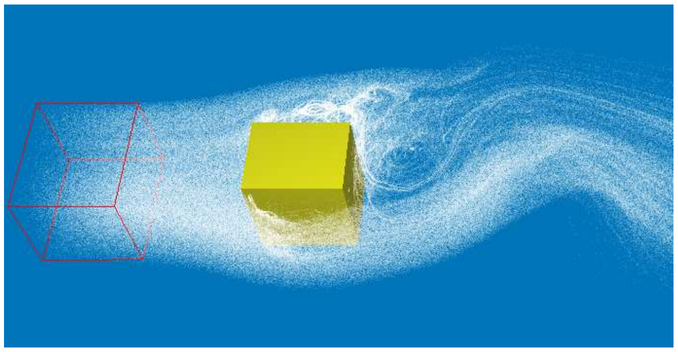

Fig. 7. The data set shows a time step of a 3D time dependent simulation of a turbulent flow around a square block [39]. The simulation was carried out on a rectilinear grid. Particle tracing is performed on this grid, and particle positions are rendered as points.

Once particle positions have been fetched from the current container, the vertex shader transforms these positions according to the viewing parameters. Particles can then be rendered using different modes.

D. Points

Rendering of point primitives does not require any special fragment shader computations, and in particular no texture fetch has to be performed. By using this mode, millions of particles can be displayed at interactive rates. In Figure 7 we show a visualization of the turbulent flow around a block using point primitives. In the remainder of this paper, this flow field will be used to compare the visual effects that are generated by different visualization modes. Although every particle is displayed by a single pixel in screen space, the massive amount of primitives enables the simulation of real-world experiments where small but numerous particles like water vapor or gas droplets are released into the flow. On our current target architecture, the maximum number of particles stored in local video memory that can be sent through the vertex unit and rendered as colored point primitives is 250 millions per second.

E. Oriented Point Sprites

Particularly in a still image, point primitives can not easily reveal flow direction. Even in an animation it is interesting to observe that oriented iconic particle representations like arrows, vector glyphs or ellipsoids provide a much more effective means for showing flow direction. Such geometric representations, however, put the burden to the visualization almost entirely on the geometry subsystem hence limiting the number of particles that can be rendered.

Point sprites, on the other hand, give particles a similar visual appearance to geometric icons, but sprite primitives, in contrast, require far less geometry processing. Conceptually, a point sprite is a textured quadrilateral centered at the points' screen space projection. Only a single vertex is transformed in the vertex units and the rasterizer generates a contiguous block of n by n fragments around this projection. 2D texture coordinates ranging from (0,0) to (1,1) are automatically generated and used to map a given texture image. The upper image of Figure 9 shows particles that are rendered as point sprites. The image of a shaded and lit sphere is used to create the impression of a spatially extended primitive.

While point sprites can effectively render rotationally symmetric particle primitives, they produce incorrect results if used to display arbitrarily shaped geometry. This is mainly due to the loss of degrees of freedom if object transformations are performed after the projection into screen space. To overcome this drawback we employ a texture atlas similar to the one proposed by Guthe et al. [24], but we use a parametrization that is more suitable for a GPU implementation.

The texture atlas contains a 2D array of different views of the 3D particle primitive. Views are parameterized with respect to scaling factor and rotation angle around the y-axis. The parameter domain ranges from 0 to 1 and from 0 to π for scaling factor and rotation angle, respectively (see Figure 8). Here, we assume the particle primitive is aligned with the x-axis, the local direction of the vector field. To get all rotations from 0 to 2π we use the texture wrap mode *mirror*.

For rendering point sprites, a fragment shader transforms the uniform texture coordinates (u,v), which are generated for every fragment covered by this sprite, in such a way as to map into the appropriate atlas sub-image. Therefore the magnitude of the local velocity vector is used as u-offset and the arc sine of the z-component of the normalized vector is used as v-offset. To rotate the selected sub-image around the z-axis we build a rotation matrix $(x, -y)^T$, $(y, x)^T$, where x and y are the first components of the normalized velocity vector. Because all four parameters - texture coordinate offsets and velocity components - are constant for each sprite, these values can be computed in the vertex shader and passed to the fragment shader as a parameter. Because this approach requires the vertex shader to access the flow field, it can only be realized using Shader 3.0 or GLSL. Another alternative

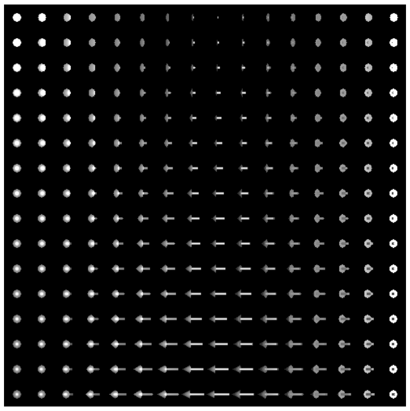

Fig. 8. Low resolution version of the texture atlas, the scale of the model changes from bottom to the top while from left to right different rotations are applied.

is to create an additional particle container, into which these parameters are written by a fragment shader. This container then can be accessed by all fragments that are covered by a point sprite.

By using this method, the virtual geometry can be rotated around two axes and scaled correctly before the rotation takes place. Hence, primitives can point into any spatial direction (see Figure 9). The overhead that is introduced by additional fragment computations is negligible. Nevertheless the direction data to be processed in the vertex shader causes the performance to drop down to about 50% if only one pixel sized points are rendered. In practical applications however, where extended points are used the application becomes fragment bound and the orientation has almost no impact on the overall performance.

