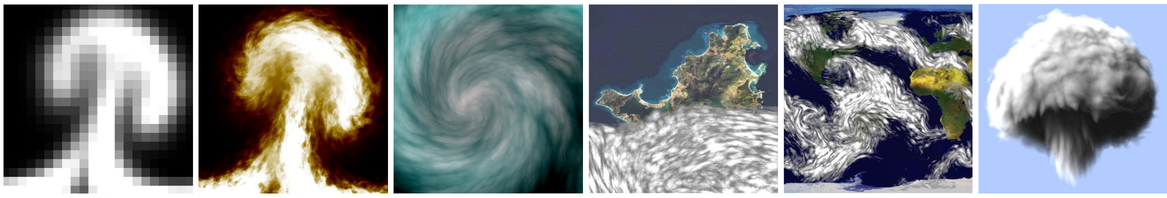
Eurographics/SIGGRAPH Symposium on Computer Animation (2003)
D. Breen, M. Lin (Editors)

# Advected Textures

Fabrice Neyret

Fabrice.Neyret@imag.fr
iMAGIS-GRAVIR† / IMAG-INRIA



**Abstract**

*Game and special effects artists like to rely on textures (image or procedural) to specify the details of surface aspect. In this paper, we address the problem of applying **textures** to **animated fluids**. The purpose is to allow artists to increase the details of flowing water, foam, lava, mud, flames, cloud layers, etc.*

*Our first contribution is a new algorithm for **advecting textures**, which compromises between two contradictory requirements: continuity in space and time and preservation of statistical texture properties. It consist of combining layers of advected (periodically regenerated) parameterizations according to a criterion based on the local accumulated deformation. To correctly achieve this combination, we introduce a way of **blending procedural textures** while avoiding classical interpolation artifacts. Lastly, we propose a scheme to add and control **small scale texture animation** amplifying the low resolution simulation. Our results illustrate how these three contributions solve the major visual flaws of textured fluids.*

## 1. Introduction

Fluids phenomena such as rivers, lava, mud, flames or clouds are more and more present in games and special effects. Numerous Computational Fluid Dynamics -oriented contributions have been proposed so far to simulate fluids. However, relying on this approach for generating detailed flows is not convenient since it suffers severe drawbacks when used at high resolution: simulation requires huge computation and storage while offering very little control to the artist. Numerous interesting arguments for not using CFD in CG systematically are given in Lamorlette *et al* [4]. Moreover, the physics of small scale phenomena can be different to the one at large scale (see footnote *I*), or parameters can be unknown which is frequent for natural objects (*e.g.,* mud, lava, foam, cloud layer...). Conversely, artists want to control the visual aspect of the details, *e.g.,* using textures. Thus, a natural solution is to simulate a low resolution fluid, then to let it *advect* (*i.e.,* carry) a user-defined texture.

Advecting textures usually consists of advecting a parameterization. This suffers from several drawbacks since it should fulfill contradictory criteria: **space and time continuity** of the result should be ensured while **statistical properties** of the texture should be preserved[I] within a range. Moreover, fluids exhibit motion and swirl at every scale. Thus the texture should not be simply passively advected: It should **amplify the low resolution animation** the same way it enhances its spatial appearance (*i.e.,* increase the resolution) as seen in the flame example on the teaser image and on the video.

In this paper, we propose a workflow for advecting textures fulfilling these three requirements. For simplicity, we illustrate it mostly in 2D but the method also applies in 3D.

† GRAVIR is a joint lab of CNRS, INRIA, Institut National Polytechnique de Grenoble and Université Joseph Fourier.
`http://www-imagis.imag.fr/Membres/Fabrice.Neyret/`

*I* The physical base for this conservation of the pattern is that small scale active phenomena often oppose large scale advection and diffusion, restoring or recreating the characteristic pattern. For example the shape of crust chunks in lava corresponds to mechanical and thermal local constraints. The shape of individual clouds in a cloud layer is connected to Benard-cell like local circulation. Vortices are permanently recreated in a turbulent flow. Same for foam, etc.
We certainly don't want to simulate this small scale physically. Moreover, the artist knows the global aspect this should have – or he wants this to have.

### 1.1. Previous Work

We will not review here the previous work on Computational Fluid Dynamics for CG. In terms of performances, the fastest algorithm is the *Stable Fluids* [11, 12] approach, allowing real-time simulation at low resolution by suppressing the time-step constraint. However, it still has a $N \log N$ complexity (where N is the number of grid vertices) and requires a huge amount of memory so that it can become impractical at high resolutions, especially in 3D.

