# Lagrangian Texture Advection: Preserving both Spectrum and Velocity Field

Qizhi Yu, Fabrice Neyret, Eric Bruneton, Nicolas Holzschuch

# Lagrangian Texture Advection: Preserving both Spectrum and Velocity Field

Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch

**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 key texture properties (features, spectrum) while advecting the texture with the underlying flow — which distorts it. In this paper, we present a new, Lagrangian, method for advecting textures: the advected texture is computed only locally and follows the velocity field at each pixel. The texture retains its local properties, including its Fourier spectrum, even though it is accurately advected. Due to its Lagrangian nature, our algorithm can perform on very large, potentially infinite scenes in real time. Our experiments show that it is well suited for a wide range of input textures, including, but not limited to, noise textures.

**Index Terms**—Computer Graphics, Texture, Animation, Particles, Lagrangian methods

## 1 Introduction

Animated fluids are frequently used in Computer Graphics applications, whether in virtual worlds, special effects or video games. As it is difficult to model the complete behavior of a fluid, animators and designers resort to texture mapping for finer surface details, whether small displacements, variations of the normals, or foam and debris being transported. But applying 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 stay realistic; yet on the other hand, we want the texture to keep its original properties[1]. As the fluid movements introduce large and cumulative distortions, shearing and stretching the original texture, solving both requirements is a difficult task.

In this paper, we present a new, Lagrangian, technique for the advection of textures. Our technique takes as input a flowing fluid, whose velocity field is known, and a texture (either 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 (see Fig. 1).

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 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. Due to its Lagrangian nature, the complexity of our algorithm only depends on the pixels that are actually generated. Thus it works on very large scenes, potentially unbounded, in real-time.

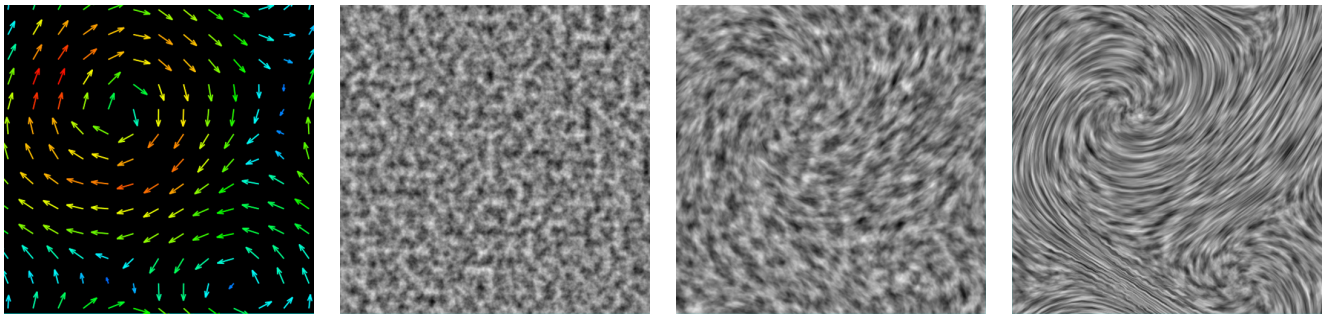
Obviously, our algorithm does not apply to all possible input textures. It requires that we can blend together different areas of the input texture and yet create a satisfying result. We rely on a "smart blending" approach for procedural textures, but we expect our algorithm to perform poorly on images with highly structured content; however, we found that it works well with a large range of input textures (see Fig. 4, 5, 6 and 11, as well as the accompanying video), including noise textures, foam, ripples, lava... Interestingly, these textures correspond to the kind of features we most want to apply on realistic animated fluids.

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.

- *Université de Grenoble and CNRS, Laboratoire Jean Kuntzmann, BP 53, 38041 Grenoble Cedex 9, France*
- *INRIA Grenoble Rhône-Alpes, Montbonnot, 38334 Saint Ismier Cedex, France*

1. Note that in the case of scientific visualization or for some dedicated effects, stretching can be desireable in order to convey information on the flow field, even huge stretching in the case of Line Integral Convolution. Here we address the opposite case of mostly reality-inspired imagery where the pattern mimics a fast regeneration process (ripples, foam, small-scale cloud convection) or the transportation of unstretchable details (bubbles, gravel).



(a) Velocity field (b) Input texture (c) Our algorithm (d) Naïve algorithm

Fig. 1. Our algorithm takes as input a velocity field *(a)* and a texture, here a Perlin noise texture *(b)*, and produces a texture that follows the velocity field while retaining the local properties of the input texture *(c)*. Simply advecting the original texture with the flow distorts the texture, introducing artefacts *(d)*. See also the accompanying video. In all our figures depicting a velocity field, the colors of the arrows represent speed, based on hue (from blue (slow) to red (fast)).

## 2 Previous Work

Particle systems were introduced early [1], [2] as a way to add fine details for modelling and animation, *e.g.,* for explosions, fires or plants. Since then, particles systems have been used widely in animation (see, *e.g.,* [3], [4]). Today, moving particles with attached sprites, or *spryticles*, are ubiquitous in Computer Graphics applications such as games and special effects [5], [6]. Designers and programmers like them because they provide enhanced realism with local control and output-dependant complexity.

Several papers use moving particles to enhance realism in moving fluids: *e.g.,* warped blobs for turbulence details [7] or velocity noise for turbulence [8]. A separate direction of research [9] used texture sprites attached to particles in a moving fluid, to create realistic textured rivers. All these papers share similarities with our work in the sense that they add Lagrangian advected details patches to an Eulerian fluid simulation. The fundamental difference is that they advect rigid particles, such as discs or spheres, while we advect *deformable* particles. Allowing the particles to be deformed while they are advected by the flow allows for continuity of movement between overlapping particles, removing one key limitation of particle advection.

Previous texture advection methods [10], [11], [12] used an Eulerian approach where a texture is advected and deformed by a velocity field. It has been used in visualization [13], for fluid animation [14], and for special effects in motion pictures. Our aim has been largely inspired from these: we take the same input (a velocity field and a texture) and produce the same output (a texture advected by the velocity field). The underlying approach, however, is radically different: these papers rely on an Eulerian formalism, while we use a Lagrangian approach. The Lagrangian approach allows for local adaptation to finer details, and restricting computations to the areas where the flow actually is.

Flow-guided texture resynthesis methods [15], [16], [17], [18] also share the same input (a texture and an animated flow) and output (an animated texture) as our algorithm. The main difference is that they use a global energy minimization using neighbor-based similarity criteria in the input textures, while we focus on minimizing local distortions. As a consequence, these methods keep the large-scale features, at the expense of conformance to the velocity field, while we keep local features and enforce conformance to the velocity field, at the expense of large-scale features. Our algorithm is better adapted to noise-based textures and to images with only local structure. Our local approach also requires less computations than the global minimization. As a consequence, our algorithm runs in real-time (less than 30 ms per frame) with no pre-computations, compared to several minutes per frame for, *e.g.,* [15].

Our algorithm also builds on the following previous work: our generic formalization of an advected texture input (see section 3.2) is inspired from the "smart blending" approach of [12] to prevent ghosting artifacts, and our indirect reconstruction method is inspired from the *Virtual Textures* [19], [20] acceleration structure and deferred evaluation.

## 3 Our Algorithm

### 3.1 Overview

Our algorithm is designed as a complement for a fluid simulation. We take as input the animated velocity field of a running fluid, computed separately. We want to add details to this fluid, using a procedural or image texture (see Fig. 2 and the accompanying video).

