Spectrum-preserving texture advection for animated fluids

Qizhi Yu, Fabrice Neyret, Eric Bruneton, Nicolas Holzschuch

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Spectrum-preserving texture advection for animated fluids

Qizhi Yu — Fabrice Neyret — Eric Bruneton — Nicolas Holzschuch

N° 6810

Janvier 2009

Thème COG

ISSN 0249-6399 ISRN INRIA/RR--6810--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Spectrum-preserving texture advection for animated fluids

Qizhi Yu *, Fabrice Neyret * †, Eric Bruneton* , Nicolas Holzschuch *

Thème COG — Systèmes cognitifs
Équipes-Projets Evasion

Rapport de recherche n° 6810 — Janvier 2009 — 17 pages

Abstract: Texturing an animated fluid is a useful way to augment the visual complexity of pictures without increasing the simulation time. But texturing flowing fluids is a complex issue, as it creates conflicting requirements: we want to keep the texture properties (features, spectrum) while conforming to the underlying flow — which distorts the attached texture. In this paper, we present a new method for texturing animated fluids. Our method ensures that the moving texture always follows the velocity field of the fluid, while maintaining key properties of the original texture. Our algorithm runs in real-time; our experiments show that it is well suited for a wide range of input texture, including, but not limited to, noise textures.

Key-words: Texturing animation, Animated fluids, Spryticles, Particles, Lagrangian methods

* Laboratoire Jean Kuntzmann et INRIA Grenoble
† CNRS

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Advection de texture pour fluides animés, préservant le spectre

Résumé : Texturer un fluide animé est une faon pratique d'augmenter la complexité visuelle des images sans augmenter le temps de simulation. Mais texturer des fluides animés est un problme complexe à cause de buts condradictoires: on souhaite conserver les propriétés de la texture (éléments caractéristiques, spectre) tout en respectant le mouvement du fluide – qui déforme la texture. Dans cet article nous présentons une nouvelle méthode pour texturer des fluides animés. Notre méthode fournit une texture animée qui suit à chaque instant le champ de vitesse du fluide, tout en prservant les propriétés principales de la texture originelle. Notre algorithme fonctionne en temps-réel; nos expriences montrent qu'il est bien adapt pour de nombreux types de textures d'entrée, y compris les textures de bruit.

Mots-clés : Animation de texture, Fluides animés, Spryticles, Particules, Méthodes Langrangiennes

1 Introduction

Animated fluids are frequently used in Computer Graphics, whether in virtual worlds, special effects or video games. As it is difficult to model the complete behavior of the fluid, animators and designers resort to texture mapping for finer surface details, such as foam, normal mapping and smaller waves. But mapping a texture on a flowing fluid, such as a river, creates conflicting requirements. On one hand, we want the texture to follow the flow exactly, so that the fluid movements are clearly visible. On the other hand, we want the texture to keep its original properties. As the fluid movements introduce large and cumulating distortions, shearing and stretching the original texture, solving both requirements is a difficult task.

In this paper, we present a new technique for the advection of textures on a flowing fluid. Our technique takes as input a flowing fluid, whose velocity field is known, and a texture (procedural or image). We produce as output an animated texture whose features follow exactly the velocity field, while keeping several key properties of the input texture, including its local appearance.

Our algorithm works as follows: we start by placing sample particles along the flow. These particles are advected by the flow. A grid is attached to each particle, and this grid is also advected and deformed by the flow. Each grid is mapped to a fixed area of the input texture. To maintain texture properties, particles are eliminated when the distortion of their grid becomes too large. We maintain a constant particle density over the flow, killing or generating new particles when needed. In a final step, we reconstruct the texture by blending together these textured grids. The method is simple enough that it runs in real-time on standard GPUs.

Obviously, our algorithm does not apply to all possible input textures. It requires that we can blend together different areas of the input texture and yet create a satisfying result. We expect our algorithm to perform poorly on highly structured textures; however, we found that it works well with a large range of input textures, including noise textures, foam, bubbles... These textures correspond to the kind of features we expect to see on an animated fluid.

To measure the quality of animated textures, we suggest two criteria: the Fourier spectrum and the optical flow; both are computed on the output of our algorithm. Our experiments show that the optical flow of the animated texture matches exactly the input velocity field, while keeping the Fourier spectrum of the input texture.

Our paper is organized as follows: in the next section, we review previous work on detail advection methods for animated fluids. We then present our algorithm (Section 3). In Section 4, we present our results and compare them to existing work. Finally, in Section 5, we conclude and present avenues for future work.

2 Previous work

Particle systems proposed by [14, 15] is an efficient way to add details to scenes and animation (fire, explosions, vegetation...). Since then, particles have been generalized in animation, e.g. [16, 18]. Moving particles with attached sprites, or *spryticles*, are now ubiquitous in Computer Graphics applications such as games and special effects [23, 3].