Three classes of approaches exist in the literature to put textures in motion:

- Simple advection of the parametric space:
  For 2D fluids and 3D mist, Ebert [2, 1] procedurally deforms the space on which the procedural texture is computed. $(u, v)$ texture coordinates are simply advected instead of density[II]. The drawback is that the noise texture stretches along time since the parameterization is more and more deformed. Thus the statistical properties of the noise are not preserved. Moreover, the amount of stretch evolves in time even on a steady flow.
  Musgrave's fire balls model [6] follows a similar approach without suffering stretching. In his specific case the absolute deformation of the domain is local (at the expense of realism) and the objects have limited spatial extension.
  For scientific visualization purposes, Max and Becker [5] rely on the blending of three textures that are periodically regenerated in time. This principle is also used by Stam [11] for faking high resolution smoke by combining density textures with the low resolution flow. This regeneration prevents the stretching from increasing on steady flows. However, the amount of stretching depends on the regeneration rate (called *latency*). Tuning this rate is an issue since it should be tuned differently depending on the location. We discuss this problem further in section 1.3. Moreover, blending introduces classical image interpolation artifacts such as ghosting effects and contrast loss which alter the statistical properties of the texture.

- Relying on particles instead of parameterization to control the location of texture patterns: DreamWorks introduced *Spriticles* for the feature movie *Quest for Eldorado*. PDI attached spherical hypertextures to particles to make turbulent flames for *Shrek* [7]. A model for combining local parameterization and continuity constraints was proposed for lava flows [14]. In the first and last cases, continuity is still an issue, which limits the use of these techniques to choppy aspects of fluid surfaces. Note that each of these three methods also propose a small scale animation (*i.e.,* at higher resolution than the particle sampling): cartoon-animated texture for the first one and time-dependent procedural texture for the two others.

- Introducing time as a fourth dimension in the procedural texture parameters: This is commonly done in CG industry to get animated cloud, sky or mist based on Perlin noise [8]. Since statistical time properties of swirling flows differ from statistical space properties, a *flownoise* model with embedded time properties has been introduced [10].
  However, these procedural methods embedding time only address the amplification of animation details. To address global advection (*i.e.,* long range motion) one has to rely on either of the two classes of approaches mentioned above.

*II* Note that the Stam *et al* idea of inverse warping of rays for rendering distorted blobs in [13] is quite equivalent.

### 1.2. Requirements for Quality Advected Textures

There are two requirements for good-looking advection:

- Continuity in space (no cracks) and time (no popping, steady result on steady flows).
- Control of the spatial statistical properties (*i.e.,* the maximum stretch allowed), as justified in footnote *I*.

Methods of the first class mentioned above only ensure the first requirement while methods of the second class only ensure the second. Thus, correct advection is currently an open problem.

Two more requirements concern the temporal properties:

- Blending operations when combining textures should not alter the statistical space properties. Classical flaws are *ghosting effects* due to the addition of images in which a remarkable feature doesn't show at the same place and *contrast loss* due to averaging.
- In addition to their advection, the visual details brought by the texture should be animated at small scale. This small scale animation should be related to the flow activity (*i.e.,* peaceful or turbulent).

Despite morphing techniques [15] addressing feature preservation during the transition between images exist, no such technique has been developed for animated textures. The parameters of procedural textures can be interpolated, but this would not make sense for texture coordinates (see Figure 1). Thus blending issues are not addressed by existing methods.

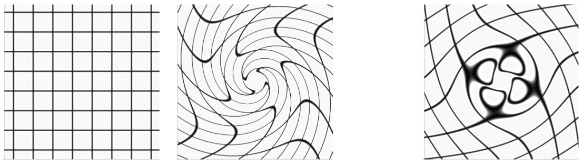


**Figure 1:** From left to right*: canonical mapping, swirled mapping, linear interpolation of the two mappings. This illustrates that interpolating between parameterizations can yield arbitrary results. Moreover, the weight shift along time induces unwanted motion.*