The simplest algorithm, mapping a texture to the fluid and letting it be deformed by the flow, is not acceptable: with time, the flow heavily distorts the texture, resulting in visible artifacts, even with a noise texture (see Fig. 1).

We generate a set of *deformable textured grids* that are advected with the flow. We start with a random Poisson disk distribution of *particles* and create regular grids

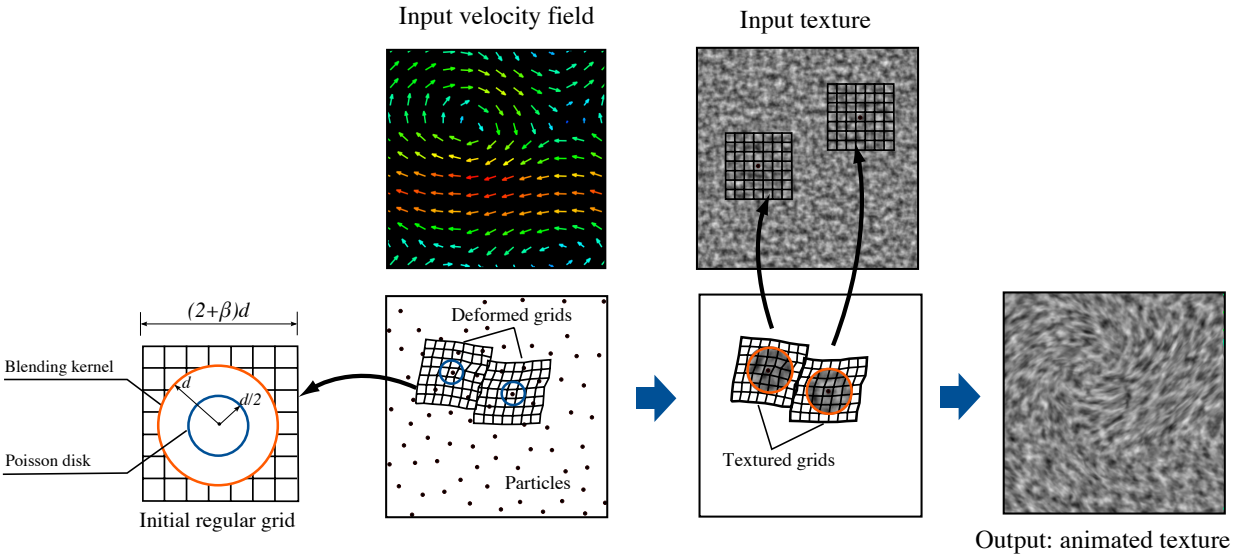



Fig. 2. Overview of our algorithm. We attach deformable grids to a set of particles which keep the Poisson-disk distribution. Each grid is mapped to a fixed area of the input texture. The particles and the nodes of the grids are both advected with the input flow. At rendering, we blend the textured grids and achieve an animated texture.

centered on these particles. Each grid is mapped to a random area of the input texture. At each time step we:

- Advect the grid vertices with the flow; set the position of each particle to the centroid of its advected grid.
- Maintain a uniform distribution of particles by killing and creating particles if necessary. We also kill particles whose grid is too distorted. We create regular grids for the new particles, using random areas of the input texture.
- 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. Grids are still advected and blended after their particle's death while they fade out.
- Render the animated texture either by directly drawing and blending the textured grids, or by using an indirection structure to recover the grids covering a given pixel.

Fig. 2 provides the overview of our algorithm. In the next section, we define precisely what is our input data. The remainder of this section details each step of the algorithm: placing the particles and advecting the grid vertices (Section 3.3), blending between neighboring grids (Section 3.4) and rendering the advected texture (Section 3.5).

### 3.2 Formal definition of our input data

Our input data has two key components: a velocity field and an input texture, plus optional scalar fields.

- The **velocity field** is defined on a low resolution grid, as velocity vectors at nodes. Such velocity fields are visualized in Fig. 1a, 2, 6b and 8a.
- Optionally, one or several scalar fields, also defined on a low resolution grid, can be provided to design the coarse aspect of the texture (such as the shape of the flame in Fig. 4 or the cloud bed in Fig. 5b).
- We formalize the **input texture** in a generic way:

$$T(\boldsymbol{x}) = F\left(a_0(\boldsymbol{x}), a_1(\boldsymbol{x}), a_2(\boldsymbol{x}), \ldots, a_j(\boldsymbol{x}) \ldots\right)$$

where the $\boldsymbol{x}$ is the texture coordinates, the $a_j$ are channels defined on these coordinates, and $F$ encodes the final aspect of the texture. The $a_j$ can be either defined as functions or sampled into 2-dimensional arrays (*i.e.*, image textures).

This definition is motivated by the joint use of image and procedural noise in real-life shaders, and by the channel-based decomposition used in [12] for Perlin noise (to ensure an artifact-free reconstruction during blending — see sections 3.4 and 3.5). It is generic enough to encode almost any texture-based representation, from classical image mapping to Perlin noise:

- Classical image mapping: a single channel $a_0$ contains the image texture, and $F$ is the identity.
- A texture based on Perlin noise:

$$\begin{aligned} P(\boldsymbol{x}) &= \sum_j \frac{1}{2^j} \left|2b(2^j\boldsymbol{x}) - 1\right| \\ T(\boldsymbol{x}) &= G(\boldsymbol{x}, P(\boldsymbol{x})) \end{aligned}$$

In this case, the $a_j$ correspond to several instances of the band limited noise function $b$ scaled by $\frac{1}{2^j}$, and $F$ is a function summing the absolute values and applying G. G creates the final procedural effects such as marble, clouds or fire, based on lerp, clamping, look-up textures, domain displacement...

Our technique recomposes a texture $T'(\boldsymbol{x})$ of arbitrary size having the same visual properties than $T(\boldsymbol{x})$, in a

way that is advectable:

$$T'(\boldsymbol{x}) = F\left(a'_0(\boldsymbol{x}), a'_1(\boldsymbol{x}), a'_2(\boldsymbol{x}), \ldots, a'_j(\boldsymbol{x}) \ldots\right)$$

where $a'_j$ are the recomposed version of $a_j$ channels. During the animation we advect all the $a'_j$ channels, then compute $F$ on these. The quality of the resulting texture depends on the decomposition between $a_j$ and $F$: our advection algorithm contains a blending phase while non-additive or highly structured perceptual features are prone to blending artifacts. So the texture $T$ should be decomposed such that the $a_j$ blend linearly, while $F$ can contain non-linear effects. Since $F$ is likely to contain costly functions, a good decomposition also brings efficiency.

The advected texture is meant to be used exactly like an ordinary texture (image or procedural), in a user-defined shader specifying the materials of a surface. This shader can be complex and mix several textures to control the various parameters of its BRDF, and various aspects of the appearance (color, transparency, bump, displacement...).

Numerous channels $a'_j$ might be advected in parallel. For simplicity, in most of the following we will denote $R$ the vector $(a_0, a_1, a_2, \ldots, a_j, \ldots)$ regardless of the number of channels, and $R'$ the corresponding one with the $a'_j$. We will refer to $R$ as the *reference texture*. $T' = F(R')$ will denote the final advected texture.

### 3.3 Particle Sampling and Distortion

#### 3.3.1 Particle Distribution

We maintain a dynamic Poisson-disk distribution of the particles using a modified version of the *boundary sampling* algorithm [21]; this algorithm creates a Poisson-disk sampling of a region in linear time, with the guarantee that there are no points left at a distance more than $d$ from a sample point. The boundary sampling algorithm works as follows: maintain the boundary of existing samples points (the set of points that are at a distance $d$ from exactly one sample point), then insert a new sample point on this boundary, recompute the boundary and iterate until the boundary is empty.

