Editors: Cláudio T. Silva, csilva@nyu.edu
Daniel Weiskopf, weiskopf@visus.uni-stuttgart.de

TEXTURE-BASED FLOW VISUALIZATION

By Rudolf Netzel and Daniel Weiskopf

Texture-based visualization is a powerful and versatile tool for depicting steady and unsteady flow.

The visualization of flow is commonly used in many areas of science and engineering. The illustration of cloud movement in weather forecasts is one example that we see on a daily basis. More advanced variants are employed for the visual analysis of the simulation results from computational fluid dynamics (CFD) by simulation experts in fields like automotive engineering or environmental sciences.

There are many different approaches to flow visualization. We can distinguish between point-based direct visual mappings (for example, by arrow glyphs that show the flow at the location of the glyph), particle-tracing methods (for example, streamline, streakline, or pathline-based visualization), and feature-based visualization (for example, methods that extract and show vortices).[1] Particle-tracing methods are popular because they're able to show the transport of virtual (massless) marker particles with the flow. Therefore, they resemble experimental flow visualization and are quite intuitive to understand and use. One example that almost perfectly mimics experimental methods is dye-based flow visualization.[2] Other, more abstract visualizations just draw a few curves of streamlines or pathlines.

However, one of the grand challenges of any of the particle-tracing methods is the seeding problem: Where should we place the seed points for particle tracing? If those positions aren't properly chosen, the particle traces could miss relevant parts of the flow—there's just no visual information if there's no particle trace (see Figure 1a). The typical particle-tracing methods employ a sparse representation: they just show a few of the particle traces that sparsely cover the domain. The sparser the representation, the more challenging the seeding problem is. With this observation in mind, we could just follow the opposite strategy by going for a dense visual representation. Why not draw streamlines "everywhere?" In this way, no parts of the flow are missed and the seeding problem just doesn't appear (see Figure 1b). Such dense representation is best achieved by texture-based flow visualization.[3]

Here, our goal is to explain the principles and basics of texture-based flow visualization, show its advantages and challenges, and provide hands-on descriptions of how it can be implemented and used.

(a) **(b)**

Figure 1. (a) Sparse versus (b) dense representation. The sparse representation misses showing features such as small vortices and flow saddle points.

Line Integral Convolution

To accomplish a dense representation, computed from an input image, it's necessary to assign every point in the final image a value that encodes the local flow or aids the viewer in understanding the flow behavior. The input and output images are called *textures* in this context. A single element of the texture is a *texel*, analogous to the pixel of a picture.

Line integral convolution (LIC)[4] is the role model of texture-based flow visualization. LIC was originally designed for steady flow. First, we'll

restrict ourselves to steady 2D flow and later relax those restrictions for more general applications. The LIC process takes a texture and flow field as inputs, and outputs the final texture that shows the visualization. Typically, the input texture is a white-noise image. Figure 2 illustrates the LIC process.

LIC makes use of the ability of the human visual system to recognize curves according to spatial frequencies. Texel values are convolved along streamlines with a kernel that implements a low-pass filter. In this way, LIC leads to low spatial frequencies along the streamline—it blurs the input image along the streamlines. The original spatial frequency of the input image is preserved orthogonal to the streamlines; the typical example of white-noise input has very high spatial frequency. This difference in spatial frequencies along and perpendicular to the lines allows our human visual system to see those lines.

Mathematically, the LIC computation reads:

$$O(\mathbf{z}) = \int_{-\infty}^{\infty} k(s) I(\mathbf{x}(s; \mathbf{z})) ds.$$

Here, O is the output texture, I is the input texture, k the kernel. The streamline $\mathbf{x}(s; \mathbf{z})$ depends on the seed point $\mathbf{z}$ and is parameterized by s. The streamline is computed by particle tracing, solving the ordinary differential equation that integrates along the flow:

$$\frac{d\mathbf{x}(s; \mathbf{z})}{ds} = \mathbf{v}(\mathbf{x}(s; \mathbf{z}))$$

with the initial condition $\mathbf{x}(0; \mathbf{z}) = \mathbf{z}$.