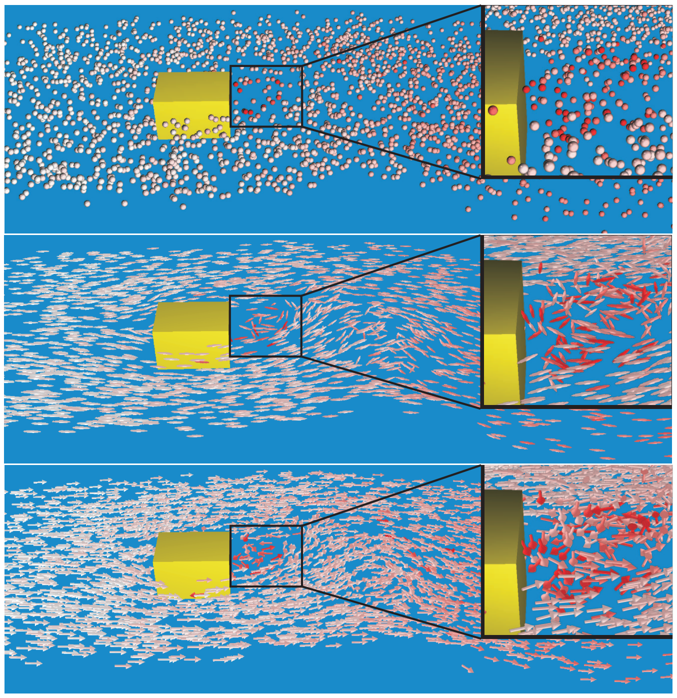

Fig. 9. From top to bottom, flow visualization using simple textured sprites, oriented ellipsoidal sprites and oriented arrow sprites are shown.

IV. Sorting

As many examples have shown, the rendering of particles using transparent point sprites can enhance the visual perception of spatial relationships and it allows for the simultaneous visualization of exterior and interior structures (see Figure 10).

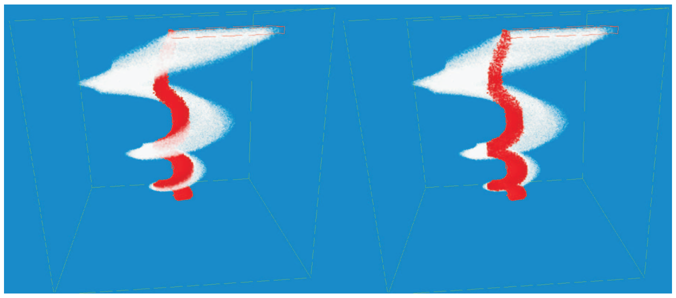

Fig. 10. Velocity magnitude is color coded ranging from opaque red (low) to transparent white (high). First, particles are rendered in correct visibility order. Next, particles are rendered in the order of their incarnation.

The over operator is the most common way to blend transparent objects. It takes into account the color attenuation due to accumulated opacity along the viewing direction. Because the over operator is not commutative, transparent objects need to be depth sorted before they can be rendered in back-to-front visibility order. To avoid read back of data to the CPU, we have integrated a GPU sorting network into our particle engine.

The sorting routine accounts for the architecture of todays graphics processors. Recent GPUs can be thought of as SIMD computers in which a number of processing units simultaneously execute the same instruction on their own data. Considerable effort has been spent on the design of sorting algorithms amenable to the data parallel nature of such architectures. Bitonic merge sort [40] is one of these algorithms. Compared to other sorting algorithms like Quicksort or Heapsort, it is well suited for such architectures because its sequence of operations is fixed and not dependent on the data to be sorted.

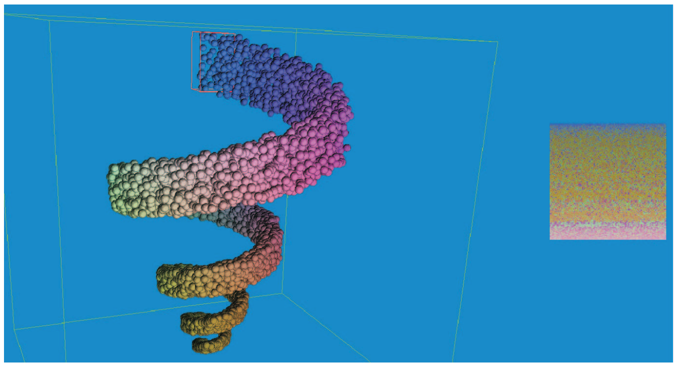

Fig. 11. By sorting texture values according to their distance to the viewer, particles can be rendered in correct visibility order

Purcell et al. [41] and Kipfer and Westermann [42] propose GPU implementations of the bitonic merge sort. The latter approach minimizes both the number of instructions and texture operations to be executed. On the GPU, a texture is built that contains for every particle a floating point sorting key – the distance to the viewer in the current scenario – as well as an integer floating point identifier – a reference into the appropriate position in the current particle container. Both values are stored in the R and G color components, respectively. Because the graphics pipeline as implemented on recent cards is highly optimized for the processing of RGBA samples, two consecutive entries in each row – including sorting key and identifier – are packed into one single RGBA texture sample. Thus, coherence between adjacent entries with respect to memory access and arithmetic operations can be exploited.

Table II compares the GPU bitonic merge sort with an optimized data-dependent sorting routine of the C++ STL. Both algorithms were run on key/index pairs of equal bit width including the final reorder pass to exchange particle positions according to index permutations. As can be seen, the GPU solution has the potential to outperform the CPU solution. Sorting can now be relocated freely between the processor and the graphics card without performance penalty.

