Wavelet Project Report
Wavelet Turbulence for Fluid Simulation

Alexis SONOLET, Clément MALLERET

1 Introduction

Wavelet Turbulence for Fluid Simulation is a research paper from Theodore Kim, Nils Thurey, Doug James and Markus Gross. In this paper they describe a method that is very useful for fluid simulations : using noise generated with a wavelet transform they can create new details from a low-resolution simulation and then create a higher resolution simulation that matches on average the original simulation. Therefore it is really useful to artists which now don't have to simulate a full-resolution simulation, and can work on low-resolution models.

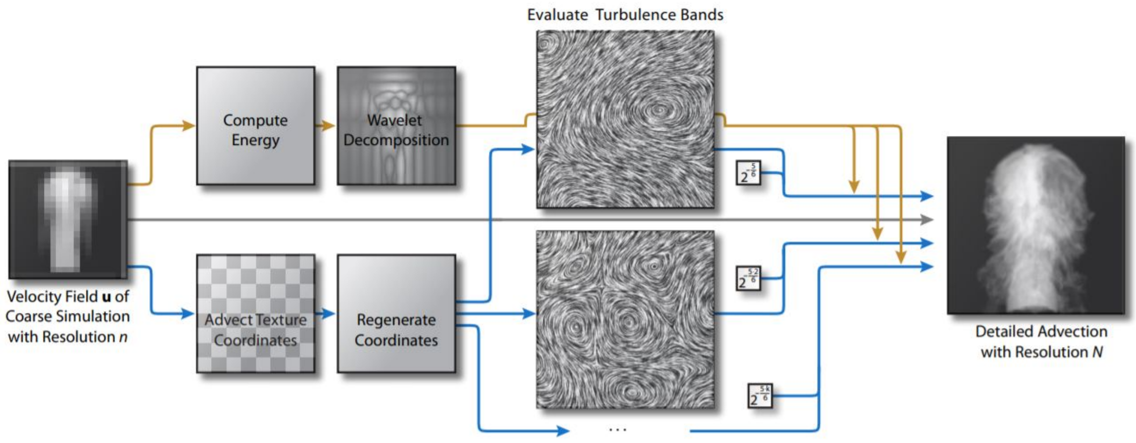
Evaluate Turbulence Bands
Compute Energy
Wavelet Decomposition
Velocity Field **u** of Coarse Simulation with Resolution *n*
Advect Texture Coordinates
Regenerate Coordinates
$2^{-\frac{5}{6}}$
$2^{-\frac{5\cdot 2}{6}}$
$2^{-\frac{5\cdot k}{6}}$
...
Detailed Advection with Resolution *N*

FIGURE 1 – Overview of the steps to upscale the simulation

2 Initial simulation

So first we had to find an initial simulation... which was way easier said than done. As a matter of facts, that may be the part where we spent much of our time.

First we decided to use the Academy Award winner OpenVDB format (first because it sounded cool, then because we found VDB files on the official website). We learned to use their API `pyopenvdb`, and we tried to read the density grid from the file we found. It worked to read the metadatas, but it failed to read the datas. We thought that the `read` function from the API was broken, so we tried to rewrite the binding in C++, but because of non-compiling libraries from OpenVDB, we couldn't do it.

Then we decided to make the fluid simulation ourselves. We then found a paper from Jos Stam [3], but the implementation was not really convincing (the paper is providing code for 2D Navier Stokes solver, but we had to transpose it to 3D). It didn't work so we decided to use a C/C++ fluid simulation library, and then export a simulation in a text file, and then open it and parse it with Python. We tried with a program found on GitHub [4], but because of non-compiling libraries (again !), it didn't work. Finally we tried to use Mantaflow [5], which worked, but we had no visualization of the simulation (the GUI on mantaflow used Qt, and it caused linkage errors), so we gave up on Mantaflow.

Finally we tried to run the simulation directly on Blender, and found out that we could get the simulated files by exploring the cache folder of Blender. So we found the VDB files, tried to open it using `pyopenvdb`, and... it worked. Probably because the files we downloaded first were generated with EmberGen and Houdini, and may have some syntax differences with thoses generated with Blender, but anyway, it worked : we had a simulation.