The convolution integral of LIC covers an infinitely long distance. In practice, the kernel has finite support, effectively reducing the integration to the length of that support.

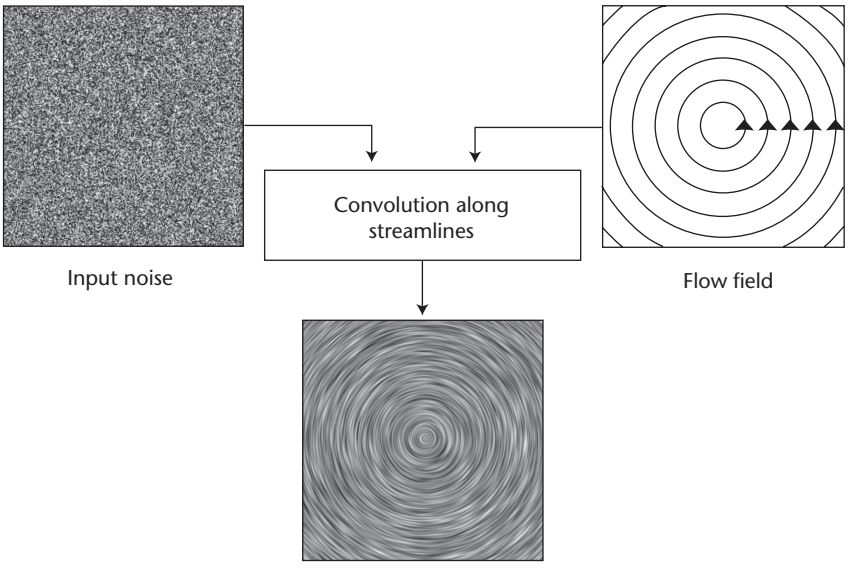

Figure 2. Schematic process of line integral convolution (LIC) computation: It takes a noise texture and the flow field as input, and blurs the noise texture along the streamlines by applying the line integral convolution.

The computation of LIC does convolution and particle tracing simultaneously. We typically use an explicit solver such as fourth-order Runge-Kutta (or even just the Euler method) with constant step size. While stepping along the streamline with the explicit solver, the discretized convolution integral is being accumulated.

This computation is embarrassingly simple to parallelize, because particle tracing and convolution are independent for each texel. The sidebar "LIC on the GPU" shows the straightforward, highly parallel implementation on GPUs. This parallelization helps solve the issue of high computational costs: for each texel, the full particle tracing and convolution must be computed.

Control of Visualization Parameters

The "look" of the LIC output is controlled by the choice of input texture and kernel. The input texture essentially plays the role of the seeds in a traditional streamline visualization, because the texels of the input texture are smeared out along the streamlines. White noise is the typical choice for a fully dense representation, because it covers the whole domain with "seeds." White noise contains all spatial frequencies up to those limited by the spacing (resolution) of the input texture. However, LIC works regardless of the type of input texture. For example, LIC can even mimic sparse streamline rendering, when using a sparse noise texture: a texture that consists of just a few randomly distributed dots. Figure 3 shows the effect of the choice of input noise.

The choice of kernel affects the look of the streamlines. The length of the kernel's support controls the length of the streamlines. The kernel's shape determines the profile along streamlines and the characteristics of the LIC filter. Typically, we want to have a kernel with characteristics of a low-pass filter to allow the human visual system to recognize lines in the texture. A popular choice of the kernel is the box (or rectangle) filter. The advantage of the box filter is its very simple computation (all weights along the streamline are identical), which can be utilized for accelerated computations by Fast LIC.[5] Another common choice is an exponential drop-off, which implicitly comes with the incremental computation via texture advection, as we'll describe in a bit. Finally, the (truncated) Gaussian