TABLE II
SORTING PERFORMANCE

sorter	#keys	megakeys/sec
Bitonic merge sort	256^2	7.2
ATI X800 XT	512^2	6.4
	1024^2	5.1
`std::sort`	256^2	5.4
P4 3.0 GHz	512^2	5.4
	1024^2	5.0

Because sorting becomes the major performance bottleneck in the particle engine, alternative strategies have to be considered. One alternative is to lay out a full sort of particles over multiple rendering passes. Therefore a sorting routine is required that yields "smoother" intermediate results than the bitonic merge sort. The odd–even merge sort is such an algorithm, and it has been shown to be well suited for this purpose [43]. It has the same number of stages and therefore the same complexity as the bitonic merge sort, and we can utilize similar coding optimizations as for the bitonic merge sort. In particular for the rendering of transparent point sprites, the odd–even merge sort gives visually pleasant results even in case the particles set is incompletely sorted. It allows us to spend as much time of one frame as we want for sorting, thus keeping the overall simulation time step within a fixed time limit.

In this scenario, another advantage of GPU sorting becomes apparent. The layout of sorting steps over multiple frames on the CPU still requires the entire particle set to be down- and uploaded from and to the GPU. Due to bandwidth limitations no more than $15/2$ fps (see the experiment above) can be achieved. Using GPU sorting, on the other hand, we can exactly determine the number of sorting steps per simulation pass until sorting becomes the performance bottleneck.

In our particle system, sorting is integrated as an additional rendering pass subsequent to the advection step. A particular shader performs the sort and reorganizes particle positions and attributes in the current container accordingly (see Figure 11). None of the other stages in the particle system are affected by the sort.

V. Interactively Visualizing Derived Flow Attributes

FOR flow analysis, additional flow properties are usually computed and visually encoded. In general, scalar feature volumes containing derived quantities like velocity magnitude, divergence, vorticity magnitude, or higher order flow characteristics are built in a pre-process. They can either be visualized directly or they can provide additional sources for particle attributes in particle based approaches.

Although this kind of pre-processing is advantageous in that generated data sets can be sampled very efficiently, e.g. by means of hardware accelerated tri-linear interpolation, its drawbacks are manifold. First, this approach puts the burden of the visualization process almost entirely on the pre-process prohibiting the use of such techniques in interactive environments like computational steering or in-vivo imaging. Second, due to the enormous memory overhead imposed by additional feature volumes, out-of-core techniques are required for the visualization of time-resolved sequences. Due to limited memory on recent GPUs, the use of additional feature volumes besides the flow field is impossible for reasonably sized data sets.

As the proposed particle engine is supposed to provide an interactive means for the visual analysis of steady or even time-resolved flow on the GPU, any pre-process has to be avoided. Thus, computation of additional flow properties needs to be integrated into the particle engine. Once particle advection has been carried out, in an additional rendering pass a fragment shader computes for each particle the required attribute. The following attributes can be derived:

- **Velocity**: The sampled vector field V.
- **Divergence**: By $\nabla \cdot V$ we compute the extent to which the vector field flow behaves like a source or a sink.
- **Enstrophy**: The circulation per unit area at a point in the flow field is computed as $|\nabla \times V|$.
- $\boldsymbol{\lambda_2}$: It shows the second eigenvalue of the matrix $S^2 + \Omega^2$, where S and Ω is the symmetric and antisymmetric part of the velocity gradient matrix, respectively [44]. A value less than zero indicates a vortex.

Scalar attributes are encoded as colors and they are used as additional rendering attributes. To show λ_2 values, the user selects an iso-value – usually equal to zero – and the opacity of each particle primitive is set such as to highlight the corresponding λ_2 iso-surface. The closer the λ_2 value at the current particle position is to the selected threshold, the more opaque the rendered particle primitives are. A linear modulation function determines the fall off of opacity from the selected threshold. As can be seen in Figure 12, this approach provides an effective means for generating a surface like appearance even though particle tracing is used. Even more importantly, the particle-based approach does not require the computation of an additional scalar feature volume containing λ_2 values.

For computing point-wise flow characteristics two different techniques are currently supported: nearest neighbor fetch and tri-linear interpolation.

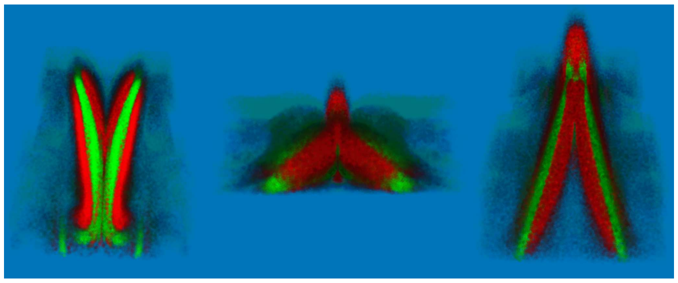

Fig. 12. Vortex structures as seen from different views are extracted on a per particle basis on the GPU. λ_2 values are computed using the fragment shader from Klein et al. [45], and color is used to indicate values less than (red) and greater than (green) zero. The data set of resolution $256^2 \times 50$ shows a simulated flow transition in a boundary layer. For 1/4 million roughly 40 fps.

The nearest neighbor fetch computes the flow quantity only at the grid point closest to the particle position and assigns this value to the particle. Partial derivatives are approximated by central differences, which are computed from adjacent samples in the 3D flow texture. For tri-linear interpolation this procedure is performed eight times – once for each adjacent grid point. Finally, the particle attribute is computed by tri-linear interpolation between these values.

The most time consuming λ_2 shader computes the Jacobian of the vector field and solves a linear system of equations to compute the eigenvalues of the Jacobian. If particle primitives are rendered as points using nearest neighbor fetch, a loss in performance of about 40% is introduced. Tri-linear interpolation slows down the performance about a factor of two. On the other hand, if particles are rendered as sprites the relative loss in performance is quickly becoming negligible. This is due to the fact that the λ_2 value only has to be computed once for every sprite regardless the number of fragments that are generated.