At each frame, we update dynamically the Poisson-disk distribution of particles: first, particles are advected using the velocity field; then, particles are killed in areas where the particle density becomes too high; finally, we insert new particles at a distance $d$ of other particles to maintain the Poisson-disk distribution.

For each particle, we create a regular grid (see Fig. 2, left), centered on it, of width larger than $2d$. Combined with the properties of the boundary sampling algorithm, this guarantees a gap-less coverage of the fluid.

The criterion for killing particles must be slightly different than the criterion for creating them, to avoid infinite loops of creation/destruction. We kill particles if they are at a distance less than $(1-\alpha)d$ from another particle. This removes the symmetry between creation and destruction, at the cost of a slight increase in the number of particles. To maintain this increase under control, we keep small values of $\alpha$ (in our implementation, $\alpha = 0.25$).

#### 3.3.2 Grid Advection and Particle Deletion

At each time step, we advect all the vertices of the grid with the velocity field of the flow. We use the new positions of the vertices to compute the new position of the particle as the center of mass of the grid vertices.

We kill a particle when the distortion of its associated grid becomes inconsistent with our quality criteria:

1) if the grid no longer covers the blending kernel,
2) if the distortion of the grid itself becomes too large,
3) if the grid folds over locally.

At their creation, grids are slightly larger than kernel size, to avoid triggering condition 1 too early. We create grids of width $(2+\beta)d$, with a small $\beta$ (in our implementation, $\beta = 0.6$). The size of the grid is a compromise between particle lifetime and the number of vertices it will require to ensure a given resolution.

#### 3.3.3 Estimating the Grid Distortion

For a single triangle in the grid, we compute distortion from the initial state with the singular values $\gamma_{\min}$ and $\gamma_{\max}$ of the Jacobian of the transform between the original triangle and the advected triangle [22], [23]. We define the distortion of a single triangle as:

$$\delta_t = \max\left(\gamma_{\max}, \frac{1}{\gamma_{\min}}\right) \tag{1}$$

We then define the *quality* of a triangle as the ratio of its distortion with the maximum acceptable distortion, $\delta_{\max}$:

$$Q_t = \max\left(\frac{\delta_{\max} - \delta_t}{\delta_{\max} - 1}, 0\right) \tag{2}$$

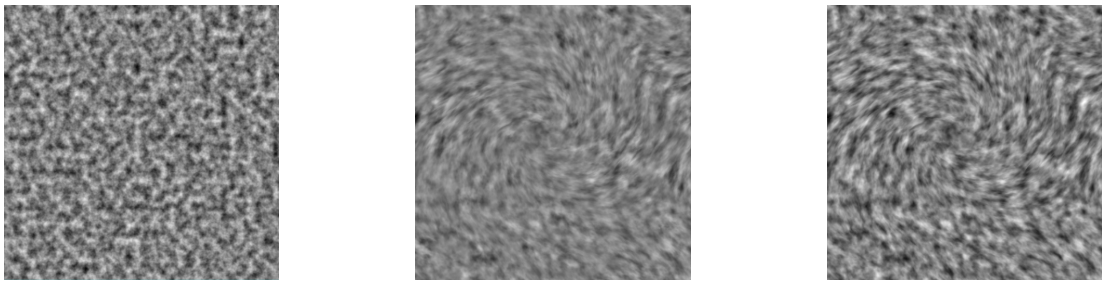
$Q_t$ is equal to 1 for an un-distorted triangle, and is equal to 0 for a triangle where the distortion is larger than $\delta_{\max}$. For each grid vertex $V$, we then compute its quality, $Q_V$ as the mean of the quality of its incident triangles. We kill a particle if, for any vertex in the grid, we have $Q_V < \frac{1}{2}$ (*i.e.*, we keep a margin of quality for the fading-out).

#### 3.3.4 Dealing with Boundaries

If the flow has boundaries, when a grid straddles one of these boundaries, the vertices outside of the boundary are not used for practical computations, but they are still advected to avoid unnecessary distortions. We extrapolate the velocity field outside of the boundary of the flow with a push-pull algorithm [23].

### 3.4 Blending and Continuity

To each advected grid $i$, we associate a domain $R_i$ in the reference texture $R$, by associating $(u, v)$ coordinates to each vertex of the grid at particle creation (in our implementation we choose the domains randomly). Each grid has its own texture mapping $\boldsymbol{u}_i(\boldsymbol{x})$, and its value at



(a) Original texture (b) Blending with Eq. 3 (c) Blending with Eq. 4

Fig. 3. Comparison of our two blending strategies. Note how Eq. 4 conserves the features of the original texture, while Eq. 3 loses contrast and shows heterogeneities.

a given location $\boldsymbol{x}$ is $R(\boldsymbol{u}_i(\boldsymbol{x}))$, where $\boldsymbol{u}_i(\boldsymbol{x})$ is obtained by interpolation of the values at grid nodes. At each frame, we reconstruct $R'$ by blending the textured grids taking into account the distortion created by the advection, then display the final advected texture $T' = F(R')$. To ensure a continuous blending both in space and time, we associate a weight $w_V$ to each vertex of the grid. The weight value of the grid $i$ at a given location $\boldsymbol{x}$ is $w_i(\boldsymbol{x})$, obtained by interpolation of the values at grid nodes.

For each pixel of the intermediate texture $R'$ to reconstruct, we take all the grids covering it and compute the weighted sum of the corresponding textures. The simplest way to do it is:

$$R'(\boldsymbol{x}) = \frac{\sum w_i(\boldsymbol{x}) R(\boldsymbol{u}_i(\boldsymbol{x}))}{\sum w_i(\boldsymbol{x})} \quad \text{for } i \in \text{ all grids} \tag{3}$$

Assuming the $R_i$ are independent samples of the reference texture $R$, Eq. 3 gives an intermediate texture $R'$ with the same mean as the reference texture, but the variance has been modified, changing the visible characteristics of the texture.

We use the following equation to blend the grids together while keeping the characteristics of the reference texture: its mean $\widehat{R}$ and its variance $\sigma_R^2$:

$$R'(\boldsymbol{x}) = \frac{\sum w_i(\boldsymbol{x})(R(\boldsymbol{u}_i(\boldsymbol{x})) - \widehat{R})}{\sqrt{\sum w_i^2(\boldsymbol{x})}} + \widehat{R} \tag{4}$$

Using this equation instead of the naïve blending of Eq. 3 restores the proper contrast of the texture; by comparison, Eq. 3 fades the contrast and introduces heterogeneities due to the varying number of grids contributing to a pixel (see Fig. 3). For a detailed proof of the properties of Eq. 4, please refer to the Appendix.

#### 3.4.1 Vertex Weights

The weight for each vertex is defined as the product of a spatial component and a temporal component.

$$w_V(t) = K_s(V) K_t(t) \tag{5}$$

The *temporal component* is simply a linear fade-in at the beginning of the life of a particle and a linear fade-out after the particle has been killed:

$$K_t(t) = \begin{cases} \frac{t}{\tau} & \text{if} \quad t < \tau \\ 1 & \text{if} \quad \tau < t < t_K \\ 1 - \frac{t - t_K}{\tau} & \text{if} \quad t_K < t < t_K + \tau \end{cases} \tag{6}$$

where $t_K$ is the time at which the particle is killed, and $\tau$ is the duration of the fading.