3 Synthesize higher resolution fluid

3.1 Interpolating

The first step to create a higher-resolution simulation is to create a base grid for it, interpolating the density and velocity fields to get the values of the intermediates voxels that will be created here. We use there a basic trilinear interpolation. This step does not inject any extra information to the simulation yet.

For that, we will actually inject new synthesized information to the velocity, and deduce the new densities by advecting the interpolated ones with thoses new velocities.

3.2 Synthesize a new velocity

3.2.1 Why doing this ?

So now let's dive in **why** exactly we are doing all of this. We are trying to upscale the fluid simulation, and we're starting with a low resolution one. In reality what "low-resolution" means, is that in the velocity field we have lacking frequencies. Let's look at some example (see Fig. 2).

In this schema, we can see a representation of the velocity field (made continuous using interpolation). As it is shown, it consists in a superposition of different grids, each one representing a different frequency. Each grid consists of a multitude of cells (that can be seen as

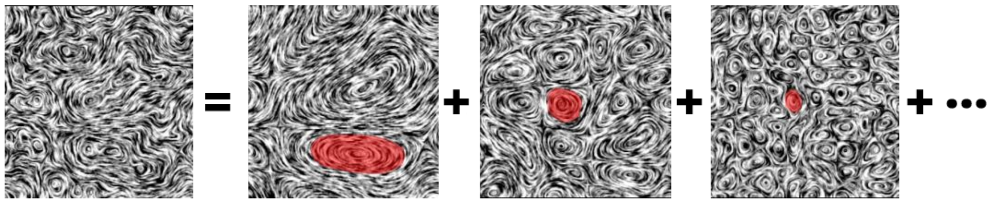

FIGURE 2 – Example of frequency decomposition

the voxels). Each grid cell energy can be computing like that :

$$e_x = \frac{1}{2}|v_x|^2$$

And we can get the total energy of the grid :

$$E = \sum_x e_x$$

Then we can draw the curve of the log of the energy in relation of the log of the frequency :

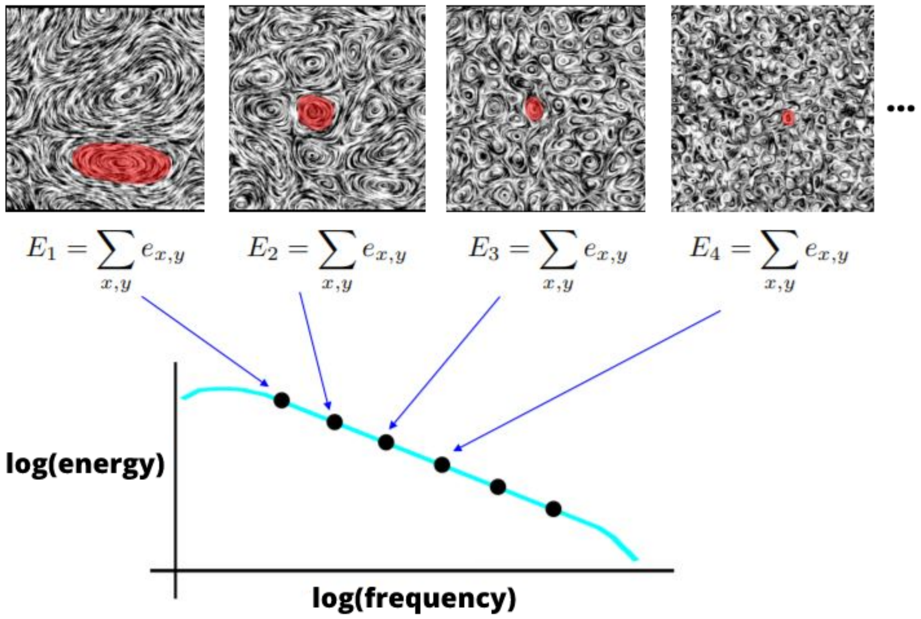
$E_1 = \sum_{x,y} e_{x,y}$
$E_2 = \sum_{x,y} e_{x,y}$
$E_3 = \sum_{x,y} e_{x,y}$
$E_4 = \sum_{x,y} e_{x,y}$
log(energy)
log(frequency)

FIGURE 3 – Contributions of the energy of each frequency

The slope of this curve has been found to be constant, and computed by Andreï Kolmogorov to be equal to $-5/3$ (more info here [6]). So why does it matter ? Because as we said earlier, after interpolating the velocities, we still lack of detail, because we lack of frequencies in the interpolated velocities (see Fig. 4).