Stam and Fiume [21] render "warped blobs" to account for detailed turbulence effects in a moving fluid: blob particles are carried by the flow, like particles. At rendering time, rays intersecting a blob are back-projected in time to the initial density distribution to be marched. More recently, Narain *et al.* [11] place blocks of velocity

noise attached to particles in a flow, to create turbulence effects. Yu *et al.* [24] add texture sprites attached to particles in a moving fluid, to create textured rivers. Our algorithm shares many key points with these works; the main difference is that they are advecting rigid particles, usually spheres or disks, while we are advecting deformable grids. This enables us to reconstruct a smooth homogeneous movement after blending particles, without secondary motion or sliding effects.

The texture advection idea was first introduced by Max *et al.* [9, 8]. Since then, this technique has been used for visualization [22] and for 2D or 3D fluid animation [20]. The most recent work on this topic was [12]. This approach has been used for special effects in motion pictures, and our algorithm has been largely inspired from these papers. The main difference is that these papers rely on an Eulerian formalism, while we rely on a Lagrangian formalism. The Eulerian approach means that there must be a parameterization for the whole domain, making local adaptation difficult and wasting calculation and storage for empty areas.

Our algorithm can be seen as a combination of these two approaches, particles (Lagrangian) and texture advection (Eulerian), giving us the benefits of both.

Animated texture synthesis methods [6, 5, 4] have the same input (a texture and an animated flow) and output (an animated texture) as our algorithm. The main difference with our work is the nature of the input textures, and the features we chose to conserve. These algorithms use neighbor-based similarity criterion in example images, and loosely conform to the input flow. We focus on minimizing local distortions and we enforce a strict conformance to the flow, but our current scheme doesn't establish a relationship between the content of neighboring sprites. We believe that both approaches have their benefits, depending on the target application: ours is well adapted to noise textures (and procedural textures using them as input), and to images with weak or local structures. Furthermore, the simplicity and locality of our approach allow us to have a real-time implementation, with no pre-computations, compared to several minutes per frame for [4], making it a better choice for interactive applications.

3 Our algorithm

Our algorithm takes as input an animated velocity field and an image or procedural texture. It generates as output an animated texture by blending together as small set of *deformable textured grids* advected with the input flow. We start with a random Poisson disk distribution of *particles* (see Figure 1). We then create grids centered on these particles. Initially the grids are regular, have the same size, and are associated with a random area of the input texture. Then, at each time step:

- We advect the grid vertices with the flow, and we set the new position of each particle to the centroid of its advected grid.
- We maintain a uniform distribution of particles by killing and creating particles when necessary. We also kill particles whose grid is too distorted (see Section 3.1). We create regular grids for the new particles, using random areas of the input texture.
- We compute spatial and temporal blending weights for the grids. The goal is to avoid seams and popping in the animated texture when particles are killed and created (see Section 3.2). In particular grids are still advected and blended after their particle's death while they fade out.

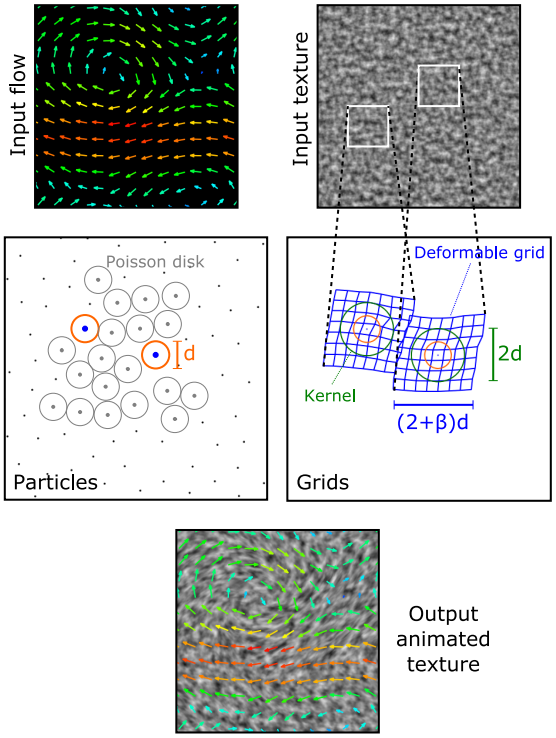


Figure 1: Overview of our algorithm. Top*: we take as input an animated velocity field and a texture.* Middle*: we advect particles by the flow. We maintain a Poisson disk distribution, killing or creating new particles when necessary. A deformable grid is attached to each particle and mapped to a fixed area of the input texture.* Bottom: *we blend these deformable textured grids to get our animated texture.*

- We render the animated texture either by directly drawing and blending the textured grids, or by using an indirection map to recover the grids covering a given pixel (see Section 3.3).

3.1 Particle sampling and distortion

The deformable textured grids must cover the whole fluid to generate an animated texture without holes. However they must not overlap too much and must not be too distorted in order to preserve the input texture properties in the generated texture. We ensure this by enforcing a dynamic Poisson disk distribution of the grid centroids (i.e., of the particles) and by killing particles whose grid is too distorted.

Particle distribution We maintain a Poisson disk distribution by using the algorithm [2], which is well adapted to dynamic updating [24]. This algorithm kills particles when their minimal distance to the others is less than d. It then creates new particles at a minimal distance d to the remaining particles. If each grid covers at least a *kernel* of diameter $2d$ around its particle (see Figure 1), then this algorithm guarantees a coverage of the plane without holes.