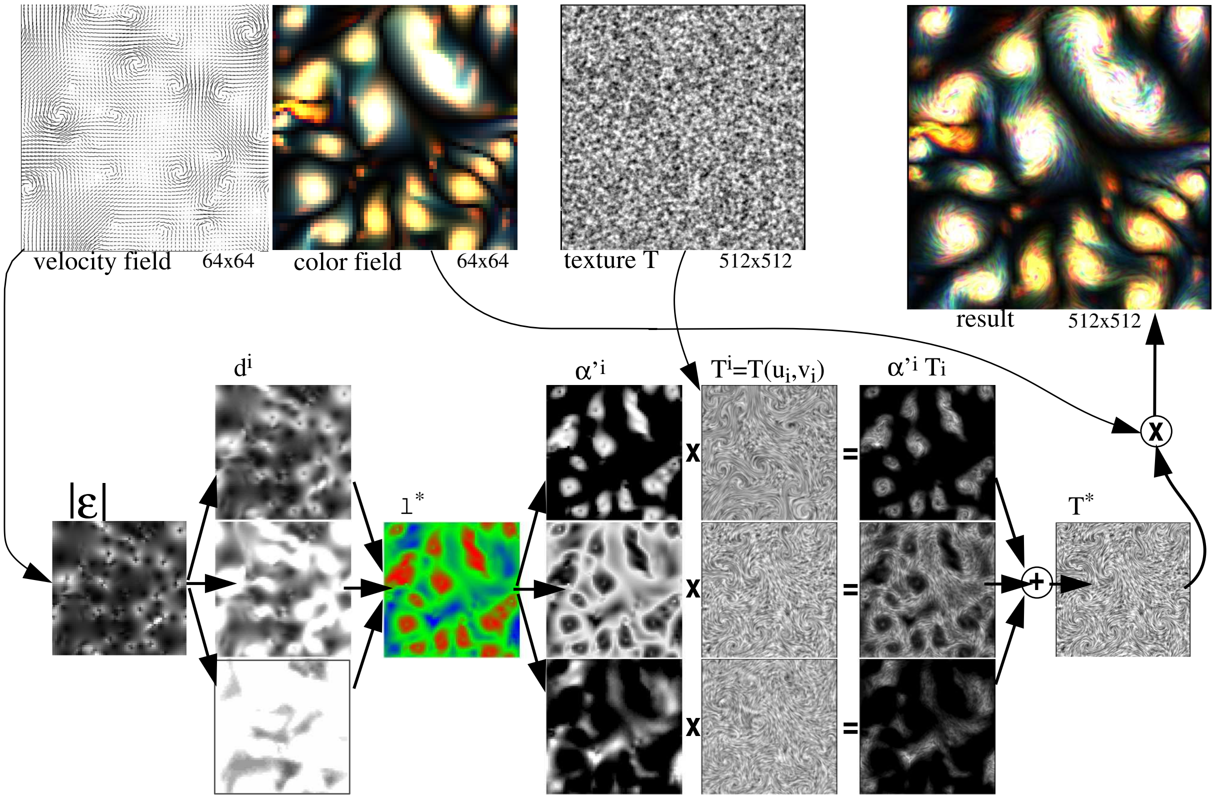



**Figure 2:** *Our advection scheme using $N = 3$ layers: Low resolution density (or color) field and velocity field are provided, as well as a texture (either image or procedural – here Perlin noise is precomputed into an image ). We compute the instant deformation (norm of the strain tensor $\|\varepsilon\|$). We use it to update the accumulated deformation $d^i$ of each layer. Knowing the target deformation $d^*$, we get the ideal (decimal) layer number $l^*$ (false colors red, green and blue stand for 1,2 and 3). The fractional part is used to weight the corresponding layer, thus yielding three masks selecting the appropriate pixels. By multiplying with the corresponding advected texture $T^i$, we get three adapted texture components that are blended to get the final texture. The resulting image is obtained by multiplying this texture by the density field. Note that each $T^i$ results from the blending of three phase-shifted textures $T^i_j$ (not displayed here). Similarly each $d^i$ results from the sum of the three $d^i_j$.*

### 1.3. Flaws of Classical Regenerated Textures

Regenerating a texture (image or procedural) consists of resetting texture coordinates to window coordinates after a given life time (or latency) $\tau$. To ensure time-continuity, the texture is affected with a weight factor $\alpha$ fading to zero at the beginning and end of the life time, *e.g.,* $\frac{1}{2}\left(1-\cos(2\pi\frac{t}{\tau})\right)$. Blending three $\frac{2\pi}{3}$ phase-shifted textures ensures a constant weight [5]. It works visually well for pseudo-periodic[III] textures due to an optical illusion: when following a fading spot in motion, the observer identifies it with surrounding spots rather than with the spot reseted to the initial location. This requires that the spot has traveled more than a pseudo-period (consider a factor of 10 in practice).