VI. Visualization Geometry

To provide additional visual cues for perceiving complex 3D flow structures, we have integrated stream line construction and rendering into the particle engine. The visualization of stream ribbons has been built on top of this construction process. It is worth noting that we do not expect stream line construction on the GPU to perform faster than optimized software solutions. If numerical integration schemes with adaptive step size control are employed, significantly less samples need to be reconstructed. Such data dependent schemes, however, can not be implemented efficiently on data parallel vertex or fragment processors. As a matter of fact, our system supports stream line construction using a fixed step RK3(2) scheme, but it allows the user to visualize the local truncation error and to reduce the step size accordingly. At the same time the number of integration steps might be increased.

A. Stream Lines

For stream line construction, particle positions are initialized such as to place particles close to each other in adjacent texture samples. In this way, during stream line integration texture cache coherence can be exploited most efficiently. We use two ping-pong buffers – 2D texture maps – that are subsequently interpreted as render target and as container to be read from. Throughout the construction process, the container holds the current position of all particles initially released into the flow. By the procedure described above, as many fragments as there are stream lines are generated – $L \times L$ throughout the following discussion. Both the current container and the 3D texture storing the vector field data are bound to different texture units. In a fragment program, the container is read to retrieve particle positions at the current time step, and the velocity texture is sampled multiple times to numerically integrate to the next positions. Updated positions are rendered into the second ping-pong buffer, which becomes the particle container in the next pass.

To generate stream lines, not only have particle positions to be computed but entire particle traces must be stored. These traces are packed into a texture atlas, which has to be large enough to hold $L \times L \times T$ positions. Here, T is the number of steps to be performed along the trajectories. If this number exceeds the maximum texture size, multiples of these atlases are required.

After the current particle container has been updated, its content is copied into the atlas. Therefore, the atlas is specified as render target, and the rendering output is directed into the appropriate texture area. A simple fragment shader reads the particle container and writes current positions into the atlas. Ping-pong rendering and copying is performed as many times as specified by the user via a maximum line length.

Once particle trajectories have been computed and stored, respective texture samples are interpreted as control points. These points are finally rendered as polylines. Using Shader 3.0 or GLSL, one single polyline consisting of T control points is built prior to stream line construction. To the i^{th} point along the line, the texture atlas position of the i^{th} particle position along the first stream line is assigned by means of 2D texture coordinates. Now the polyline is rendered $L \times L$ times, each time specifying a 2D texture coordinate offset such as to reference stream lines consecutively. In a vertex shader, which receives this offset as a shader constant, the atlas is sampled and the texture value is interpreted as vertex position.

Using OpenGL memory objects, an indexed array containing references into the texture atlas is built. The atlas – interpreted as the coordinate array by OpenGL – is sent through the GPU again, and the indexed array is rendered. In this way, an additional data structure is required to store coordinate indices, but the application program only has to perform one call to change the semantic of the atlas from texture to coordinate array.

In Figure 13, we show stream lines in the reference data set. As can be seen, the particle system generates stream lines of variable length. Because many traces will be terminated before T steps have been carried out, respective fragments become idle in upcoming passes. Fortunately, recent graphics hardware allows one to conditionally exit a fragment shader program thus enabling the corresponding fragment unit to process subsequent items out of the fragment stream. This kind of acceleration, however, only works effectively if contiguous fragment groups are discarded. Due to the coherence between

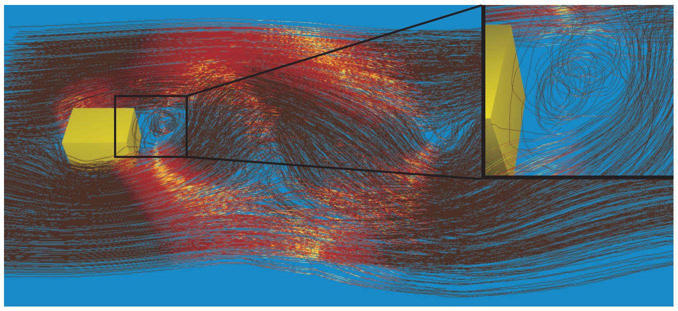
Fig. 13. 1K stream lines of length 1K can be reconstructed and rendered at 20 fps

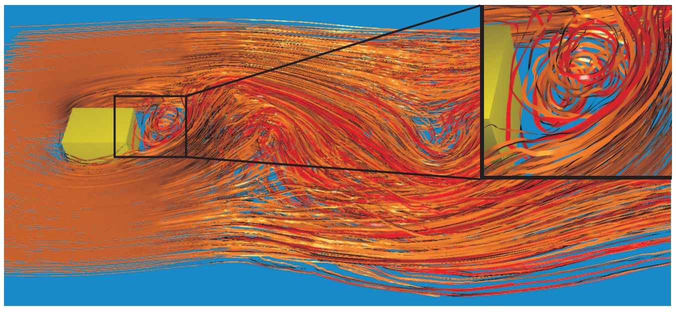
Fig. 14. 1K stream bands of length 1K can be reconstructed and rendered at 14 fps

stream lines released at adjacent positions in the probe texture, likely this likely in the current applications.