The *spatial component* merges three factors: the quality around each grid vertex ($Q_V$, defined in section 3.3.3), a fall-off with the distance to the particle (in our implementation we take it linear), and a continuity factor ensuring a weight 0 on the boundary of the grid (to avoid spatial discontinuities during blending):

$$K_s(V) = \left(1 - \frac{\|V - \boldsymbol{p}\|}{d}\right) d_V Q_V \tag{7}$$

$$\text{where} \quad d_V = \begin{cases} 0 & \text{if } V \in \text{grid boundary} \\ 1 & \text{otherwise} \end{cases} \tag{8}$$

The weights are computed for each vertex. During reconstruction, weights at arbitrary locations are interpolated from vertices values.

### 3.5 Reconstruction and Rendering

The advected texture $T'(x) = F(R'(\boldsymbol{x}))$ is used like a standard texture: mapped on a surface of the scene. $\boldsymbol{x}$ denote the texture coordinates of this mapping. $T'$ has to be computed at each frame. Rendering is done in two passes: in the first pass, we prepare data to compute each channel $a'_j$ of $R'$. During the final rendering of the scene, for each pixel of the textured surface, we compute the full texture function $T'(\boldsymbol{x})$. We present two different methods for reconstruction: a simple one (*Direct reconstruction*, section 3.5.1), and a more sophisticated one (*Indirect reconstruction*, section 3.5.2), that performs better on complicated scenes.

#### 3.5.1 Direct reconstruction

For direct reconstruction, in the first pass:

- Allocate the intermediate texture $R'$ at the required resolution (see discussion in 3.5.3), with one channel for each $a'_j$, plus one channel for the $\sum w_i$ (resp. $\sum w_i^2$ if using Eq. 4).

- For each particle $i$: splat its grid into the texture (*e.g.,* using a render target and the ordinary drawing API), thus accumulating the $\sum w_i(\boldsymbol{x})a_j(\boldsymbol{u}_i(\boldsymbol{x}))$ and the $\sum w_i(\boldsymbol{x})$ (resp. $\sum w_i^2$) into their respective channels.

During rendering in the fragment shader, for a given pixel (having texture coordinates $\boldsymbol{x}$):

- For each channel $a'_j$: finalize the channel value computation by dividing the accumulated values by $\sum w_i$ (resp. by $\sqrt{\sum w_i^2}$ if using Eq. 4).
- Compute the texture value for the current pixel, $F\left(R'(\boldsymbol{x})\right)$.
- Use the texture value as we would with a standard texture.

#### 3.5.2 Indirect reconstruction

The indirect reconstruction method is inspired from *virtual textures* [19], where a tile grid in texture space is used to store the ID of covering sprites. In a first pass:

- Allocate a (coarse) tile grid in texture space, covering the textured domain $T'$.
- For each advected particle:
  - Store its identity in all the cells that its grid intersects,
  - Generate a low-resolution image of its deformed grid (*e.g.,* using a render target and the ordinary drawing API), storing the deformed $\boldsymbol{u}_i(\boldsymbol{x})$ and $w_i(\boldsymbol{x})$ fields.

During rendering in the fragment shader, for a given pixel (having texture coordinates $\boldsymbol{x}$):

- Find which cell of the tile grid corresponds to the current pixel.
- For all of the deformed particle grids $i$ covering this cell:
  - Find the position corresponding to the current pixel in the associated low-res images $\boldsymbol{u}_i$ and $w_i$.
  - fetch $\boldsymbol{u}_i(\boldsymbol{x})$ and $w_i(\boldsymbol{x})$ (*i.e.,* texture access with bilinear interpolation).
  - fetch $R(\boldsymbol{u}_i(\boldsymbol{x}))$.
- Sum all $w_i(\boldsymbol{x})R(\boldsymbol{u}_i(\boldsymbol{x}))$, divide by $\sum_i w_i$ (resp. by $\sqrt{\sum_i w_i^2}$ if using Eq. 4).
- Compute the texture value for the current pixel, $F(R'(\boldsymbol{x}))$.
- Use the texture value as we would with a standard texture.

#### 3.5.3 Discussion

The direct method is simpler and easier to implement, but needs to compute and store the whole texture, at the required resolution, for all channels. Depending on the geometry of the scene, the required resolution might be quite large (if there are textured objects close to the viewpoint, or if the surface being mapped is wide — *e.g.,* a whole river), and required memory grows with the number of textured objects since visibility cannot be accounted for. In this situation the memory and computational cost of the direct method become prohibitive. It gets worse for a large number of channels $a_j$.

The indirect method is more efficient, both in memory and computation time: the tile grid is much smaller than the size of the texture itself, the images computed for each grid can be very low resolution, do not depend on the channels of the input texture, and only the sampled texels are evaluated.

## 4 Results and Comparison

All pictures and timings in this paper and in the companion video[2] were computed on an Intel Core i7, running at 2.67 GHz, with an Nvidia GeForce GTX 275.

### 4.1 Results

As you can see on Fig. 4 and 5, as well as the accompanying video, our algorithm can be used in many graphics applications for adding details to low resolution simulation, whether it is for fire, clouds or rivers. The advected texture can be used to change the colors of the flow, or its normals, or even as a displacement map.

### 4.2 Performance and Timings

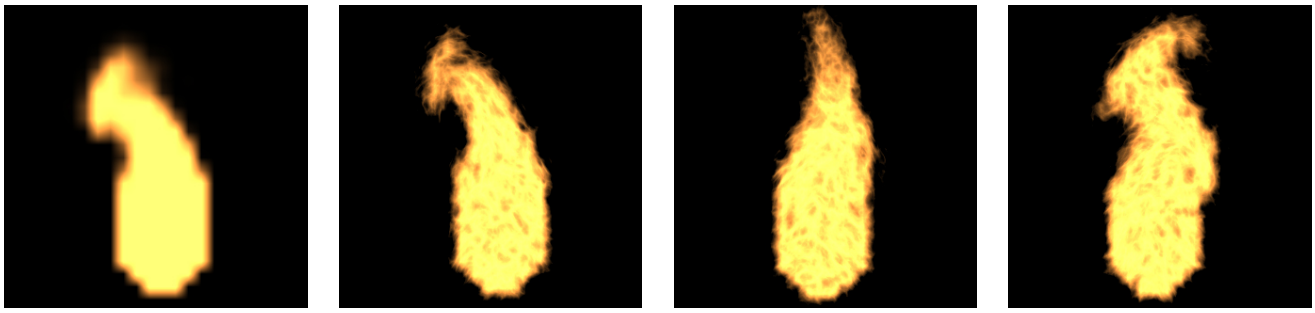
One of the strongest advantages of our method is that it runs in real-time, making it useful for interactive applications, such as video-games, exploration of virtual worlds, just-in-time generation of content and virtual modeling.

For Fig. 4, 6, 7, 8 and most of the video sequences, we used a fluid covering the entire picture, an output texture size of 512×512, and 300 grids of 8×8 vertices (including grids being faded in or faded out). The timings correspond to the fire example (Fig. 4).

- Using *direct reconstruction* (section 3.5.1), the total overhead of computing and rendering the advected texture is just 9 ms. This corresponds to 6.5 ms of CPU time for handling particles and grids (interpolating velocities, evaluating deformation and maintaining Poisson distribution) and 5.5 ms of GPU time for reconstructing the advected texture. The total time is less than the sum because the two processors operate partly in parallel.
- Using *indirect reconstruction* (section 3.5.2), with our implementation, the rendering time is 25 ms. The time for handling particles and grids is the same as with direct rendering, 6.5 ms.