Thus the latency must neither be too high (to prevent large stretching) nor too low (to preserve the illusion of motion). The ideal value depends on the velocity and the deformation, so it should be adapted to the fluid state. The problem is that at this stage there is a single global latency value while there is a range of velocity values along the simulated space (see Figure 3). A major contribution of our paper is to propose a solution to this problem.

### 1.4. Overview of Our Approach

Our three contributions consists of **locally adapting the texture latency** which we describe in section 2, **blending the procedural noise in frequency space** which we describe in section 3, and **defining a control mechanism for small scale animation** which we describe in section 4. Results are presented in section 5.

These three steps define a *complete animated fluid texturing scheme* relying on procedural textures. If image textures are used instead (*e.g.,* for basic hardware-accelerated rendering), the first step still applies.

[III] Note that one cannot expect this to work for *any* texture: the same way than seamless pattern mapping on surfaces is well-posed for some categories of patterns (*e.g.,* isotropic) and ill-posed for others (*e.g.,* strongly oriented), invisible blending between two arbitrary regions of a texture is possible only for reasonable texture contents (namely, self-similar and isotropic at large scale).

## 2. Adapted Advection (see Figure 2)

We rely on a fluid simulation on a low resolution grid and we advect texture coordinates as well as density similarly to Stam [11, 12].

We consider a set of $N$ *layers* $\{T^i,\ i = 1..N\}$, where each layer is made of three regenerated texture parameterizations $\{(u^i_j, v^i_j),\ j = 1..3\}$. Assuming linear blending is used (we will release this in section 3) we have $T^i = \sum_{j=1}^{3} \alpha^i_j . T(u^i_j, v^i_j)$ where T() is the texture map, and $\alpha^i_j = \frac{1}{3}\left(1 - \cos\left(2\pi\frac{t - t0^i_j}{\tau^i}\right)\right)$. Each layer has its own *latency* $\tau^i$ (*i.e.,* life duration of each texture), thus is adapted to a range of velocity. The idea is to choose at each location, the layer $T^*$ that is most adapted to the local deformation. Similar to MIP-mapping, we chose and blend the two layers that most closely bracket the desired value, thus ensuring continuity in space and time.

To forge our criterion, we maintain a per-vertex measure of the accumulated deformation $d^i_j$ of each texture $j_i$ (this data has to be advected with the fluid as well). We define the accumulated deformation $d^i$ of the layer $i$ as the sum of the accumulated deformation of its three texture parameterizations $j_i$. Note that although this value increases for a texture $j_i$ according to its age, the average on the three phase-shifted textures is quasi-constant for a steady flow.

The user provides a target amount of deformation $d^*$ balancing between motion illusion quality and maximum stretching allowed. We obtain the ideal (decimal) layer number $l^*$ to be used through back-interpolation: for $i$ such that $d^i < d^*$ and $d^* < d^{i+1}$, $\quad l^* = \frac{d^* - d^i}{d^{i+1} - d^i}$.
Then, we proceed with our '*temporal tri-linear MIP-mapping*' by blending the layers $i$ and $i+1$ with weights $1 - f$ and $f$ respectively, where $f$ is the fractional part of $l^*$. Thus the final texture (assuming linear blending) is $T^* = (1-f)T^i + fT^{i+1}$ . This texture value should then be multiplied by the local density (or color) interpolated from the grid values (this is handled by OpenGL if hardware textures are used).

To ease the notations we note $\alpha'^i$ the factor of $T^i$ (*i.e.,* $f$, 1-$f$ or 0) so that we can write $\quad T^* = \sum_i \alpha'^i T^i$ .
We also note $\quad \alpha'^i_j = \alpha'^i \alpha^i_j \quad$ so that $\quad T^* = \sum_{i,j} \alpha'^i_j T(u^i_j, v^i_j)$ .
But keep in mind that we are blending 6 textures (2 layers each made of 3 regenerated textures) and not $3N$ at each vertex[IV].

[IV] We rely on per-vertex adaptation criterion to allow implementation on a simple hardware. This can occasionally lead to the combination of more than 6 textures in a given pixel. However, note that this criterion can easily be used per pixel if necessary on software and fragment program implementations.