B. Stream Ribbons

While stream lines illustrate the direction of flow, stream ribbons also show rotation about the flow axis by twisting a ribbon-shaped geometry [16]. Starting with a random normal vector at the seed point of a particle trajectory, at subsequent points the incremental rotation of this vector according to the rotation about the flow direction is computed as

$$\theta_{i+1} = \theta_i \cdot \frac{1}{2}(rot(\vec{v}) \times v')$$

Here, $rot(\vec{v}) = \nabla \times v$ is the rotation of the vector field and v' is the normalized flow direction. The sum of all increment angles up to a certain point on the trajectory is used to rotate the normalized projection of the previous normal vector into the plane orthogonal to the flow direction at this point. By rendering points on the trajectory and end points of the normal projection vectors in alternative order, a ribbon-shaped triangle strip is formed.

On the GPU, we construct particle traces as described above, and we use the alpha component of each texture element to store the accumulated increment angle. All four components – particle position and increment – are finally written into the texture atlas. During ping-pong rendering, positions and increments are updated.

To generate stream bands we proceed as follows. After having computed the atlas containing positions and increment angles, a second atlas is built that contains the other rim of each stream band. A fragment shader reads from the original atlas, rotates the initial normal vector according to the accumulated increment angles and writes the end points of the rotated vectors into the second atlas. To display the bands using Shader 3.0 or GLSL, a pre-computed triangle strip is rendered T times, each time taking the respective displacement values from either atlas. Using OpenGL memory objects, a texture twice as large as the atlas is generated, which contains the respective control points in correct order. This texture is interpreted as a vertex array, and it is rendered as a set of triangle strips.

In Figure 14, stream ribbons as they occur in the reference data set are displayed. Due to the extra work to be done for computing accumulated increment angles, rotating projected normal vectors as well as reading and writing the second texture atlas, timings are of about a factor of 30% slower compared to the construction and rendering of stream lines. To shade the ribbons, a vertex normal is computed in the fragment shader during the construction step. These normals are stored in a separate texture, and they are sampled in the geometry units at the time the triangle strip is rendered.

VII. Conclusion and Future Work

As demonstrated in the key experiment, the proposed GPU particle engine significantly outperforms CPU engines for flow visualization. In a number of different examples this statement could be verified. The ability to trace massive particle sets at interactive rates in combination with alternative visualization options to reveal local flow properties enables virtual exploration of large fields in a way similar to real-world experiments. The possibility to integrate numerically and data intensive computations for flow analysis into the rendering process distinguishes the GPU engine from previous approaches.

Besides particle advection, the engine provides a variety of visualization options to visually convey relevant structures in 3D steady flow fields. Even though we did not yet compare the performance of these techniques to optimized CPU implementations, our experiments have shown interactive construction and rendering of large scale visualization geometry in realistically sized flow fields.

As the images in the color plate demonstrate, the particle engine also provides an effective means for visualizing 2D flows. By using massive particle sets in combination with oriented sprites, LIC-like visualizations can be achieved at interactive rates. This includes higher-order integration schemes thus providing numerically accurate particle traces. In the future, we plan to extend the engine for the visualization of flow on triangular surfaces.

In addition to flow fields given on Cartesian grids, visualization techniques need to be capable of dealing with non-uniform or structured grids. For instance, the Navier-Stokes simulation of turbulent flow, which is shown in Figures 4 to 14, was carried out on a rectilinear grid. Along two coordinate axes the grid is refined towards the region right behind the block. To visualize this data set on the GPU, we built two textures that define the mapping from physical coordinates to non-uniformly sized grid cells. The resolution of these textures is chosen with respect to the smallest cell size. Note that we do not resample the grid itself, but we store the non-linear 1D mapping for each axis separately. The additional memory requirement imposed by this approach is insignificant. On the GPU, prior to accessing the vector field data, two additional texture fetches have to be carried out to transform the local particle position to the computational domain. In the current scenario, the extra texture fetches slow down the performance by a factor of two.

With regard to more general grid structures, we believe that a similar performance gain as seen for uniform grids can be achieved. This assumption is evidenced in the timing statistics given in this paper. It has been shown, that for both computation and memory bandwidth bound applications the GPU outperforms CPU based particle integration schemes. As integration in non-uniform grids requires additional numerical and memory access operations, i.e. evaluation of adjacency information, transformation of points and velocities, (barycentric) interpolation, computation of Jacobians and inverse Jacobians, improved performance on current and upcoming graphics cards can be expected.

A GPU data structure that accommodates particle tracing in tetrahedral grids was presented by Weiler et al. [46]. Although in a different setting, it was shown that essentially the same data structures can be implemented on the GPU than on the CPU. By encoding adjacency information in additional texture maps and by clipping particle paths against element faces, GPU particle tracing in tetrahedral grids should be possible in a very similar way to a CPU implementation. As barycentric interpolation in simplicial elements can be performed efficiently on the GPU, local exact integration in tetrahedral grids is possible as well [13].

In addition, the ability to perform conditional per-fragment computations on the GPU has particular advantages for rendering non-uniform grid structures. This functionality allows one to aboard fragment shader computations and to make available the fragment processor for operations on the next fragment. Adaptive algorithms like the stencil walk for point location in P-space will greatly benefit from this functionality.

Multiblock grids, on the other hand, require new approaches to leverage current GPU architectures most efficiently. Such grids consist of multiple, potentially overlapping uniform grids of varying size and resolution. Locating the block a particle is passing through takes a significant amount of work in traversing multiblock grids. Such tests could be realized efficiently by using rendering functionality on current graphics cards. By rendering the bounding boxes of each block, the viewing distance of respective front and back faces as well as a unique block id can be drawn to an intermediate texture. This information, in turn, can be accessed by particles to determine whether they are inside or outside the rendered block. To avoid repeated executions of this process, as many blocks as possible have to be processed in one pass. In particular, by rendering all blocks, depth peeling could be utilized to discard those blocks that have already been processed.