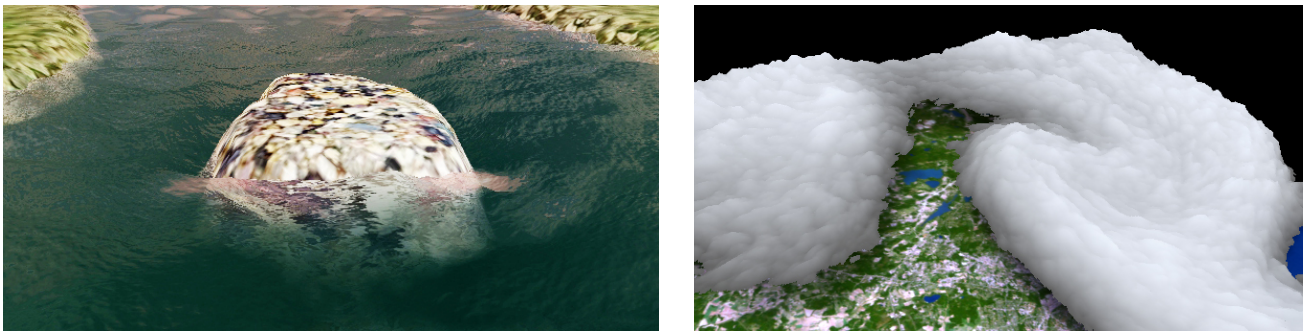
The reasons for the difference of performance are twofold. First, we are in the worst case for indirect reconstruction and the best case for the direct reconstruction: the entire fluid domain is displayed on screen, and we use a very simple shader. Second, our GPU implementation of virtual textures is not optimized: we simply implemented a regular tiling with a fixed number

2. available at http://evasion.imag.fr/Membres/Qizhi.Yu/.



(a) Low res. simulation (b) Frame 0 (c) Frame 60 (d) Frame 120

Fig. 4. Using a Perlin noise reference texture advected by our method to add details at (512×512) resolution to a low resolution 2D fire simulation (32×32 velocity and marker density fields, cf (a) ). Here, the reference texture is analytic (no storage): 4 bands of time varying noise are used, $a_j(\boldsymbol{x}) = b_j(2^j\boldsymbol{x}, t)$ and $F(\{a_j\}) = LUT\left(dens(\boldsymbol{x}, t) + scale\sum\frac{1}{2^j}(1 - |2a_j - 1|)\right)$. See also the accompanying video.



(a) Water waves (b) Cloud layer

Fig. 5. Applications of Perlin noise textures advected by our method. (a) Using one advected texture to modulate the height field of a river surface for representing large waves, and using another one as the normal map for representing small waves. We use 4 bands of time varying noise for each, with $F(\{a_j\}) = scale\sum\frac{1}{2^j}a_j$. (b) Using an advected texture to modulate the thickness of a cloud layer. We use 4 bands with $F(\{a_j\}) = scale\sum\frac{1}{2^j}|2a_j - 1|$. See also the accompanying video.

TABLE 1
Timing results for a zoom in the fire example, using a fixed-size viewport ($256 \times 256$).

| Zoom factor | Texture resolution | Direct method | Indirect method |
|---|---|---|---|
| 1 | $256 \times 256$ | 9 ms | 10 ms |
| 4 | $1024 \times 1024$ | 20 ms | 10 ms |
| 16 | $4096 \times 4096$ | 410 ms | 10 ms |

of slots per tile (256, which is overestimated). It would be feasible to avoid blending the zero-contribution of unused slots, or even to use dynamics structures as in [20].

If we zoom on the object, more texture resolution is needed. The direct method requires to allocate and compute the full size texture even if only a part is visible on screen, while the rendering time of the indirect method remains constant since only visible pixels are rendered (see Table 1). The same behavior would occur for a rotation from facing view to grazing view angle.

The computation cost of the *advection phase* is proportional to the overall number of vertices: doubling the number of particles or doubling the number of vertices per grid will both have the effect of doubling the computation time for advection.

The computation cost of the *rendering phase* depends on: the number of texels on which we make the computation (the total size of the texture for the direct reconstruction, the number of sampled — *i.e.*, visible — texels for the indirect reconstruction), the number of channels, and the cost of $F$.

### 4.3 Evaluation and comparison

#### 4.3.1 *Evaluating the Quality of the Animated Texture*

In order to evaluate the quality of an animated texture, we suggest two criteria: the optical flow of the final animated texture and its Fourier spectrum. Both properties are computed on the generated animated texture, $T'$, on the fly. For optical flow, we used the Lucas-Kanade method [24] in the OpenCV library, and set the search window size to $1/20^{\text{th}}$ of the output picture size.

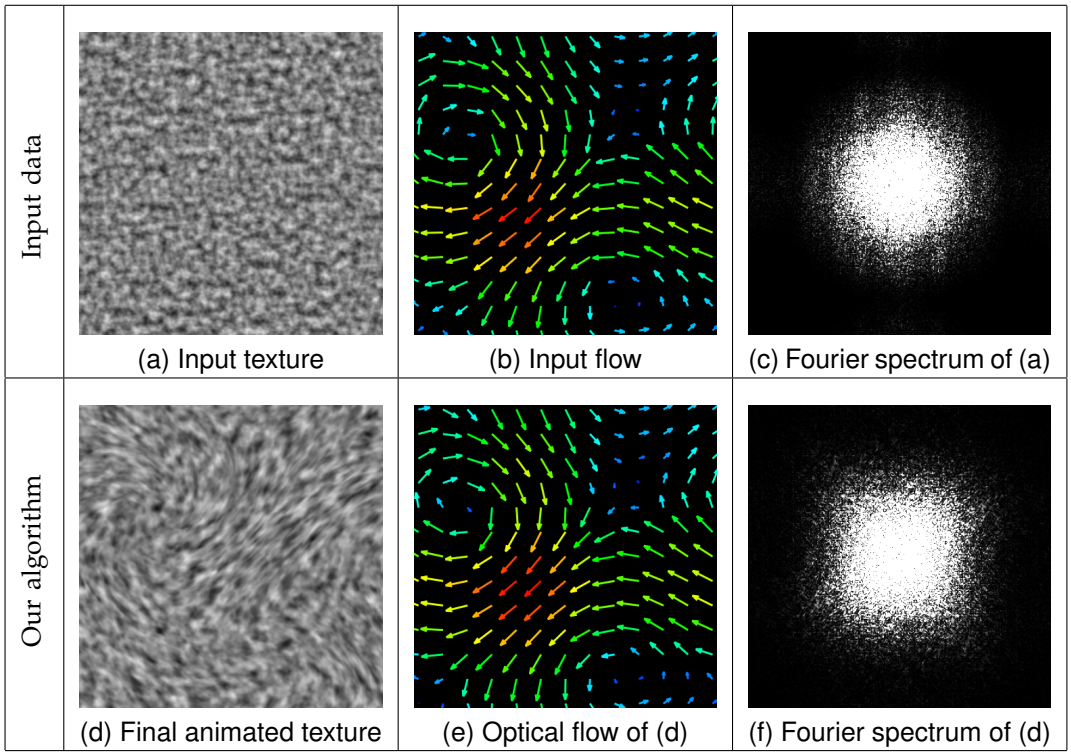



Fig. 6. Advecting a Perlin noise texture using our method. The optical flow (e) is computed directly on the final animated texture (d) and matches perfectly with the input flow (b). The Fourier spectrum of the final animated texture (f) also matches well with that of the input texture (c). See also the accompanying video.

Ideally, the optical flow of the synthesized animated texture should match the velocity field we used as input, while its Fourier spectrum should match the Fourier spectrum of the input texture. As can be seen in Fig. 6 and the accompanying video, our algorithm works very well on both points.