### 2.1. Measure of the Deformation

In continuum mechanics, deformation is measured by the *strain tensor* $\varepsilon = \frac{1}{2}(G + G^t)$ where $G$ is the velocity gradient matrix $(\frac{\partial v_i}{\partial x_j})_{i,j}$ (taking the symmetric part of the gradient matrix allows us to cancel the effect of solid rotations). If a scalar measure is required, the norm $||\varepsilon|| = \sqrt{\sum_{i,j} \varepsilon^2_{i,j}}$ is used.

As stated above, for each texture $j$ of each layer $i$ we maintain the *accumulated local deformation* $d^i_j(x,y)$ since its last regeneration. This is the time integral of the instant deformation $||\varepsilon||$ for a given fluid parcel (so this data has to be advected with the fluid as well). At each time step and for each vertex we compute $||\varepsilon||(x,y)$ and update the values: $d^i_j(x,y) \mathrel{+}= \ ||\varepsilon||(x,y)dt$ .

Note that integrating a norm prevents any reverse motion to cancel a previous deformation. This case does not occur with turbulent fluids. However, if such effect was to be simultated, one would simply have to store and update the integral of the tensor itself, then to evaluate the norm to be compared with $d^*$.

| **Global:** | ($d^*$ and $\tau^i$ are user defined, others are internal) |
|---|---|
| $d^*$ | *the target amount of deformation.* |
| $\tau^i$ | *the latency of layer i.* |
| $t0^i_j$ | *the time of the last regeneration of texture $j_i$.* |
| | *(in fact we store and update $t - t0^i_j$ ).* |
| $\alpha^i_j$ | *the weight of texture $j_i$* |
| | *(stored to avoid the cosine evaluation at every vertex)* |

| **Local:** | (to be advected with the fluid) |
|---|---|
| $(u^i_j, v^i_j)$ | *the texture parameterization $j_i$.* |
| $d^i_j$ | *the local accumulated deformation of texture $j_i$.* |
| $l^*$ | *the decimal ideal layer number for at this vertex.* |

**Table 1:** *Summary of the introduced parameters*

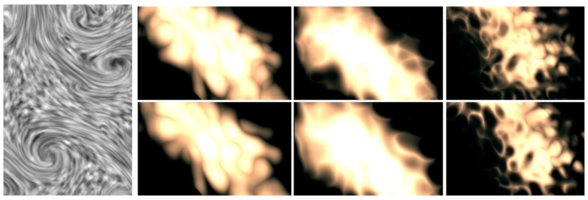


**Figure 3:** Left: *texture streched in fast regions due to low latency.* Right, top: *ghosting artifact on procedural noise with classical blending of 3 textures.* Right, bottom: *our blending without artifact.*

## 3. Blending Procedural Textures

At this stage, we have defined how to passively advect textures along the flow. This implies the blending of 6 textures (in general). As stated in the introduction, computing the simple blending of textures (*i.e.,* doing the weighted sum) produces artifacts such as ghosting effects and contrast fading: non-corresponding pattern features are super-imposed and rendered with a weight less than 1 (see Figure 3). If an explicit texture map is used, there is no easy way to do better. However, we can improve this in the case of procedural noise such as Perlin noise (or hypertextures [9] in 3D):

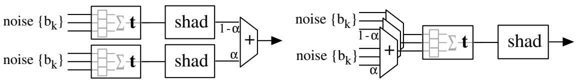



**Figure 4:** *Classical vs our blending of 2 procedural samples.*

Since pseudo-random noise base functions $b()$ share the same pseudo-period and are uncorrelated, their linear combination is a correct base function as well. So is the resulting linear turbulent[V] noise value obtained by adding the base function of harmonic frequencies. Things turn bad when a non-linear transform is used when combining the harmonic components. Typically, a nice and frequently used turbulent function is $t(x) = \sum_{k=0}^{n} \frac{1}{2^k} |2b(2^k x) - 1|$: the absolute value produces a discontinuity of the derivative that fits well with the look of fluffy clouds, choppy waves or landscapes carved with valleys. This non-linear feature will yield the same blending artifacts as mentioned above. To avoid this, we separate Perlin-based procedural shaders into two parts: all the base functions $\frac{1}{2^k} b(2^k u^i_j)$ are evaluated for each texture parameterization $u^i_j$ and blended classically so that we obtain a blended *spectrum* $\{\frac{1}{2^k} b^*_k(),\ k = 0..N\}$. Then the shader that computes the visual attribute from this spectrum is applied (including the chosen turbulent sum $t()$ and the user-defined shader $shad()$ based on this turbulence). *I.e.,* instead of computing the linear blending as defined in section 2:

$$\sum_{i,j} \alpha'^i_j shad(t^i_j()) \quad \text{with} \quad t^i_j() = \sum_{k=0}^{n} \frac{1}{2^k} f(b(2^k u^i_j))$$
$$\text{(with } f(b) = |2b-1| \text{ in the example above),}$$

we compute instead: (See Figure 4)

$$shad(t^*()) \quad \text{with}$$
$$t^*() = \sum_{k=0}^{n} \frac{1}{2^k} f(b^*_k) \quad \text{and} \quad b^*_k = \sum_{i,j} \alpha'^i_j b(2^k u^i_j)$$

Note that this scheme also saves computation since the potentially complex shader $shad()$ is now evaluated only once per pixel or voxel.

[V] A procedural texture based on Perlin noise usually consists of a *shader shad*() transforming one or several *noise* signals into color, transparency or bump. The noise signal itself is the result of a process: an interesting *turbulent* noise $t()$ is produced from a fractal combination of a simple base noise $b()$.

## 4. Animating Small Scales

Passively advecting the texture is generally not sufficient: Since the texture increases the apparent spatial resolution, we need to make this new detailed visual information swirl at this scale the same way the low resolution scales move. *I.e.,* we need to make the small scales *alive*. As stated in the introduction, this is classically achieved by adding an extra dimension to the noise so that it is evaluated both in space and time. Alas the time statistical properties of fluids are very different from their spatial statistical properties and thus poorly represented by pseudo-random noise. *Flownoise* [10] has been introduced to provide a better swirling behavior to Perlin noise. The key idea is to apply a user-defined rotation along time to the base gradient vectors of regular Perlin noise[VI]. The problem is then to tune these rotation parameters along space and scales.

Our idea is inspired by the 1941 theory of Kolmogorov [3] (after a previous remark by Richardson that "large eddies break into smaller eddies and so on to viscosity"): an *energy cascade* in the Fourier domain can be observed and quantified showing that energy travels from the large to the small scales. Thus the rotation at a small scale is related to the vorticity at a larger scale with a delay. However, the Kolmogorov cascade applies on an inertial range (*i.e.,* homogeneous distribution in space) while we want to apply it on a fluid with heterogeneous activity. We assume that the inertial range condition is valid within each grid cell, but that the fluid activity can change from cell to cell.

To model the local vorticity spectrums corresponding to the energy cascade, we store and maintain at each vertex the vorticity $\omega_i$ to be applied at each procedural scale below the grid cells size (*e.g.,* 4 values if the grid is $32^2$ or $32^3$ and the final image is $512 \times 512$). These $\omega_i$ are used as rotation parameters for the flownoise. Note that this data has to be advected with the fluid as well.

At each time step and for each vertex, we simulate the energy transfer through scales by applying the relaxation $\omega_k := \beta_k \omega_k + (1-\beta_k)\omega_{k-1}$ where $\beta_k = 2^{-\frac{dt}{\tau_k}}$ ($\tau_k$ is the characteristic delay of transfer between scale $k-1$ and $k$) and $\omega_0$ is the vorticity corresponding to the grid wavelength. Energy cascade theory suggests that $\tau_k = \gamma^k \tau_0$ , so that the cascade is defined by the two numbers $\gamma$ and $\tau_0$. But we consider that the tuning of these 4 transfer delays $\tau_k$ should be left to the user since it allows him to control the 'activity' of the fluid.

[VI] The base noise used for Perlin noise is defined by $b(x) = \sum_i \chi(x - x_i) g_i(x - x_i)$ with $g_i(\vec{d}) = \vec{g}_i.\vec{d}$ the gradient function associated to the node $i$ of the virtual cell surrounding $x$, and $\chi(\vec{d})$ a drop-off kernel. The components of the gradient vector $\vec{g}_i$ are random values.

For flownoise the base noise function applied with a rotation R(t) is defined the same way, with $g_i(\vec{d}) = ROT(\vec{g}_i, R(t)).\vec{d}$ . The rotation speed usually depends on the scale.

### 4.1. Evaluating $\omega_0$

In principle $\omega_0 = \hat{\omega}(\frac{2}{h})$ where $\hat{\omega}$ is the Fourier transform of the vorticity and $h$ the grid cell size. However, we assumed that the inertial range condition only applies within a cell, so we should measure the energy frequencies only within a neighborhood and not through the whole domain.