LIC ON THE GPU

Line integral convolution (LIC) computation is performed for each texel independently, using the same computational kernel. Therefore, an almost trivial implementation on graphics processing units (GPUs) is possible. We briefly illustrate the main parts of a typical OpenGL shader-based implementation (OpenGL 3.3; see www.opengl.org for background reading). The computation consists of two elements. The first one renders a domain-filling quadrilateral (directly to the viewport for immediate visualization or to a framebuffer object for off-screen rendering). The second element is the per-texel computation, implemented by a fragment shader. This shader is executed for each LIC texel when the quadrilateral is rasterized.

Figure A shows the relevant parts of the fragment shader. For each texel, the convolution with the noise texture is computed along a streamline. The current texel is the center of the convolution kernel and is initialized with its weighted texture value, followed by two loops that compute the rest of the weighted texel value sum that approximates the convolution integral. The first loop follows the streamline in the downstream direction until the kernel's end is reached. The second loop follows the streamline in the upstream direction. The convolution and tracing of the streamline are computed simultaneously in those loops. In this version of the code, we adapt the final color to normalize the sum of the kernel weights to one. In this way, we guarantee that the overall brightness of the noise texture isn't changed by the convolution.

```
float kernel_weight;
float current_kernel_position;
vec2 current_tex_pos;
vec3 texColor;
vec4 weighted_color;
vec4 final_color;

//calculate steps, assuming an odd number for the kernel length
int integration_steps = (kernel_length-1)/2;
float kernel_step = 1.0/float(kernel_length-1);

//calculate initial value at center
texColor = texture(tex, texCoords).rgb;   // lookup in noise texture
kernel_weight = texture(kernel, 0.5).r;   // lookup in kernel texture
final_color = vec4(kernel_weight*texColor, kernel_weight);

//process first half of the kernel in downstream direction
current_tex_pos = texCoords + RK4(texCoords);
current_kernel_position = 0.5 + kernel_step;

for(int i = 1; i <= integration_steps; i++){
          // accumulate color for the convolution integral
          texColor = texture(tex, current_tex_pos).rgb;
          kernel_weight = texture(kernel, current_kernel_position).r;
          weighted_color = vec4(kernel_weight*texColor, kernel_weight);
          final_color = final_color + weighted_color;

          // one step ahead along the streamline
          current_kernel_position = current_kernel_position + kernel_step;
          current_tex_pos = current_tex_pos + RK4(current_tex_pos);
}

//process second half of the kernel in upstream direction, similar to the loop above
//...

//normalize the sum of the weights to 1.0
final_color = final_color/final_color.a;
gl_FragColor = final_color;
```

Figure A. The relevant part of the OpenGL fragment shader for LIC. After the initialization part, the shader loops along the downstream and upstream directions of the streamlines to perform the particle tracing and convolution.