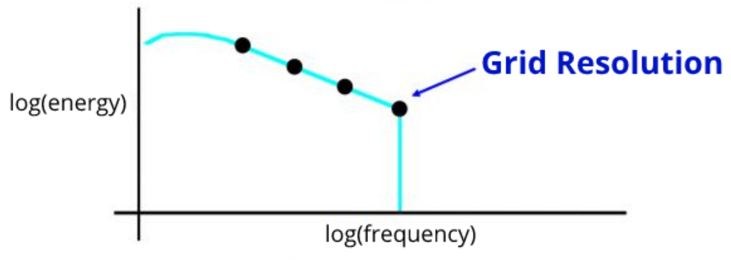

FIGURE 4 – The initial grid resolution limits the simulation resolution

So what we can do with a simulation limited in resolution is to find the first frequencies in the velocities (using wavelet transform). Then we can begin to draw this curve, and then to complete it with new generated frequencies, and use the $-5/3$ slope to match the curve.

3.2.2 Creating noise

The first step for this is to artificially create the missing part of the energy/frequency curve. For that, we will first need to create some noise, that we will use as a basis to add new information. We actually create a "noise tile", a big list of pre-computed noise values, that we can reuse as much as we want, to avoid re-computing it each time.

Creating the noise tile we use is actually... An entire different paper[7]. As we preferred to focus on this paper, we used the implementation of the author to generate a noise tile, that we then used. Note that we used wavelet noise, but Perlin noise could also have been used.

But this function is a scalar function : we want to add information to velocities, so we need a 3D vector field. Using this noise tile, noted ω, as a basis, we can construct a divergence-free 3D vector field (needed to have a physically correct simulation), with the following formula :

$$w(x) = \left(\frac{\partial \omega_1}{\partial y} - \frac{\partial \omega_2}{\partial z}, \frac{\partial \omega_3}{\partial z} - \frac{\partial \omega_1}{\partial x}, \frac{\partial \omega_2}{\partial x} - \frac{\partial \omega_3}{\partial y} \right)$$

Instead of using 3 different tiles $\omega_1, \omega_2, \omega_3$, we actually use offsets of ω, as it is just noise.

Now, we got some 3D noise. Time to use it to construct a field that follows Kolmogorov's theory of turbulence.

3.2.3 Computation of the velocity

We construct the turbulence function, y, this way :

$$y(x) = \sum_{i=i_{min}}^{i_{max}} w(2^i x) 2^{-\frac{5}{6}(i-i_{min})}$$

i_{min} and i_{max} are used to control the frequency band in which we generate the new information : indeed, we want to only add the missing frequencies, not the ones that are already there.

This turbulence will create a field whose energy follows this famous -5/3 slope. However, we miss a final part : the slope is now correct in the energy/frequency curve, but its starting point may not actually correspond : we need to weight it correctly. A too high weight will cause a discontinuity in the curve, brutally increasing at the new frequencies, and creating a lot of details in a smooth part of the fluid, while a too low weight will result in a similar discontinuity, except that the values will be brutally decreasing, and will not create details in highly-turbulent zones of the fluid.

We could first weight the turbulence globally, computing the total energy of the fluid. This would respect the energy/frequency curve : the total energy would follow the $-5/3$ slope. However, the problem of this method is that it is... global, meaning that we will add the same amount of details everywhere in the space. If the fluid is perfectly plane on a side for example, this will add small eddies to it, resulting in a poor simulation.

That's where wavelets intervene. We will weight the turbulence by the highest frequency's coefficient of the wavelet transform of the energy at the considered point of the fluid. This allows the turbulence to vary in the space : the weight will be high in points where there is high-frequency precursors, adding more details in turbulent places, and it will be low in places with no high-frequency precursors : we don't add details in smooth places of the fluid. Intuitively, we actually look at the energy/frequency curve locally, and complete it according to its local "end" due to the limitation of frequency. It allows much more fidelity in the resulting simulation, while still respecting Kolmogorov's theory.