Our initial idea was to evaluate $\omega_0$ using a selective high pass filter on $\omega$. But we observed that numerical dissipation as well as discretization of the operators make this value unreliable. So we simply assume that $\omega_0$ is proportional to the norm of the vorticity $\omega$ at this vertex. If large scale eddies do exist in a simulation, a non-selective high pass filter could be used such as $\bar{\omega}^n - \bar{\omega}$ where $\bar{\omega}^n$ is the average of $\omega$ in a $2^n$ neighborhood and $\bar{\omega}$ is the global average.

| **Global:** | ($\beta_k$ is user defined, others are internal) |
|---|---|
| $\beta_k$ | *the transfer coefficients between scales $k-1$ and $k$ (which are procedural scales,* i.e., *sub-cell).* |

| **Local:** | (to be advected with the fluid) |
|---|---|
| $\omega_k$ | *the local vorticity for scale k.* |

**Table 2:** *Summary of the introduced parameters*

## 5. Results

We have implemented the presented workflow upon the *stable fluid* solver based on FFT [12]. We relied on 3 layers (*i.e.,* 9 textures) for all our examples. For classical texture maps or precalculated procedural noise we can take advantage of standard hardware-accelerated OpenGL, thus our implementation is real time in these cases. Note that only linear blending can then be used. For procedural textures – which requires per-pixel calculations – our software renderer requires 1 to 20 seconds per frame depending on the complexity of the shader and of the image (the ability of the newcoming graphics boards to compute complex fragment shaders should ease this task).

The various parameters our method adds to the simulation share the same low resolution as the grid (and are advected like the density field). Only the given image texture (if any) and the final images are high resolution, thus very little memory is needed. Moreover, using procedural noise allows us to predict whether material will appear or not in a grid cell. Thus the high resolution evaluations are done only in useful regions. This implies that the rendering cost grows less than linearly with respect to the number of grid vertices. Conversely, increasing the resolution of the simulated grid yields linear increase of the storage and super-linear increase of the computation time.

The animations joined to the paper (check the CDROM and our web site) illustrates the effects of advected texture with low or high constant latency, and with locally adapted latency. Some rely on image textures (the two first examples, *i.e.,* the color flow of Fig 5a and the paste of Fig 5b), while the others rely on procedural Perlin noise (the clouds layer and the fireball) using the *flownoise* extension [10].

We have generated many different kinds of images and animations as shown in the teaser image, in Figure 5, and on the video (note that the atmospheric animations do not pretend any plausibility!). The resolution grid was $64 \times 64$, but most images were taken in a $16 \times 16$ region of interest. The final resolution is generally $512 \times 512$. We also did early experiment in 3D, thus showing that our scheme applies to 3D as well: the last sequence of the video shows how our method can amplify both the shape and animation in a $8 \times 8 \times 8$ region of a 3D flow. The rendering is then the most demanding part since volumetric hypertexture rendering [9] is required (please don't pay attention to the poor rendering quality of our hardware-based volume shader prototype). Note that in 3D, vorticity is a vector. We used random orientations for this test, but this issue deserves deeper investigation.

The combination of our new parameters with the fluid parameters and the procedural texture parameters opens a very wide field of tuning for experimentation.

## 6. Conclusion

This paper has presented a complete solution for animating a texture advected by a fluid. We rely on 3 steps that were unsolved in previous approaches:

- Advecting the texture without stretching it by introducing a local adaptive scheme triggered by the accumulated local deformation.
- Blending the base textures without interpolation artifact (ghosting effect), at least for Perlin procedural texture.
- Handling animation below the size of the simulation grid relying on flownoise, and controling it to ensure coherency with the simulated fluid activity.

Our results show that visually pleasing advected texture can be obtained. This allows us to fake very high resolution fluids based on correct low resolution CFD. Moreover, the user can phenomenologically control the aspect of details similarly to the usual scheme for surfaces. A purely physical approach would have required to model and solve the small scale phenomena: both tasks are generally very difficult since natural objects (*e.g.,* foam, lava...) combine numerous phenomena (some not understood or with unknown parameter values), and require solving non-linear equations at very high resolution.