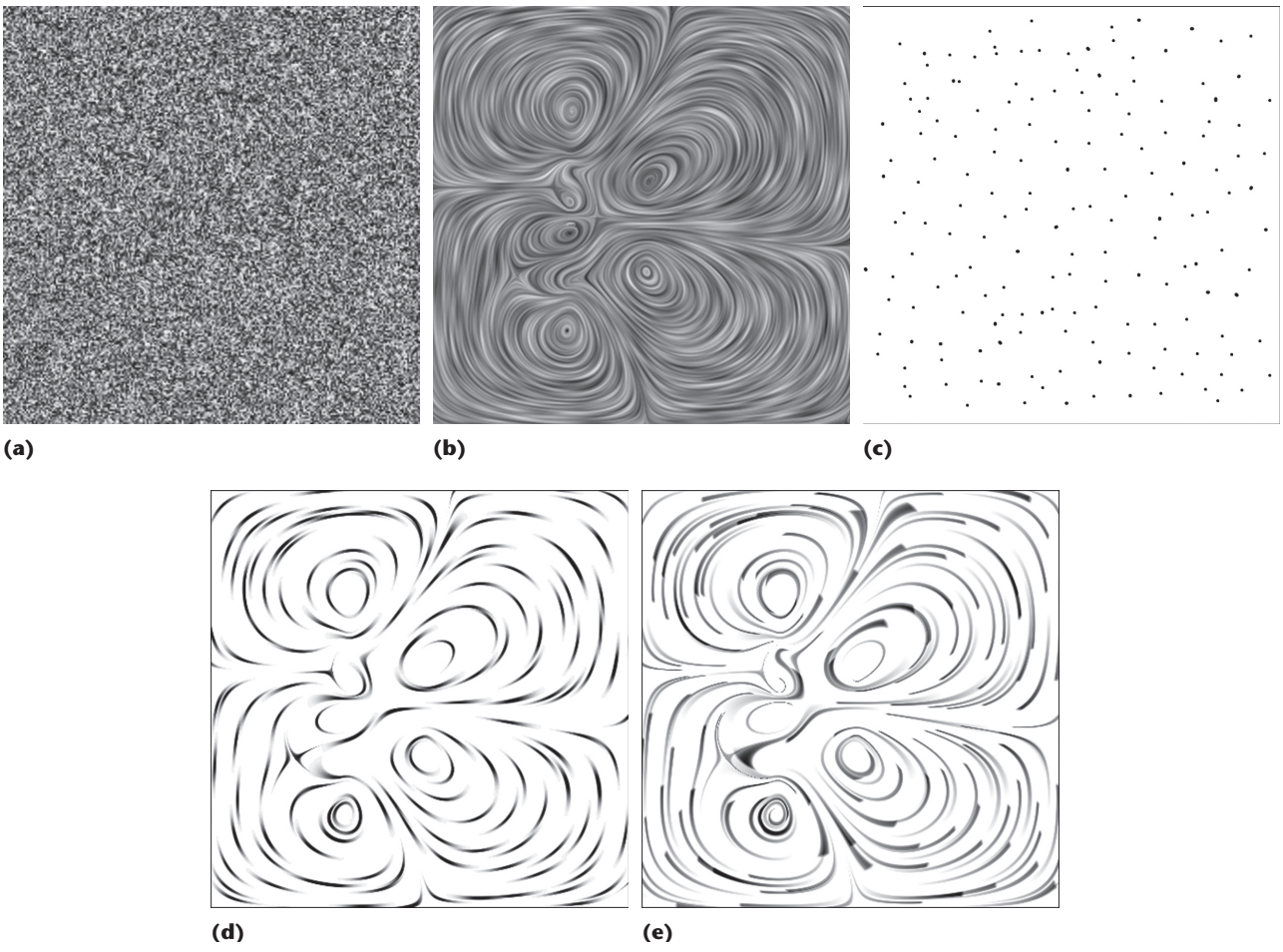

Figure 3. Impact of choice of input noise: (a) white noise texture, (b) LIC result for that white noise input, (c) sparse noise texture, (d) LIC result for sparse noise, and (e) LIC result for sparse noise, but with an asymmetric filter kernel that shows the downstream direction.