We then interpolate (again) this wavelet transform to each point of our higher-frequency fluid. We now have everything we need to compute the final velocity :

$$v(X) = I(v, X) + 2^{-\frac{5}{6}} I(\hat{e}(x, n/2), X) y(X)$$

With :

- v : velocity
- I : Interpolation function
- $\hat{e}$: Wavelet transform of the energy
- n : the resolution of the lower-resolution grid : its highest frequency allowed by such a resolution is n/2
- X : coordinates in the higher-frequency (synthesized) grid
- x : coordinates in the lower-frequency (base) grid

3.3 Advect the densities with the new velocities

So now that we have regenerated the velocity field with new frequencies, we have to advect the density along these new velocities. Advection is the transport of a scalar (in this case) value within its environment. Here the advection will take the original interpolated densities, and move them along the newly generated velocities in order to blend in the newly created details.

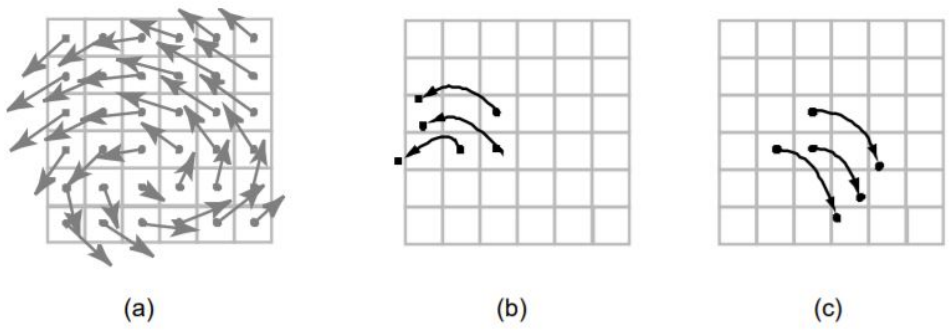

FIGURE 5 – The advection step

The advection function looks at each cell in one by one. The idea given by the article from Jos Stam [3] is that instead of moving the cell centers forward in time (b) through the velocity field shown in (a), we look for the particles which end up exactly at the cell centers by tracing backwards in time from the cell centers (c). So basically, in each cell, it takes the velocity, follows it back in time, and sees where it lands. It then takes a weighted average of the cells around the spot where it lands, then applies that value to the current cell.

The reason we trace back values in time is because if we go forward in time, we may have particles that will step out of the grid (b), but if we trace them backward in time, we are sure that the origin exists in the grid. Once the advection step is done, we're basically over. Thanks to the OpenVDB format we can export the new grid into a VDB file, and then visualize it using Blender directly. Note that we'll have to use the density to visualize the smoke. If we take an other example : drops of dye in water, we'll have to use the dye as a new grid like density, and also advect it.

4 Results

So after implementing everything it almost worked immediately. We had only one parameter to adjust : the time step (we initially set it to 0.01, but 0.1 works better). As this value worked very well we decided not to change it afterwards. But if we were to think about its influence, it only intervene in the advection step, and the bigger the time step is, the more the simulation expands with the advection step (so a little time step means that the simulation will not change a lot, while a big time step means that the densities will spread farther away from the original simulation).

Last but not least, as the paper aims to implement a **visual** simulation (i.e. it doesn't have to be physically accurate, but visually realistic), we don't have any other way to validate the results than to check it out with our eyes. So here are some of our renders :