Acknowledgment

This work was financially supported by the German Research Foundation (DFG) in the Priority Programm 1147 “Bildgebende Messverfahren für die Strömungsmechanik”. We would like to thank Simon Stegmaier from the University Stuttgart for providing the λ_2-shader.

References

[1] NASA Ames Research Center, “Windtunnel Experiments,” http://windtunnels.arc.nasa.gov/.

[2] A. Adrian, “Particle imaging techniques for experimental fluid mechanics,” *Annual Review of Fluid Mechanics*, vol. 23, pp. 261–304, 2003.

[3] L. Hesselink and T. Delmarcelle, *Scientific visualization - advances and challenges*. Academic Press, 1994, ch. Visualization of vector and tensor data sets, pp. 367–390.

[4] R. Peikert and M. Roth, “The ’parallel vectors’ operator - a vector field visualization primitive,” in *Proceedings IEEE Visualization 99*, 1999, pp. 263–271.

[5] G. Scheuermann and X. Tricoche, “Topological methods for flow visualization,” in *Visualization Handbook*, C. Johnson and C. Hansen, Eds. Academic Press, 2005.

[6] P. G. Buning, “Numerical algorithms in CFD post-processing,” van Karman Institute for Fluid Dynamics, pp. 1–20, 1989.

[7] R. Haimes and D. Darmofal, “Visualization in computational fluid dynamics: a case study,” in *Proceedings of the 2nd conference on Visualization ’91*, 1991, pp. 392–397.

[8] F. Post, “Fluid flow visualization,” *Focus on Scientific Visualization*, pp. 1–40, 1993.

[9] C. Teitzel, M. Hopf, and T. Ertl, “Scientific visualization on sparse grids,” in *Proceedings of Scientific Visualization - Dagstuhl ’97, Heidelberg*, H. H. G. M. Nielson and F. Post, Eds. IEEE Computer Society, IEEE Computer Society Press, 2000, pp. 284–295.

[10] A. Sadarjoen, T. van Walsum, A. Hin, and F. Post, “Particle Tracing Algorithms for 3-D Curvilinear Grids,” in *Proc. 5th Eurographics Workshop on Visualization in Scientific Computing*, 1994.

[11] J. P. M. Hultquist, “Constructing stream surfaces in steady 3D vector fields,” in *Proceedings of the 3rd conference on Visualization ’92*, IEEE. IEEE Computer Society Press, 1992, pp. 171–178.

[12] D. N. Kenwright and D. A. Lane, “Interactive time-dependent particle tracing using tetrahedral decomposition,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 2, pp. 120–129, 1996.

[13] M. Nielson, I.-H. Jung, N. Srinivasan, J. Sung, and J.-B. Yoon, “Tools for computing tangent curves and topological graphs for visualizing piecewise linearly varying vector fields over triangulated domains,” in *Scientific Visualization: Overviews, Methodologies and Techniques*, M. Nielson, H. Hagen, and H. Müller, Eds. IEEE Computer Society Press, Los Alamitos, California, 1997, pp. 527–562.

[14] P. Kipfer, F. Reck, and G. Greiner, “Local exact particle tracing on unstructured grids,” *Computer Graphics Forum*, vol. 22, no. 2, pp. 133–142, june 2003.

[15] C. Teitzel, R. Grosso, and T. Ertl, “Efficient and Reliable Integration Methods for Particle Tracing in Unsteady Flows on Discrete Meshes,” in *Proc. 8th Eurographics Workshop on Visualization in Scientific Computing*, W. Lefer and M. Grave, Eds. IEEE, 1997, pp. 49–56.

[16] S. Ueng, K. Sikorski, and K. Ma, “Efficient streamline, streamribbon, and streamtube constructions on unstructured grids,” in *Transactions on Visualization and Computer Graphics*. IEEE, 1996, pp. 2:100–110.

[17] S. Bryson and C. Levit, “The virtual windtunnel: an environment for the exploration of three-dimensional unsteady flows,” in *Proceedings IEEE Visualization*, 1991, pp. 17–24.

[18] M. Malte Zöckler, D. Stalling, and H.-C. Hege, “Parallel line integral convolution,” *Parallel Computing*, vol. 23, no. 7, pp. 975–989, 1997.

[19] R. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy, “Real-time out-of-core visualization of particle traces,” in *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, 2001, pp. 45–50.

[20] M. Cox and D. Ellsworth, “Application-controlled demand paging for out-of-core visualization,” in *Proceedings IEEE Visualization*, 1997, pp. 235–244.

[21] M. Zöckler, D. Stalling, and H. Hege, “Interactive visualization of 3d-vector fields using illuminated stream lines,” in *Proceedings of Visualization ’96*. ACM, 1996, pp. 107–113.

[22] H. Hauser, R. S. Laramee, and H. Doleisch, “State-of-the-art report 2002 in flow visualization,” VRVis Research Center, Vienna, Tech. Rep. TR-VRVis-2002-003, 2002.

[23] O. Mattausch, T. Theussl, H. Hauser, and E. Gröller, “Strategies for Interactive Exploration of 3D Flow Using Evenly-Spaced Illuminated Streamlines,” in *Proceedings of SCCG*, 2003, pp. 213–222.

[24] S. Guthe, S. Gumhold, and W. Strasser, “Interactive visualization of volumetric vector fields using texture based particles,” in *Proceedings of WSCG*, vol. 10, 2002, pp. 33–41.

[25] M. Brill, H. Hagen, H.-C. Rodrian, W. Djatschin, and S. V. Klimenko, “Streamball techniques for flow visualization,” in *Proceedings of the conference on Visualization ’94*, 1994, pp. 225–231.