#### 4.3.2 Comparison with Naïve Implementation

As you can see in Fig. 1 and the accompanying video, it is not acceptable to simply advect a noise texture to follow the velocity field: after a few frames, the input texture is heavily distorted and visibly anisotropic along the directions of the flow.

#### 4.3.3 Comparison with Eulerian Texture Advection

Eulerian texture advection methods, such as [11], overcome the limitations of the naïve approach by regenerating the texture periodically. The time between successive regenerations is called the *latency*.

Max and Becker [11] used a single latency for the entire domain. Their method preserves the optical flow at the cost of stretching the texture in fast areas. Adjusting the latency for these fast areas breaks the illusion of motion (and thus the optical flow) in slow areas. See Fig. 7 and the companion video.

To overcome this limitation, Neyret [12] used several texture layers that regenerate periodically with different latencies. For each region of the fluid they pick the best layer depending on the local distortion rate, much like a MIPmap level is picked depending on the LOD. While this method performs better than [11], it can still generate distorted textures for complex velocity fields (see Fig. 8 and the companion video).

For all Eulerian methods, the texture will be explicitly advected, stored and reconstructed on the entire fluid domain; as a consequence, the resolution must fit the most demanding point of view, and the non-visible areas are computed anyway.

#### 4.3.4 Comparison with Sprite-Based Texture Advection

Yu et al. [9] simulate animated rivers by advecting sprites. Their approach has similarities with our work. However they advect solid particles, while we advect deformable grids. Their method gives a "blocky" velocity field, and 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). Deformable grids are a natural improvement over [9].

#### 4.3.5 Comparison with Flow-Guided Texture Synthesis

Flow-guided texture resynthesis techniques, such as Kwatra *et al.* [15], [16], take the same input and produce the same output as our work. There are two main differences. First, they measure texture similarities using neighborhoods while we use the Fourier spectrum, and they put little emphasis on the accurate reproduction of the input velocity field. Our experiments show that

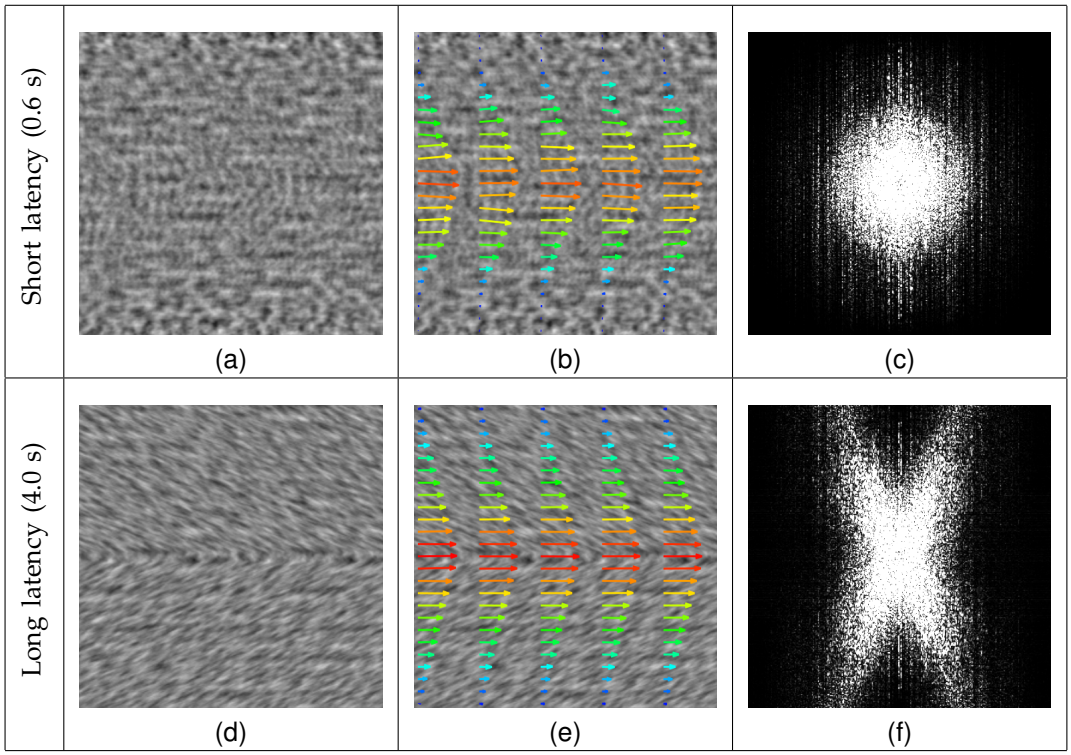
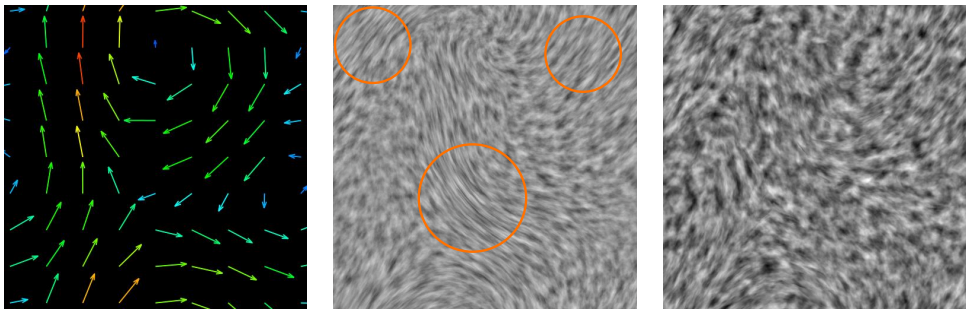



Fig. 7. Advecting a Perlin noise texture with a horizontal shear flow using the basic Eulerian texture advection method [11]. *Top*: with a short regeneration latency (0.6 s) the texture is echoed (a) and the optical flow is incorrect (b), but the Fourier spectrum is almost preserved (c). *Bottom*: with a long latency (4.0 s) the texture is too stretched (d) and the Fourier spectrum is distorted (f), but the optical flow (e) matches the input velocity field.



(a) Input flow (b) Multi-layer Eulerian texture advection [12] (c) Our algorithm

Fig. 8. Comparison with multi-layer Eulerian texture advection method [12]. We advect a Perlin noise texture with the input flow of (a). We used three texture layers with latencies of 0.6 s, 2.3 s and 4.0 s (b). In some places (marked with circles), the method still results in overstretching. Our algorithm maintains the properties of the noise texture (c).

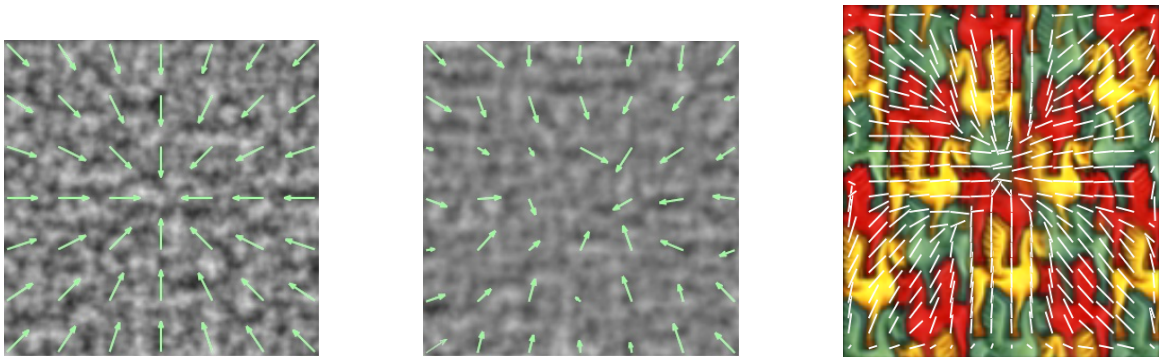
these methods tend to give rigid moving chunks around structured features and show sudden changes in the pattern. In other words the resulting optical flow does not accurately match the input flow (see Fig. 9 and the companion video).