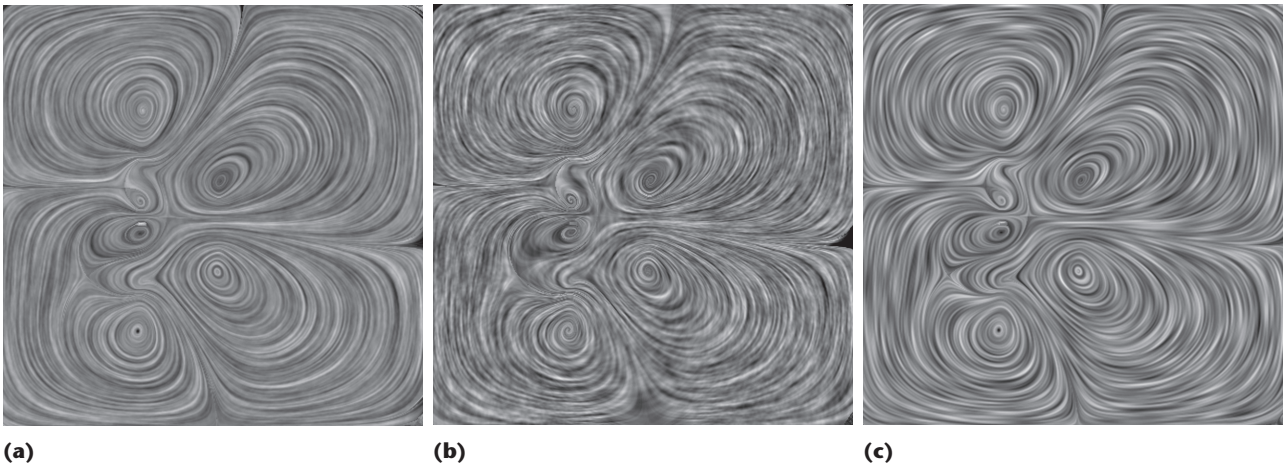

Figure 4. Effects of different kernels: (a) box filter, (b) exponential drop-off, (c) Gaussian.

function is often employed. We recommend using the Gaussian kernel, because it has much better low-pass filter characteristics than the other two filters. In fact, Gaussian is the best compromise between drop-off in the frequency domain (that is, yielding good low-pass characteristics) and the spatial domain (reducing the convolution length). Figure 4 compares the effects of those three kernels. A more detailed discussion of

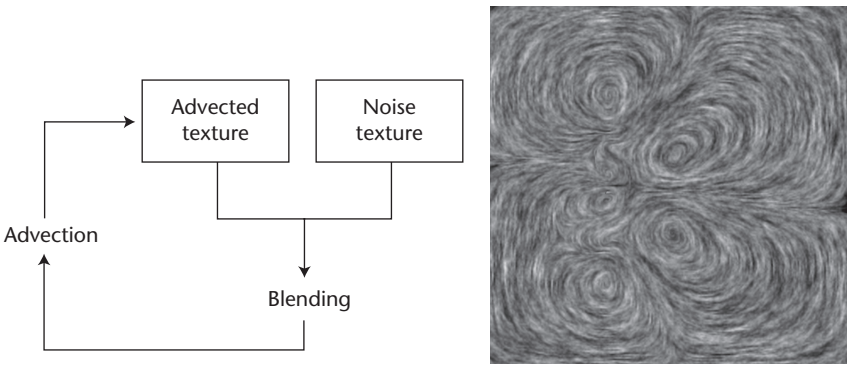

Figure 5. (a) Workflow of texture advection with (b) an example result.

the frequency characteristics can be found elsewhere.[6]

With a time-dependent kernel, even animation along streamlines can be modeled[4]: the kernel is the result of convolving a sinusoidal function with a Gaussian function. The animation time is used to change the phase of the sinusoidal function. With phase shifting, the smeared-out LIC streamlines shift their texture along the streamlines, leading to the visual effect of motion along the streamlines.

With sparse input noise, the low-pass characteristics become less important and the profile gains relevance. For example, Oriented LIC (OLIC)[7] uses asymmetric kernels to show the upstream and downstream directions. Figure 3e shows an example of OLIC.