We must avoid killing particles too often (especially young particles) since their grids are still advected and blended until they fade out. This would result in many overlapping grids, which is costly and gives a blurry animated texture. To solve this we introduce an hysteresis: we kill particles when their minimal distance to the others is less than $(1-\alpha)d$. However, α should be small to avoid increasing the particle density. In our implementation we used $\alpha = 0.25$.

Grid distortion The grids must be larger than the kernels in order to avoid holes. To ensure this we delete a particle when this criterion is not met. Hence a large initial grid size gives a long particle lifetime. However grids must not be too large to avoid increasing the number of vertices per grid. Thus we use an initial size of $(2+\beta)d \times (2+\beta)d$, with a small β. In our implementation, we used $\beta = 0.6$. We also delete a particle if its grid folds over or if its distortion becomes too large (see Section 3.2). As in [19], we evaluate the grid distortion as the distortion δ of the most distorted triangle. We compute the distortion of each triangle by using the singular values $\gamma_{min}, \gamma_{max}$ of the Jacobian J of the affine transformation between the initial and advected triangle [17]. The singular values are the square root of the eigenvalues of $J^t J$. They give the minimum and maximum length that a unit vector can get after this transformation. Precisely, we use $\delta = max(\gamma_{max}, 1/\gamma_{min})$ (1 means no distortion). From this we define a *quality* measure varying between 1 (perfect) and 0 (unacceptable), $Q(\delta) = \max(\frac{\delta_{max}-\delta}{\delta_{max}-1}, 0)$, where δ_{max} is the maximum acceptable distortion. δ_{max} is the main free parameter of our algorithm since it directly impacts the global distortion. In our examples we used $\delta_{max} = 5$ (this does not mean that we can see a 5 folds distortion: the fading hides it long before – see Section 3.2).

Flow with boundaries It is important to deal with boundaries for the flow. Eulerian approaches such as [9, 12] do not address this issue. When a grid attached to a particle covers a boundary, we advect the grid vertices outside the flow by extrapolating the velocity field out of the flow with a push-pull algorithm [17] in each grid (see Figure 2). We ignore these vertices during the computation of the grid distortion.

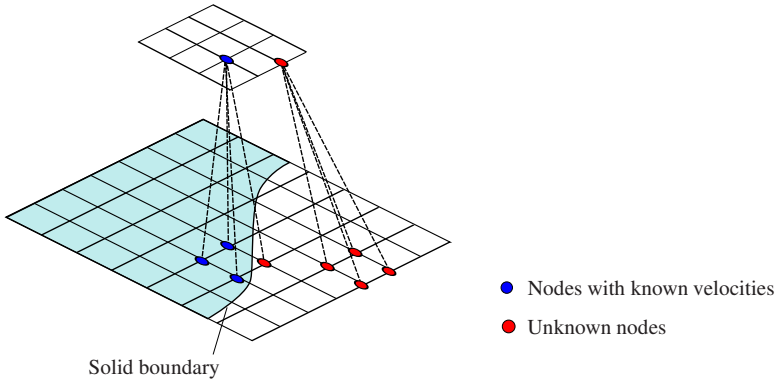


Figure 2: *Pyramid grids for extrapolating the velocities of the outside nodes in a patch overlapping boundaries. We repeatedly build coarser grids by averaging known velocities of every 2×2 nodes until we reach a grid with no unknown node. Then, we go down in the hierarchy from the coarsest grid, filling unknown nodes in a grid with the values from the neighboring coarser grid.*

3.2 Blending and continuity

We ensure the spatial and temporal continuity of the animated texture by blending the deformable grids using weighting functions, as in other blob based representations like RBF [1] or SPH [10]. We combine weights associated with the grid's geometry (kernel and distortion weights) and with the grid's lifetime.

Spatial weights The kernel is associated with a spatial weight $K_p(x)$ going from 1 at $\mathbf{p}$ to 0 at distance d of the particle $\mathbf{p}$. But this is not sufficient to avoid discontinuities. We kill particles when their grid no longer covers the kernel but since grids are still advected and blended until they fade out, holes can appear, showing the grid border. We need to continuously fade out at grid borders inside the particle's kernel. For this we use another weight $K_g(\mathbf{v})$ defined at grid vertices and interpolated linearly inside the grid triangles. This weight must go to 0 at the grid border. In our implementation we used $K_p(\mathbf{x}) = \max(1 - \|\mathbf{x}-\mathbf{p}\|/d, 0)$, and $K_g = 1$ for inner vertices 0 for border vertices.

Distortion weights In order to limit the apparant distortion we use a weight K_d based on our quality measure Q (see Section 3.1). Instead of using a global weight per grid we use a local weight to get a finer control of the appearant distortion (it is frequent to have a small distorted area in a grid). So we define K_d on grid vertices (using the average quality of the adjacent triangles) and we interpolate it linearly inside the grid triangles. We want grids to fade out when K_d gets to 0. But since a grid continues to deform after its particle has been killed, we must kill the particle before that. In our implementation we kill a particle when there is a vertex inside the kernel with $K_d(\mathbf{v}) < 0.5$.