Second, since texture resynthesis algorithms work by identifying neighborhoods (and thus structures) in the input textures, they tend to give unreliable results for textures without recognizable features, such as noise textures (see Fig. 9 and the companion video)[3], or to recognize and repeat a feature meant to be random and unique.

Lefebvre and Hoppe [18] have designed a different algorithm for flow guided texture synthesis, running on the GPU. A side-by-side comparison with their algorithm using a noise texture as input (see the companion video) shows that it does not conserve the velocity field and results in blocky artefacts and temporal discontinuities[4].

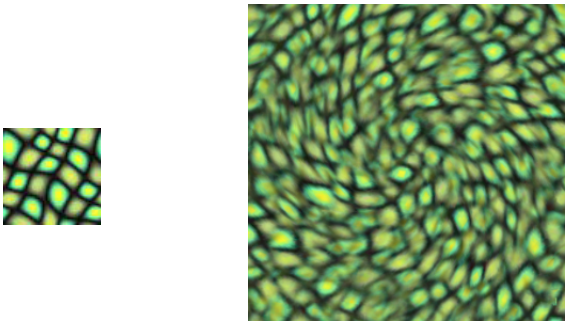
3. The texture resynthesis examples used in Fig. 9 and the video were kindly provided by V. Kwatra.

4. The texture resynthesis examples used in the video were kindly provided by S. Lefebvre.



(a) Input data (b) Flow-guided texture resynthesis [15] (c) Texture resynthesis from an image

Fig. 9. Comparison with flow-guided texture resynthesis technique [15] (see also the companion video). For (a,b) the input texture is a Perlin noise, and the input flow is a sink at the center of the picture (a). Flow-guided texture synthesis methods (b) do not match accurately the input velocity field, and the synthesized texture becomes blurry. For (c) the input texture is an image with structured pattern. Flow-guided texture synthesis methods do not match accurately the input velocity field.



(a) Input data (b) Our algorithm

Fig. 10. A failure case for our algorithm: if the input texture is an image with large regular structured features, our algorithm does not preserve them (see also the companion video).

### 4.4 Discussion

Due to its properties our algorithm works well with noise textures and procedural textures. Our experiments show that it also applies to a large range of input textures (see Fig. 11 and the accompanying video), including bubbles, foam and froth. Our algorithm places a single requirement 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 perceptual features are resistant to blending. For example, our algorithm works well with pictures of bubbles because blending together two pictures of a bubble produces a convincing bubble (or two bubbles glued together). It does not work, however, when blending texture images with large regular structured features, such as a checkerboard texture (see Fig. 10 and the companion video).

For procedural textures, which often show very strong structures at small and large scales, the quality of our algorithm depends strongly on the decomposition of the original texture between $F$ and the $a_j$ channels. If the $a_j$ blend nicely, the algorithm will produce a nice result even if $F$ creates structured patterns. *E.g.*, for Perlin noise, a blending artifact-free quality result is obtained by storing a vector of base noise in the $a_j$ (see [12] for illustrations and comparisons).

We think that the set of texture images 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 realism.

Flow-guided texture synthesis algorithms [15], [16], [18] preserve large-scale features of the input texture but loose other texture properties, do not conform accurately to the input flow, and in some cases require a long precomputation and several minutes per frame [15], [16]. We think that both algorithms have their benefits, depending on the application 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 ultra-light cost of our algorithm 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 experiment with volumetric rendering (in particular, Gigavoxel [25] approach shares the virtual texture principle). We would also like to extend our method to use a Poisson disk sampling in screen space as in [9], to get a view dependent LOD mechanism. One of our long term goal is to integrate detailed lively water and clouds in real-time landscape browsers such as GoogleEarth or games.

In the scope of texture synthesis techniques, we could try to replace the random selection of domains in the reference texture for new grids with a smarter method, in order to conserve larger features, or structures. Also, it would be interesting to study how to decompose some example pattern images into $F$ and $a_j$, as a better conditioning for computations. Finally, we think that our approach could be adapted to parameterization-free texturing in the spirit of [20], [26].

### Acknowledgements

We would like to thank Vivek Kwatra (Google Inc.) for providing the results of their algorithm on noise textures and Sylvain Lefebvre (INRIA, France) for many helpful discussions. Thanks to Antonio Garcia Castaneda (UCL) for proofreading an early version of this paper. The work of the first author was supported by a Marie Curie PhD grant through the *VISITOR* project.

## APPENDIX
## DETAILED PROOF OF EQ. 4

We assume that the $R_i = R(\boldsymbol{u}_i(\boldsymbol{x}))$ are independent variables. We also assume that the variations of $R_i$ are of much higher frequency than the variations of the weights $w_i$, so that the $w_i$ can be treated as constants with respect to the $R_i$. With these hypotheses, the mean of the resulting texture $R'(\boldsymbol{x})$ is:

$$\begin{aligned}\widehat{R'} &= \frac{\sum w_i(\boldsymbol{x})(\widehat{R_i} - \widehat{R})}{\sqrt{\sum w_i^2(\boldsymbol{x})}} + \widehat{R} \\ &= \frac{\sum w_i(\boldsymbol{x}) \times 0}{\sqrt{\sum w_i^2(\boldsymbol{x})}} + \widehat{R} \\ &= \widehat{R}\end{aligned}$$

The variance of $R'(\boldsymbol{x})$ is, by definition, the variance of this sum:

$$\sigma^2_{R'} = \sigma^2\left(\frac{\sum w_i(\boldsymbol{x})(R(\boldsymbol{u}_i(\boldsymbol{x})) - \widehat{R})}{\sqrt{\sum w_i^2(\boldsymbol{x})}} + \widehat{R}\right)$$

If the $R_i$ are independent variables, then their respective covariance is null and we can expand the sum, using the following rules:

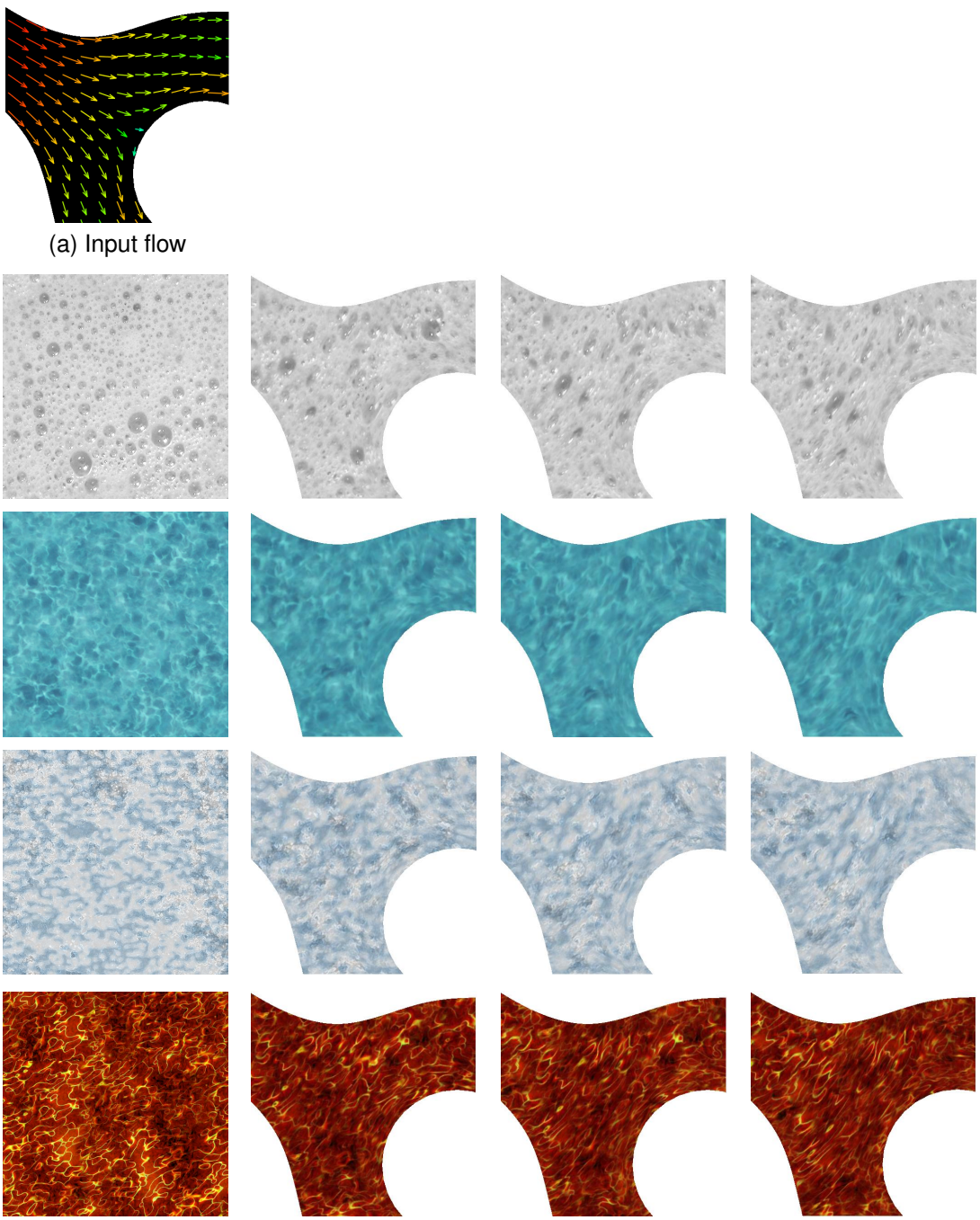
$$\begin{aligned}\sigma^2(X+Y) &= \sigma^2(X) + \sigma^2(Y) + 2\mathrm{cov}(X,Y) \\ \sigma^2(\lambda X) &= \lambda^2\sigma^2(X)\end{aligned}$$

This gives:

$$\begin{aligned}\sigma^2_{R'} &= \sum\left(\frac{w_i(\boldsymbol{x})}{\sqrt{\sum w_i^2(\boldsymbol{x})}}\right)^2 \sigma^2(R(\boldsymbol{u}_i(\boldsymbol{x}))) \\ &= \sum\frac{w_i^2(\boldsymbol{x})}{\sum w_i^2(\boldsymbol{x})}\sigma^2_R \\ &= \sigma^2_R\end{aligned}$$

## REFERENCES

[1] W. T. Reeves, "Particle systems – a technique for modeling a class of fuzzy objects," *ACM Trans. Graph.*, vol. 2, no. 2, pp. 91–108, Apr. 1983.
[2] W. T. Reeves and R. Blau, "Approximate and probabilistic algorithms for shading and rendering structured particle systems," in *Computer Graphics (SIGGRAPH '85)*, vol. 19, no. 3, 1985, pp. 313–322.
[3] C. W. Reynolds, "Flocks, herds, and schools: A distributed behavioral model," *Computer Graphics (SIGGRAPH '87)*, vol. 21, no. 4, pp. 25–34, 1987.
[4] K. Sims, "Particle animation and rendering using data parallel computation," *Computer Graphics (SIGGRAPH '90)*, vol. 24, no. 4, 1990.
[5] "Wondertouch software." [Online]. Available: http://www.wondertouch.com/
[6] D. Ikeler and J. Cohen, "The use of Spryticle in the visual FX for 'The Road to El Dorado'," in *SIGGRAPH sketches*, 2000.
[7] J. Stam and E. Fiume, "Depicting fire and other gaseous phenomena using diffusion processes," in *SIGGRAPH '95*, 1995, pp. 129–136.
[8] R. Narain, J. Sewall, M. Carlson, and M. C. Lin, "Fast animation of turbulence using energy transport and procedural synthesis," *ACM Trans. Graph.*, vol. 27, no. 5, p. 166, 2008.
[9] Q. Yu, F. Neyret, E. Bruneton, and N. Holzschuch, "Scalable real-time animation of rivers," *Computer Graphics Forum*, vol. 28, no. 2, pp. 239–248, 2009.
[10] N. Max, R. Crawfis, and D. Williams, "Visualizing wind velocities by advecting cloud textures," in *VIS '92: 3rd conference on Visualization '92*, 1992, pp. 179–184.
[11] N. Max and B. Becker, "Flow visualization using moving textures," in *ICASW/LaRC Symposium on Visualizing Time-Varying Data*, 1995, pp. 77–87.
[12] F. Neyret, "Advected textures," in *Symposium on Computer Animation*, 2003, pp. 147–153.
[13] J. J. van Wijk, "Image based flow visualization," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 745–754, 2002.
[14] J. Stam, "Stable fluids," in *SIGGRAPH '99*, 1999, pp. 121–128.
[15] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra, "Texture optimization for example-based synthesis," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 795–802, 2005.
[16] V. Kwatra, D. Adalsteinsson, T. Kim, N. Kwatra, M. Carlson, and M. Lin, "Texturing fluids," *IEEE Trans. Vis. Comput. Graphics*, vol. 13, no. 5, pp. 939–952, 2007.
[17] A. W. Bargteil, F. Sin, J. E. Michaels, T. G. Goktekin, and J. F. O'Brien, "A texture synthesis method for liquid animations," in *Symposium on Computer Animation*, 2006, pp. 345–351.
[18] S. Lefebvre and H. Hoppe, "Appearance-space texture synthesis," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 541–548, 2006.
[19] S. Lefebvre and F. Neyret, "Pattern based procedural textures," in *Symposium on Interactive 3D graphics*, 2003, pp. 203–212.
[20] S. Lefebvre, S. Hornus, and F. Neyret, "Texture sprites: Texture elements splatted on surfaces," in *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)*, April 2005.
[21] D. Dunbar and G. Humphreys, "A spatial data structure for fast poisson-disk sample generation," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 503–508, 2006.
[22] O. Sorkine, D. Cohen-Or, R. Goldenthal, and D. Lischinski, "Bounded-distortion piecewise mesh parameterization," in *VIS '02: Visualization*, 2002, pp. 355–362.
[23] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe, "Texture mapping progressive meshes," in *SIGGRAPH '01*, 2001, pp. 409–416.
[24] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *IJCAI'81: Proceedings of the 7th international joint conference on Artificial intelligence*, 1981, pp. 674–679.
[25] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering," in *ACM-SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Feb. 2009.
[26] S. Lefebvre, S. Hornus, and F. Neyret, "Octree textures on the GPU," in *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005, pp. 595–613.



(a) Input flow

(b) Input textures (c) Frame 0 (d) Frame 30 (e) Frame 60

Fig. 11. Using our method to advect non-noise image textures with a wall-bounded flow. See also the companion video, available at http://evasion.imag.fr/Membres/Qizhi.Yu/.