Texture Advection

Instead of computing streamlines at full length and performing the convolution along those curves, we can also use advection as the driving principle. Texture advection uses a texture to represent quantities to be transported. The continuity equation describes the transport of such quantity ρ with time t along the driving flow $\mathbf{v}$:

$$\frac{\partial \rho(\mathbf{x},t)}{\partial t} + \nabla \cdot (\rho(\mathbf{x},t)\mathbf{v}(\mathbf{x},t)) = 0.$$

This partial differential equation could be solved by adopting an Eulerian view; Grzegorz Karch and his colleagues describe respective numerical methods in the context of dye advection.[2] However, most texture-based flow visualization techniques instead use a semi-Lagrangian approach.[8,9] They solve the advection equation by employing the method of characteristics: the partial differential equation is reduced to a family of ordinary differential equations. Here, the ordinary differential equations are just the particle-tracing equations with the seed points set to the texel position for which the advection is computed. Typically, we use backward semi-Lagrangian advection:

$$\rho(\mathbf{x},t) = \rho(\mathbf{x} - \Delta\mathbf{x}, t - \Delta t).$$

For the step size Δt, integration backward in time yields the previous position $\mathbf{x} - \Delta\mathbf{x}$. This particle tracing introduces the Lagrangian perspective. The Eulerian perspective is still here, as well: In the Eulerian view, there are no individual particles; instead they're indirectly represented by their density distribution ρ, which could also be used to model the color distribution in a texture.

Solving the advection equation for all texels lets us move the texture along the flow. But there's no visible streamline in such a texture—the flow is only visible through animation. The trick is now to combine texture advection with texture blending (see Figure 5): after each advection, the advected texture is combined with the input noise by alpha blending.[9]

If we let this advection/blending run for some time for a steady flow, we'll eventually obtain a stationary LIC picture. Through alpha blending, the kernel of this LIC image has—by construction—an exponential drop-off.[6,9] In other words, texture advection produces an incremental computation of LIC with an exponential filter kernel.

The disadvantage is the coupling to the exponential filter, because it has suboptimal low-pass filter characteristics. This disadvantage is balanced by several advantages. Most importantly, texture advection works equally well for steady and unsteady flow; it's a good choice for unsteady and animated flow visualization. (A survey paper discusses other approaches to texture-based, time-dependent flow visualization.[3]) Another advantage is the incremental computation that leads to extremely quick results for previews.

3D Flow Visualization

The step from 2D to 3D is conceptually simple. We just replace the 2D textures for the input texture and the final image by respective 3D textures. And the flow field must be defined on a 3D domain instead of a 2D domain. The computation of LIC or texture advection doesn't change: it's still a convolution or blending along particle traces. Only the final rendering of the 3D "image" needs some work. We can't just display a 2D image, but have to visualize a 3D texture by volume rendering.[10]

Unfortunately, the extension to 3D comes with challenges related to occlusion and visual perception. The strength of 2D texture-based flow visualization turns into a major disadvantage in 3D: the dense representation leads to some serious occlusion problems. Naive 3D visualization, therefore, only shows the flow on the

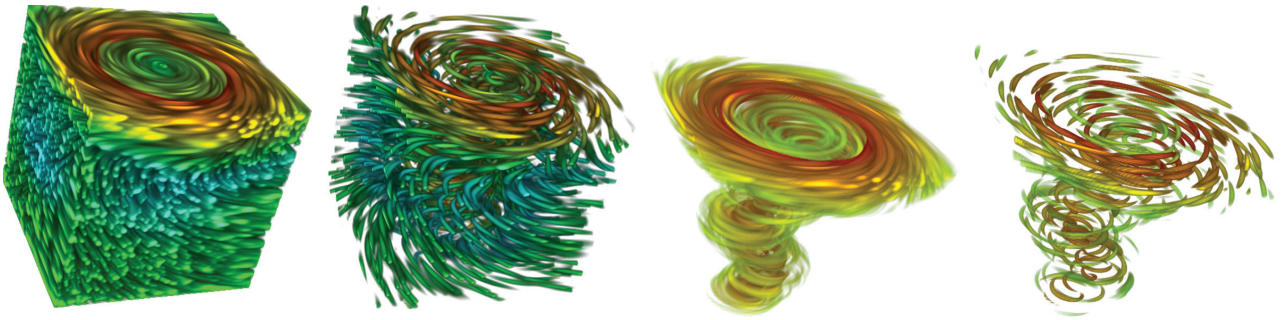

Figure 6. 3D LIC: (a) dense LIC with white noise input, (b) sparse LIC, (c) dense LIC that highlights high velocities, and (d) sparse LIC with velocity highlighting.