[26] J. van Wijk, “Spot noise: Texture synthesis for data visualization,” *Computer Graphics*, vol. 25, no. 4, pp. 309–318, 1991.

[27] R. Crawfis and N. Max, “Direct volume visualization of three-dimensional vector fields,” in *Proceedings ACM Workshop on Volume Visualization*, 1992, pp. 55–60.

[28] ——, “Texture splats for 3D scalar and vector field visualization,” in *Proceedings IEEE Visualization 93*, 1993, pp. 261–265.

[29] N. Max, R. Crawfis, and C. Grant, “Visualizing 3D Velocity Fields Near Contour Surfaces,” in *Proceedings IEEE Visualization 94*, 1994, pp. 248–255.

[30] D. Stalling and H.-C. Hege, “Fast and resolution independent line integral convolution,” in *Computer Graphics (SIGGRAPH 95 Proceedings)*, 1995, pp. 249–256.

[31] J. J. van Wijk, “Image based flow visualization,” in *Proceedings Visualization*. IEEE, 2002.

[32] V. Interrante and C. Grosch, “Visualizing 3D Flow,” *Computer Graphics and Applications*, vol. 18, no. 4, pp. 49–53, 1998.

[33] C. Rezk-Salama, P. Hastreiter, and T. Ertl, “Interactive exploration of volume line integral convolution based on 3D-texture mapping,” in *Proceedings IEEE Visualization 99*, 1999, pp. 233–240.

[34] A. Telea and J. J. van Wijk, “3D IBFV: Hardware-Accelerated 3D Flow Visualization,” in *Proceedings Conference on Visualization 2003*. IEEE, 2003, pp. 233–240.

[35] G.-S. Li, U. Bordoloi, and H.-W. Shen, “Chameleon: An interactive texture-based rendering framework for visualizing three-dimensional vector fields,” in *Visualization 2003*. IEEE, 2003, pp. 241 – 248.

[36] nVidia, “nVidia White Papers,” http://www.nvidia.com/Developer/Cg.

[37] OpenGL Architecture Review Board, “`GL_EXT_render_target`,” http://www.opengl.org/resources/features/GL_EXT_render_target.txt.

[38] nVidia, “Data Storage and Transfer in OpenGL,” http://developer.nvidia.com/docs/IO/8229/Data-Xfer-Store.pdf.

[39] R. Verstappen and A. Veldman, “Spectro-consistent discretization of Navier-Stokes: a challenge to RANS and LES,” *Journal of Engineering Mathematics*, no. 1, pp. 163–179, 1998.

[40] K. Batcher, “Sorting networks and their applications,” in *Proceedings AFIPS 1968*, 1968.

[41] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan, “Photon mapping on programmable graphics hardware,” in *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2003, pp. 41–50.

[42] P. Kipfer, M. Segal, and R. Westermann, “Uberflow: A GPU-based particle engine,” in *Proceedings Eurographics Graphics Hardware Conference*, T. Akenine-Möller and M. McCool, Eds. IEEE, 2004, pp. 115–122.

[43] A. Kolb, L. Latta, and C. Rezk-Salama, “Hardware-based simulation and collision detection for large particle systems,” in *Proceedings Eurographics Graphics Hardware Conference*, T. Akenine-Möller and M. McCool, Eds. IEEE, 2004, pp. 123–131.

[44] J. Jeong and F. Hussain, “On the identification of a vortex,” *Journal of Fluid Mechanics*, vol. 285, pp. 69–94, 1995.

[45] T. Klein, S. Stegmaier, and T. Ertl, “Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids,” in *Proceedings of Pacific Graphics ’04*, 2004, pp. 186–195.

[46] M. Weiler, M. Kraus, M. Merz, and T. Ertl, “Hardware-Based Ray Casting for Tetrahedral Meshes,” in *Procceedings of IEEE Visualization ’03*. IEEE, 2003, pp. 333–340.

Jens Krüger is a PhD student at the computer graphics and visualization group headed by Professor Rüdiger Westermann at the Technische Universität München. There he works on GPU based techniques for computer graphics and visualization. Jens’ current research is focused on GPU solutions for numerical problems, often arising in physically-based simulations. He has published papers on GPU programming at internationally recognized conferences like ACM SIGGRAPH or IEEE VISUALIZATION. In 2004 he received the ATI fellowship award, which honored him as an outstanding graduate student in areas related to computer graphics and graphics systems.

Peter Kipfer is a post-doc researcher at the Computer Graphics & Visualization Group at the Technische Universität München. He received his Ph.D. from the University of Erlangen-Nürnberg in 2003 for his work on parallel and distributed visualization and rendering within the KONWIHR supercomputing project. His current research focuses on general purpose computing and geometry processing on the GPU.

Polina Kondratieva is a PhD student at the computer graphics and visualization group headed by Professor Rüdiger Westermann. She works on GPU/CPU implementation of algorithms for visualization and image processing. Particularly, her research is focused on the development of GPU based solutions for the reconstruction of flow fields from image pairs and flow visualization.

Rüdiger Westermann studied computer science at the Technical University Darmstadt, Germany. He pursued his Doctoral thesis on multiresolution techniques in volume rendering, and he received a PhD in computer science from the University of Dortmund, Germany. In 1999, he was a visiting professor at the University of Utah in Salt Lake City, and he became an assistant professor at the University of Stuttgart, Germany. In 2000, he was appointed an associate professor at the Technical University Aachen, Germany, where he was head of the Scientific Visualization and Imaging Group. In 2002, Westermann was appointed the chair of Computer Graphics and Visualization at the Technische Universität München. His research interests include general purpose computing on GPUs, hardware accelerated visualization and image synthesis, hierarchical methods in scientific visualization, volume rendering, flow visualization and parallel graphics algorithms.