Temporal weights Finally we fade particles in and out at their creation and destruction, using two weights $F_{in}(t)$ and $F_{out}(t)$. The fading period τ can be long (in our implementation we used $\tau = 5$ seconds). F_{out} is mainly used to force the fading out of grids whose distortion would stop increasing. And if τ is longer than the particles lifetime, F_{in} and F_{out} rule the weights of all particles and are thus renormalized at the end (see below).

The total weight at a point $\mathbf{x}$ inside a grid is finally defined as

$$w(\mathbf{x},t) = K_p(\mathbf{x})K_g(\mathbf{x})K_d(\mathbf{x})F_{in}(t)F_{out}(t). \tag{1}$$

The animated texture at $\mathbf{x}$ is then computed by blending the textures $T_i(\mathbf{x})$ of the grids covering this location with (see Section 3.3):

$$T(\mathbf{x}) = \frac{\sum w_i T_i}{\sum w_i}. \tag{2}$$

To compensate the loss of contrast due to overlapping, we can also use:

$$T(\mathbf{x}) = \frac{\sum w_i T_i}{\sum {w_i}^2}. \tag{3}$$

3.3 Reconstruction and rendering

We propose two methods to blend the deformable textured grids:

- We can draw each grid one by one, accumulating the $w_i T_i$ and w_i in separate channels. A second pass combines these channels to evaluate Equation 3.
- Alternatively, we can also use an indirection structure as in [7] and [24]. In a first pass we draw for each grid its (u,v) field and its weights $w(\mathbf{x},t)$ in an *FFD image*. We also divide the fluid domain in tiles and compute for each tile the list of grids that intersect it. In a second pass a pixel shader uses these indirection maps to find the grids that may cover a given pixel, and then to get the (u,v) and $w(\mathbf{x},t)$ of each grid at this pixel. It then evaluates Equation 3.

The second method is more complex but is better adapted to the case of a sparse fluid in a large domain, such as a river. It computes only the visible pixels, and its memory usage is proportional to the number of grids, as opposed to the size of the domain. On the contrary, the first method requires textures covering the whole domain at the maximum resolution, using floats (since the accumulated values are not bounded). It thus computes all pixels, even invisible ones.

In both methods the input texture can contain either final colors or input parameters for a complex procedural shader (clouds, fire, etc). In the second case we blend the parameters *before* applying the procedural shader to avoid ghosting effects [12]. This also decreases the computation cost as the procedural shader is called only once per pixel instead of once per grid. Note that the procedural shader can use displacement mapping to create a 3D surface from the 2D animated texture (see Figure 8).

4 Results and comparison

4.1 Performance and timings

One of the strongest advantages of our method is that it runs in real-time, making it useful for video-games, exploration of virtual world, just-in-time generation of content and virtual modeling.

We tested our algorithm on an Athlon AMD 3000+ at 1.8 MHz with an NVidia GeForce 8800 GTS and a picture resolution of 1024×1024. With these settings and with roughly 150 particles, each attached to a grid of 8×8 vertices, the average computation time for each frame was 8.1 ms (123 fps), of which 5.1 ms was for the advection of vertices and 2.5 ms for final reconstruction (using the indirection-based method). The cost for Poisson disk sampling was negligible. This leaves more than 20 ms of computation time for other tasks (such as animation and rendering of the virtual world) in a real-time application (30 fps or more).

The rendering time is proportional to the overall number of vertices: doubling the number of particles or doubling the number of vertices in the grid attached to each particle will both have the effect of doubling the computation time.

4.2 Quality of the animated texture

We introduce two criteria to evaluate the quality of an animated texture: its optical flow and its Fourier spectrum. Both properties are computed on the generated animated texture. Ideally, the optical flow of the synthesized animated texture should match the input fluid velocity field, while its Fourier spectrum should match the Fourier spectrum

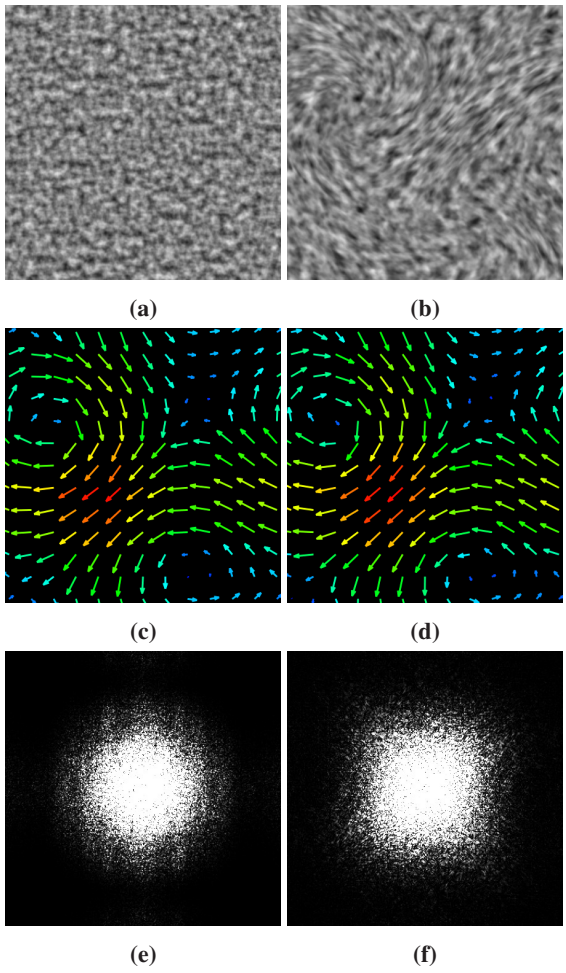


Figure 3: Quality of our generated animated textures. Left*: input texture (a), input flow (c) and Fourier spectrum (e) of input texture.* Right*: our advected texture (b), its optical flow (d) and its Fourier spectrum (f).*