front-facing part of the computational domain; everything behind is hidden (see Figure 6a). To tackle this problem, we have to consider the control of seeding—now via the right choice of noise texture. Figure 6b shows that sparse noise leads to more empty space in the 3D LIC image. In this way, we can somewhat see behind the front faces of the volume boundary. However, even more promising is the selective control of the transparency in volume rendering.[11] Figures 6c and 6d show simple examples that highlight the regions of high velocity. With more advanced definitions of regions of interest or features, we can further improve the visualization, focusing on the relevant pieces of visual information. Texture-based flow visualization is independent of such feature definitions and can be—without any modification—run with any feature highlighting and noise model.

Another issue of 3D texture-based flow visualization is the high computational load, because the runtime complexity and memory requirements scale cubically (instead of quadratically for 2D visualization). Although parallel computation on graphics hardware already helps improve the visualization performance, we recommend using additional acceleration strategies. One approach exploits hierarchical computation that can reuse intermediate computational results to save on the number of computations.[12] Another approach reuses already computed information along particle trajectories with Fast LIC.[5] Finally, we can tightly integrate the LIC computation with volume rendering to avoid unnecessary computations for occluded or fully transparent parts[13]: with a lazy evaluation of the LIC computation, only relevant visual contributions to the volume rendering are actually computed. Figure 6 is based on the latter method.

The idea of reducing to the relevant information can be taken to extremes: Why not show the flow only on a curved, yet representative surface through 3D space? This eventually leads to surface LIC. A popular and highly efficient implementation of surface LIC first projects the surface and the attached flow onto the image plane, and then essentially executes 2D texture-based flow visualization on the image plane.[14] Figure 7 shows an example of surface LIC computed with an improved image-space LIC method.[15]

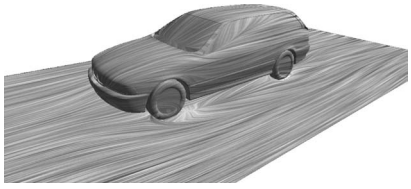

Figure 7. Surface LIC computed with an image-space method. The 3D flow field is shown only at the boundary surfaces. Surface LIC visualizes the tangential component of that flow field.

The key strength of texture-based flow visualization is its high flexibility, covering visual styles that range all the way from completely dense representations to those that resemble traditional geometric line rendering. The "look" can be easily and gradually controlled by the choice of input (noise) texture and kernel. Especially for 2D flow, the dense representation is useful because it avoids the seeding problem. The challenge of 3D visualization is the right choice of input texture, which shouldn't be too dense, and the appropriate specification of transparency. Another advantage is the simple way of including support for unsteady flow and animation.

Previously, the main roadblock in the practical use of texture-based flow visualization was its low rendering speed because of the many per-texel computations. With efficient algorithms and parallelization on GPUs, this obstacle is no longer there. Therefore, modern texture-based flow visualization is ready to be included in processes of interactive visual flow analysis. For ease of implementation, we provide a couple of examples of source codes and executables at www.vis.uni-stuttgart.de/texflowvis. This webpage also contains more background material, including links to further reading, example images, and videos. Some of the methods are also available in popular visualization frameworks like the Visualization Toolkit (VTK; www.vtk.org) or ParaView (www.paraview.org).

Acknowledgments

This work was supported by Deutsche Forschungsgemeinschaft (DFG) under grant WE 2836/4-1. The datasets used for Figures 1 and 3-5 were provided by Marco Ament, the one for Figure 6 by Roger Crawfis, and the one for Figure 7 by the BMW Group.