For future work, we would like to adapt the entire scheme to the new generation of graphics hardware offering fragment programmability, and to further explore the application of textured fluids to 3D such as cloud simulation. In such a case, most of the space is empty which should be taken into account for deep optimization. We are also interested in extending our blending approach to other procedural noises, *e.g.,* Worley noise [16].

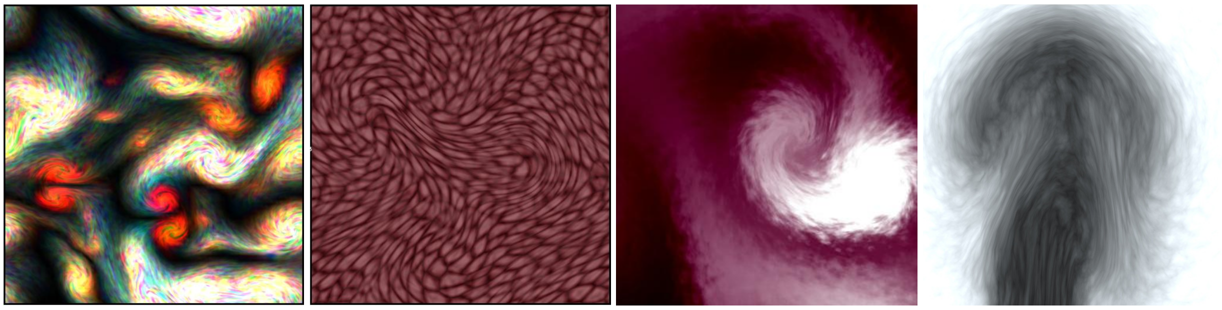


**Figure 5:** *Various kind of generated images (see also the teaser image). The two on left rely on hardware textures. The two on right are procedurally generated. Note that the maximum stretching is controlled (on the rightest it is high on purpose).*

**Acknowledgments**

We wish to thank Jos Stam for discussions on fluids and fluid deformation and François Faure for a discussion on the meaning of the strain tensor components in mechanics. Thanks are also due to Marie-Paule Cani and Laks Raghupathi for rereading and to Lionel Reveret, Jean-Dominique Gascuel and Gilles Debunne for last-minute video editing.

**References**

1. David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6.
2. David S. Ebert and Richard E. Parent. Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 357–366, August 1990.
3. A.N. Kolmogorov. The local structure of turbulence in incompressible viscous fluid for very large Reynolds number. *Doklady Akad. Nauk SSSR*, pages 9–13, 1941.
4. Arnauld Lamorlette and Nick Foster. Structural modeling of natural flames. *ACM Transactions on Graphics*, 21(3):729–735, July 2002.
5. Nelson Max and Barry Becker. Flow visualization using moving textures. In NASA Conference Publication 3321-Series, editor, *Proceedings of the ICAS/LaRC Symposium on Visualizing Time-Varying Data*, pages 77–87, 1996.
6. Ken Musgrave. Great balls of fire. Technical report, Digital Domain, 1997. http://www.kenmusgrave.com/balls_o_fire.ps.
7. PDI. 'Shrek': The story behind the screen. In *Siggraph Course Notes CD-ROM, Course #19*, pages 59–66. ACM-Press, 2001. http://terra.cs.nps.navy.mil/DistanceEducation/online.siggraph.org/2001/Courses/cd1/cnav/cnavc19.html.
8. Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.
9. Ken Perlin and Eric M. Hoffert. Hypertexture. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23(3), pages 253–262, July 1989.
10. Ken Perlin and Fabrice Neyret. Flow noise. *Siggraph Technical Sketches and Applications*, page 187, Aug 2001. http://mrl.nyu.edu/~perlin/flownoise-talk/ http://www-imagis.imag.fr/Publications/2001/PN01/.
11. Jos Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 121–128, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.
12. Jos Stam. A simple fluid solver based on the fft. *Journal of Graphics Tools*, 6(2):43–52, 2001.
13. Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. *Proceedings of SIGGRAPH 95*, pages 129–136, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.
14. Dan Stora, Pierre-Olivier Agliati, Marie-Paule Cani, Fabrice Neyret, and Jean-Dominique Gascuel. Animating lava flows. In *Graphics Interface (GI'99) Proceedings*, pages 203–210, Jun 1999. http://www-imagis.imag.fr/LAVA/.
15. G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA, 1990. ISBN 0-81868-944-7.
16. Steven P. Worley. A cellular texturing basis function. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 291–294. ACM SIGGRAPH, Addison Wesley, August 1996.