of the input texture. As can be seen on Figure 3 and the accompanying video [1], our algorithm works perfectly on both points.

4.3 Comparison with Eulerian texture advection

Eulerian Texture advection [12] is very close to our work. The main difference is that they work in an Eulerian framework, while we work in a Lagrangian framework. Parameters in an Eulerian framework apply to the entire world, which complicates the adaptation to local effects. Their solution to regenerate the texture when the local accumulated distortion is too high is to interpolate between 3 texture layers that regenerate periodically with 3 different *predefined* latencies. They choose the interpolation

[1] Available at `http://http://evasion.inrialpes.fr/Membres/Qizhi.Yu/projects/texadv/`

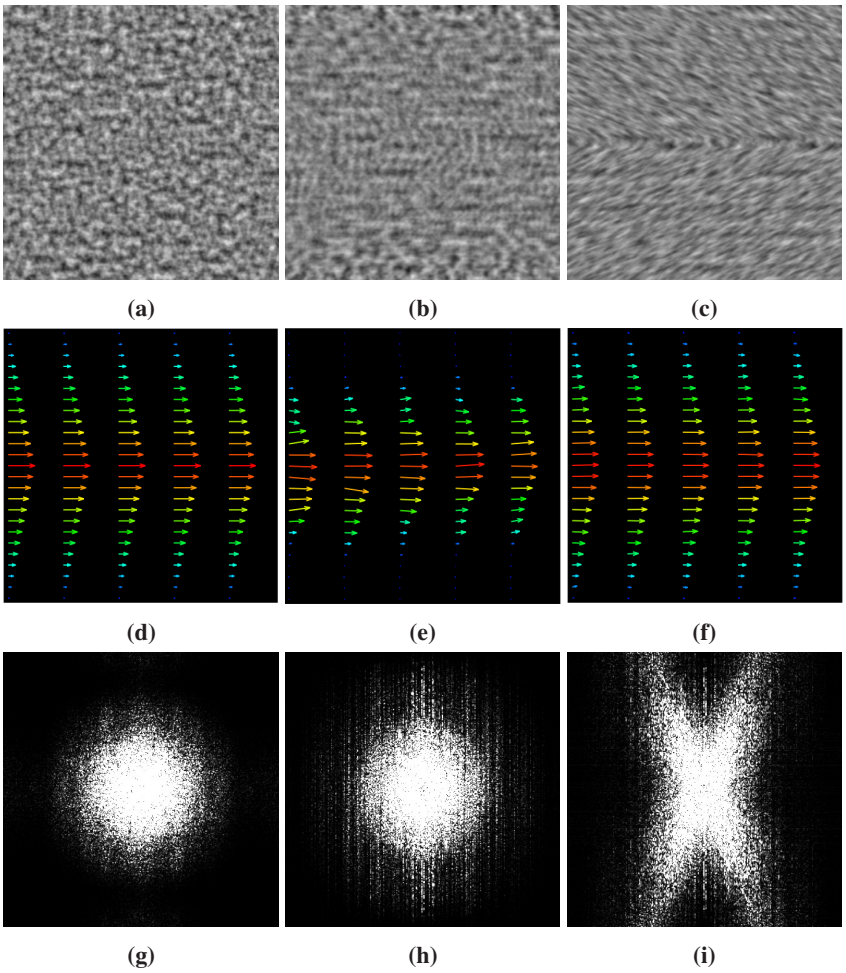


Figure 4: *Problems of Eulerian texture advection.* Left*: input texture (a), input flow (d) and Fourier spectrum of input texture (g).* Middle*: with a short regeneration latency the advected texture (b) conveys an incorrect optical flow (e), but the Fourier spectrum is almost preserved (h).* Right*: with a long latency the texture is too stretched (c) and the Fourier spectrum is distorted (i), but the optical flow matches the input velocity field (f).*

weights to get a latency inversely proportional to the local increase rate of the distortion. But the predefined latencies are bounded, while the distortion increase rate is not (areas at rest require an infinite latency). Hence for some flows they are only able to maintain either the optical flow or the Fourier spectrum of the texture, at the expense of the other (see Figure 4 and the accompanying video).