References

1. D. Weiskopf and G. Erlebacher, "Overview of Flow Visualization," *The Visualization Handbook*, Elsevier, 2005, pp. 261–278.
2. G.K. Karch et al., "Dye-Based Flow Visualization," *Computing in Science & Eng.*, vol. 14, no. 6, 2012, pp. 80–86.
3. R.S. Laramee et al., "The State of the Art in Flow Visualization: Dense and Texture-Based Techniques," *Computer Graphics Forum*, vol. 23, no. 2, 2004, pp. 203–221.
4. B. Cabral and L. C. Leedom, "Imaging Vector Fields Using Line Integral Convolution," *Proc. ACM Siggraph*, ACM, 1993, pp. 263–270.
5. D. Stalling and H.-C. Hege, "Fast and Resolution Independent Line Integral Convolution," *Proc. ACM Siggraph*, ACM, 1995, pp. 249–256.
6. D. Weiskopf, "Iterative Twofold Line Integral Convolution for Texture-Based Vector Field Visualization," *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, Springer, 2009, pp. 191–211.
7. R. Wegenkittl, E. Gröller, and W. Purgathofer, "Animating Flow Fields: Rendering of Oriented Line Integral Convolution," *Proc. Computer Animation*, IEEE, 1997, pp. 15–21.
8. B. Jobard, G. Erlebacher, and M.Y. Hussaini, "Lagrangian-Eulerian Advection of Noise and Dye Textures for Unsteady Flow Visualization," *IEEE Trans. Visualization and Computer Graphics*, vol. 8, no. 3, 2002, pp. 211–222.
9. J.J. van Wijk, "Image-Based Flow Visualization," *ACM Trans. Graphics*, vol. 21, no. 3, 2002, pp. 745–754.
10. S.P. Callahan et al., "Direct Volume Rendering: A 3D Plotting Technique for Scientific Data," *Computing in Science & Eng.*, vol. 10, no. 1, 2008, pp. 88–92.
11. D. Weiskopf, T. Schafhitzel, and T. Ertl, "Texture-Based Visualization of Unsteady 3D Flow by Real-Time Advection and Volumetric Illumination," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 3, 2007, pp. 569–582.
12. M. Hlawatsch, F. Sadlo, and D. Weiskopf, "Hierarchical Line Integration," *IEEE Trans. Visualization and Computer Graphics*, vol. 17, no. 8, 2011, pp. 1148–1163.
13. M. Falk and D. Weiskopf, "Output-Sensitive 3D Line Integral Convolution," *IEEE Trans. Visualization and Computer Graphics*, vol. 14, no. 4, 2008, pp. 820–834.
14. R.S. Laramee et al., "ISA and IBFVS: Image Space-Based Visualization of Flow on Surfaces," *IEEE Trans. Visualization and Computer Graphics*, vol. 10, no. 6, 2004, pp. 637–648.
15. D. Weiskopf and T. Ertl, "A Hybrid Physical/Device-Space Approach for Spatio-Temporally Coherent Interactive Texture Advection on Curved Surfaces," *Proc. Graphics Interface*, Canadian Human-Computer Comm. Soc., 2004, pp. 263–270.

Rudolf Netzel is a research assistant and doctoral candidate at the University of Stuttgart, Germany. His research interests include scientific visualization and visual analytics. Netzel has a Diplom (MSc) in computer science from the University of Stuttgart, Germany. Contact him at rudolf.netzel@visus.uni-stuttgart.de.

Daniel Weiskopf is a professor at the University of Stuttgart, Germany. His research interests include visualization, visual analytics, GPU methods, computer graphics, and special and general relativity. Weiskopf has a PhD in physics from the University of Tübingen, Germany. He's a member of the IEEE Computer Society, ACM, Eurographics, and the Gesellschaft für Informatik. Contact him at weiskopf@visus.uni-stuttgart.de.

cn *Selected articles and columns from IEEE Computer Society publications are also available for free at http://ComputingNow.computer.org.*