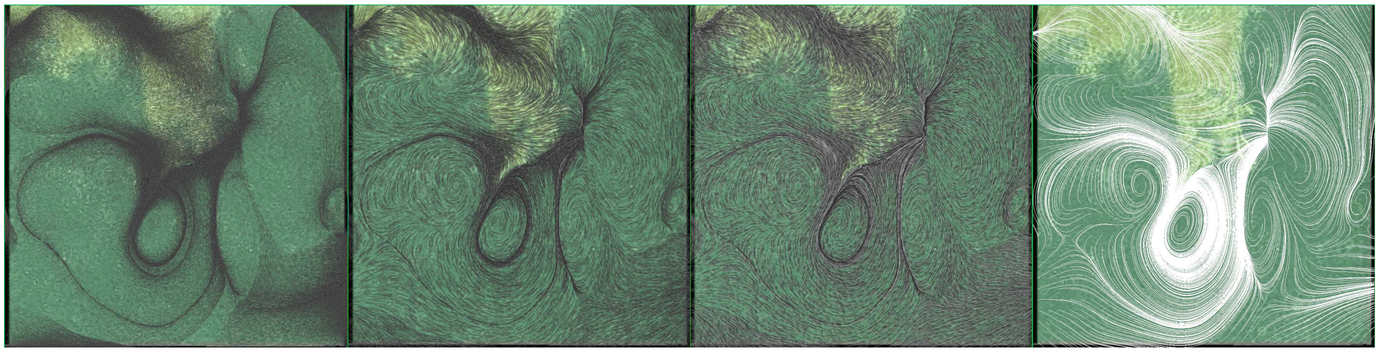

Fig. 15. *Particle-based visualization of 2D flow on a 512×512 grid. The flow is induced by micro-biological structures, which are displayed in the background image. From left to right, transparent elipsoidal sprites, enlarged transparent ellipsoidal sprites, transparent arrows and stream lines are shown. In regions of low velocity, multiple sprites overlap and produce a more dense appearance. On a $1K \times 1K$ viewport and by advecting 250K particles using a RK3(2) integration scheme in the leftmost image, the animation runs at 28 fps on the ATI X800 XT graphics card.*

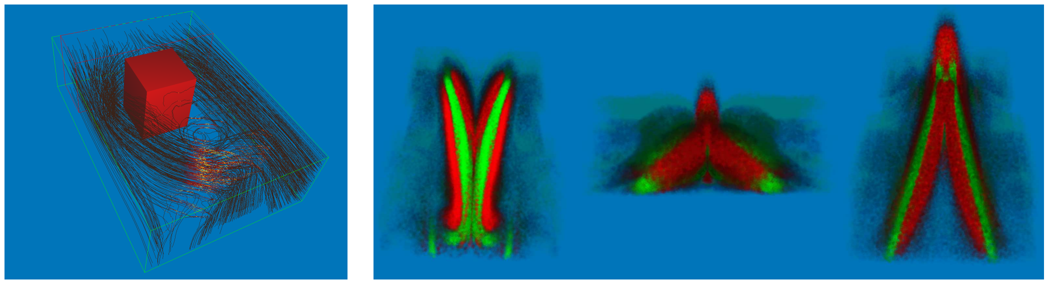

Fig. 16. *On the left, the turbulent flow around a block is visualized using shaded lines. The data is given on a rectilinear grid. Construction and rendering of 512 stream lines can be performed 35 times per second. On the right, the flow transition in a boundary layer is visualized. Red(green) depicts λ_2 values less(greater) than zero. Transparency is fading out with increasing λ_2 magnitude. Using 250K particles, the particle probe can be positioned interactively at 30 fps.*

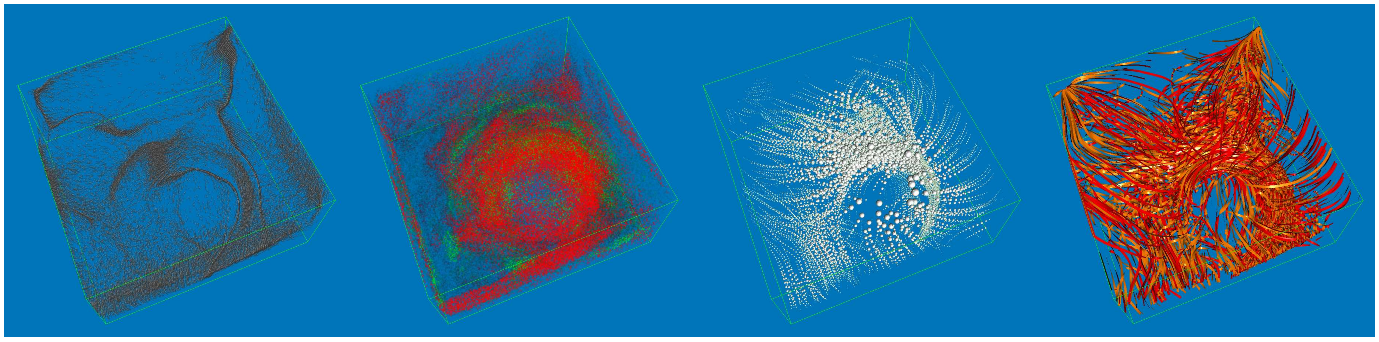

Fig. 17. *Hurricane Isabelle is visualized using different visualization options. From left to right, transparent point sprites, λ_2 color coded points, stream balls and stream ribbons are shown.*