4.4 Comparison with sprite-based texture advection

The approach of [24] to simulate rivers has similarities with our work. The main difference is that they advect solid particles, while we advect deformable grids. This gives

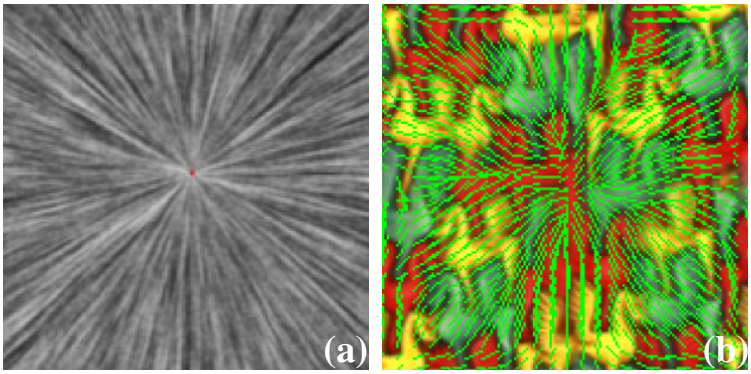


Figure 5: Comparison with animated texture synthesis. *The optical flow (b) does not accurately match the input velocity field (a) (measured here on [Kawtra et al. 2005]).*

a "blocky" velocity field, and thus unwanted secondary motions. It can also give a relative sliding motion between blended features on overlapping sprites, which can be noticeable in stretched areas (see the accompanying video). Our deformable grids are a natural improvement for such an application.

4.5 Comparison with animated texture synthesis

Animated texture synthesis algorithms [6, 5, 4] take the same input and produce the same output as our work. There are two main differences. They measure texture similarities using neighborhoods while we use the Fourier spectrum and they put less emphasis on the accurate reproduction of the input velocity field. Our experiments show that these methods tend to give rigid moving chunks around structured features and show sudden changes in the pattern. In other words the resulting optical flow does not accurately match the input flow (see Figure 5 and the accompanying video).

Another point is that since texture synthesis algorithms work by identifying neighborhoods (and thus structures) in the input textures, they tend to give unreliable results for textures without identifying features, such as noise textures. See Figure 6 for an application of the texture synthesis algorithm using a noise texture as input: the texture synthesis introduced new features that were not present in the input texture.

4.6 Discussion

Due to its properties our algorithm works well with noise textures and procedural textures which are useful for modeling various fluids in the nature such as fires, clouds and water waves (see Figure 8, Figure 7, and the accompanying video).

Our experiments show that it also applies to a large range of input textures, including bubbles, foam and froth. Our algorithm places a few requirements on the input texture in order to work correctly: the features must blend nicely by addition. In particular, this supposes that there are no significant large scale structures, and that local structures are resistant to blending. For example, our algorithm works well with pictures of bubbles because blending together two pictures of a bubble produces a con-

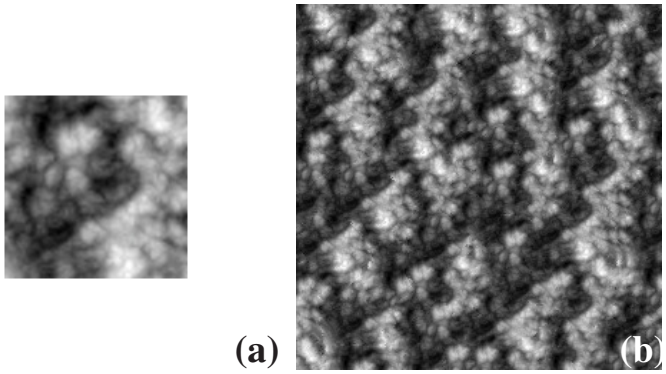


Figure 6: Comparison with animated texture synthesis. *Example based texture synthesis introduces artificial features (b) with noise input textures (a) (measured here on [Lefebvre and Hoppe 2005]).*

vincing bubble (or two bubbles glued together). It would not work, however, with *e.g.* a checkerboard texture.

We think that the set of textures that work nicely with our algorithm (foam, bubbles, froth, debris...) are precisely the kind of textures we would like to use on a moving fluid, introducing moving details that enhance the realism of the fluid.

Animated texture synthesis algorithms [6, 5, 4] preserve large-scale features of the input texture but loose other texture properties, do not conform accurately to the input flow, and require a long pre-computation and several minutes per frame. We think that both algorithms have their benefits, depending on the requirements and the input textures.

5 Conclusion and future work

We have presented an algorithm for the generation of animated textures suitable for texturing moving fluids. Our algorithm takes as input a texture and the velocity field of a moving fluid, and generates an animated texture that accurately follows the velocity field, while preserving the properties of the original texture. Our method is well suited for noise textures, as well as procedural textures based on noise, and it also works on a large variety of input textures, and a large variety of moving fluids. As our algorithm accurately follows the velocity field of the moving fluid, we believe it will have many applications in Computer Graphics, including special effects for motion pictures, simulators, video games and virtual worlds. The fact that our algorithm runs in real-time on a standard GPU makes it well suited for interactive applications.

Our algorithm could be applied directly to 3D velocity fields and 3D input textures, except for the rendering part. As future work we would like to use volumetric rendering to experiment with this. We would also like to extend our method to use a Poisson disk sampling in screen space as in [24], to get a view dependent LOD mechanism. Finally we could try to replace the random selection of the input texture area for new grids with a smarter method.