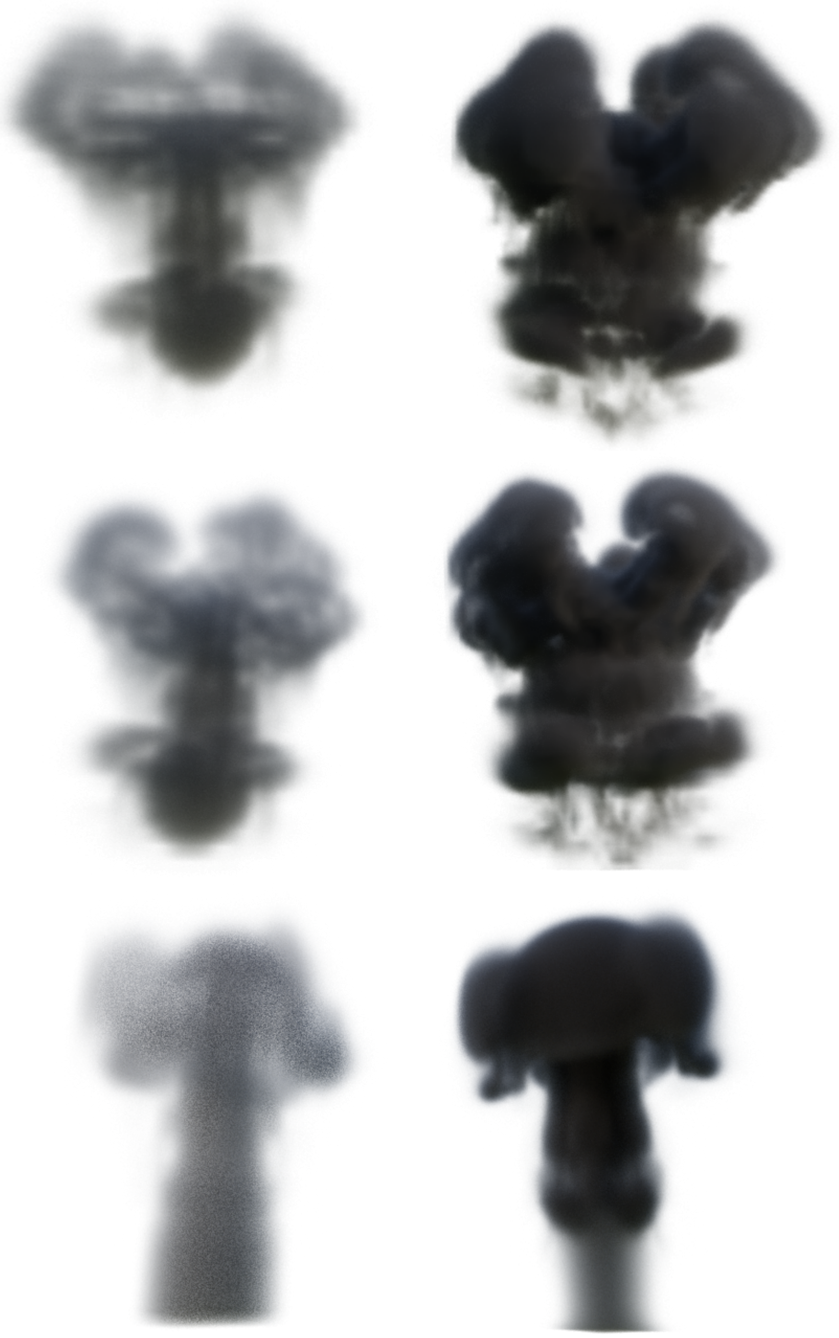

FIGURE 6 – Renders : before (left), and after (right)

We also exported a video with the two simulations side by side, in order to visualize the differences : `https://youtu.be/3StmTPVowXk`. The video was rendered in Blender 2.91 with Cycles, exported in 480p and upscaled in 720p.

5 Conclusion

So finally we managed to have a good implementation of this paper, as the visual aspects of the results are convincing. We started with a low-resolution simulation, and managed to upscale it to more than threefold ($33 \times 33 \times 33$ to $100 \times 100 \times 100$). The program is a bit long (around 1min per frame for upscaling the simulation), but we should be able to reduce the time by a hundredfold by implementing it in C. So it really is way faster than making a high-resolution simulation from the beginning.

Souce code : *https://github.com/AlexisSonolet/waveletTurbulence*

Références

[1] Wavelet Turbulence for Fluid Simulation, Theodore Kim, Doug James (Cornell University), Nils Thurey, Markus Gross (ETH Zurich)
https://www.cs.cornell.edu/~tedkim/WTURB/wavelet_turbulence.pdf

[2] Slides from the paper presentation, from the original paper authors
https://www.cs.cornell.edu/~tedkim/WTURB/wlturb_slides.pdf

[3] Real-Time Fluid Dynamics for Games, Jos Stam
https://www.dgp.toronto.edu/public_user/stam/reality/Research/pdf/GDC03.pdf

[4] Smoke Simulation program from nevermoe
https://github.com/nevermoe/SmokeSimulation

[5] Mantaflow official website : *http://mantaflow.com/index.html*

[6] The Kolmogorov Law of turbulence, Roger Lewandowski, Benoît Pinier
https://hal.archives-ouvertes.fr/hal-01244651/document

[7] COOK, R.,ANDDEROSE, T. 2005. Wavelet noise. In Proceedings of ACM SIGGRAPH
https://graphics.pixar.com/library/WaveletNoise/paper.pdf