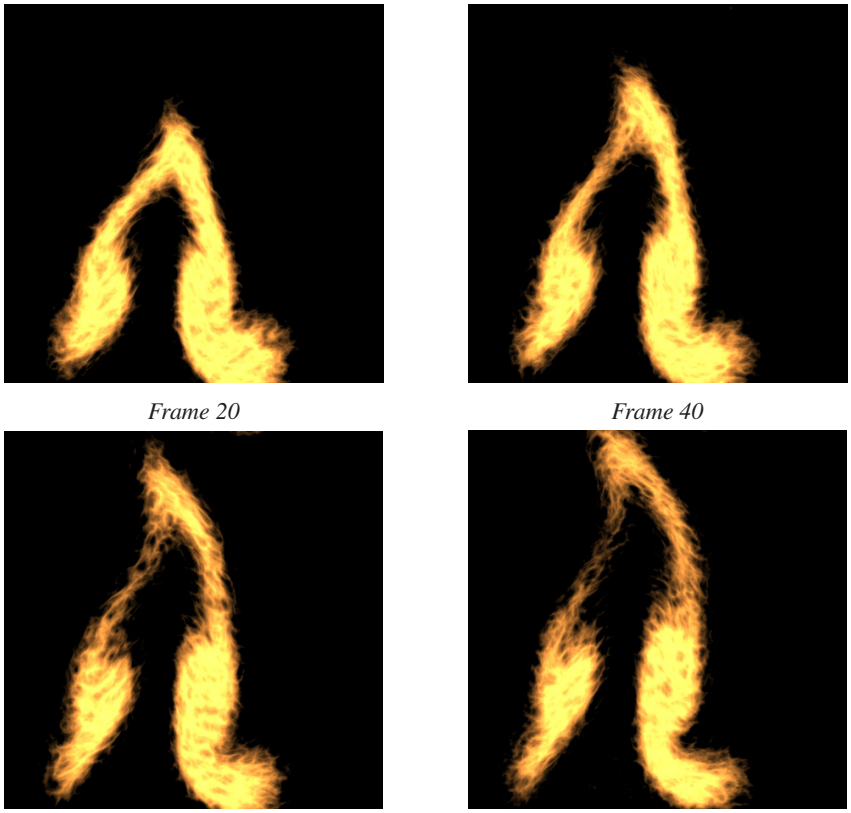


Figure 7: *Animated fires with abundant details. It is generated in several steps. A 2D density field is advected with a low resolution velocity field. Then, flow noise [13] advected with our method is used to modulate the density field. Finally, a fire shader uses the enriched density field to generate colors.*

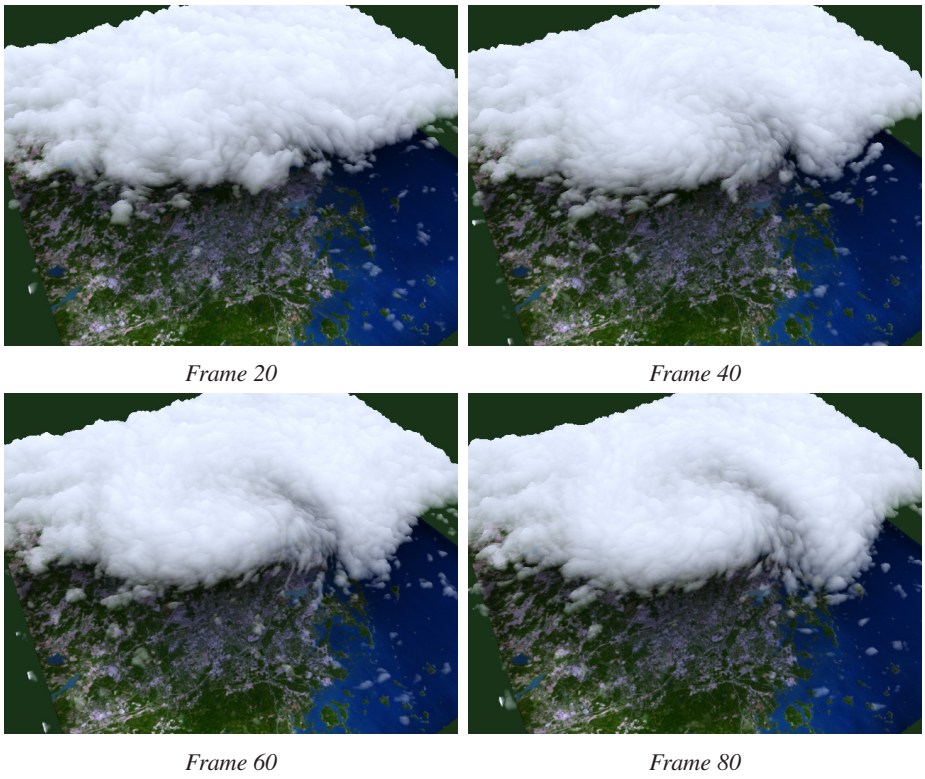
References

[1] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 67–76. ACM Press / ACM SIGGRAPH, 2001.

[2] Daniel Dunbar and Greg Humphreys. A spatial data structure for fast poisson-disk sample generation. *ACM Trans. Graph.*, 25(3):503–508, 2006.

[3] Doug Ikeler and Jennifer Cohen. The use of Spryticle in the visual FX for "The Road to El Dorado". In *ACM SIGGRAPH 2000 sketches*, 2000.

[4] Vivek Kwatra, David Adalsteinsson, Theodore Kim, Nipun Kwatra, Mark Carlson, and Ming Lin. Texturing fluids. *IEEE Transactions on Visualization and Computer Graphics*, September 2007.



Frame 20 *Frame 40*

Frame 60 *Frame 80*

Figure 8: *Animated clouds with advected details. It is generated in a number of steps. A 2D density field is advected with a low resolution velocity field. Advected flow noise is used to modulate the density field. Finally, the enriched density field is used for displacement mapping in a cloud shader.*

[5] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Transactions on Graphics, SIGGRAPH 2005*, August 2005.

[6] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 777–786, New York, NY, USA, 2005. ACM.

[7] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)*. ACM, ACM Press, 2003.

[8] Nelson Max and Barry Becker. Flow visualization using moving textures. In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data*, pages 77–87, 1995.

[9] Nelson Max, Roger Crawfis, and Dean Williams. Visualizing wind velocities by advecting cloud textures. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 179–184, 1992.

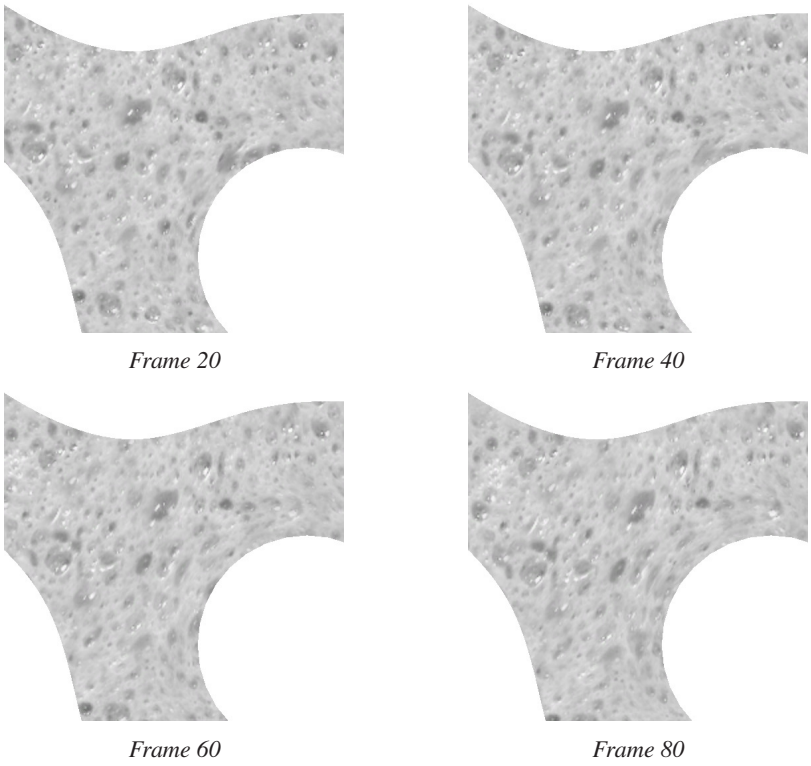


Figure 9: *Advection of a foam texture (image) in a river channel.*

[10] J. J. Monaghan. An introduction to SPH. *Computer Physics Communications*, 48:89–96, January 1988.

[11] Rahul Narain, Jason Sewall, Mark Carlson, and Ming Lin. Fast animation of turbulence using energy transport and procedural synthesis. *ACM Transactions on Graphics*, 27(5), December 2008.

[12] Fabrice Neyret. Advected textures. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 147–153, 2003.

[13] Ken Perlin and Fabrice Neyret. Flow noise: textural synthesis of animated flow using enhanced Perlin noise. In *SIGGRAPH 2001 Technical Sketches and Applications*, August 2001. http://www-imagis.imag.fr/Publications/2001/PN01.

[14] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108, April 1983.

[15] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 313–322, July 1985.

[16] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics (SIGGRAPH '87)*, 21(4):25–34, 1987.

[17] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *SIGGRAPH '01*, pages 409–416, 2001.

[18] Karl Sims. Particle animation and rendering using data parallel computation. *Computer Graphics*, 24(4), 1990.

[19] Olga Sorkine, Daniel Cohen-Or, Rony Goldenthal, and Dani Lischinski. Bounded-distortion piecewise mesh parameterization. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 355–362, 2002.

[20] Jos Stam. Stable fluids. In *SIGGRAPH '99*, pages 121–128, 1999.

[21] Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *SIGGRAPH'95*, pages 129–136, 1995.

[22] Jarke J. van Wijk. Image based flow visualization. *ACM Trans. Graph.*, 21(3):745–754, 2002.

[23] Wonder touch. `http://www.wondertouch.com/`.

[24] Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. Scalable real-time animation of rivers. *Computer Graphics Forum (Proceedings of Eurographics 2009)*, 28(2), mar 2009.

Contents

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399