Stable Fluids

by

Lukas Polthier, Johannes von Lindheim
based on Stam, 1999

Scientific Visualization, winter 15/16

Supervisor: Prof. Dr. Konrad Polthier

We hereby declare that this article is based on our work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged.

Berlin, 24. April 2016

Lukas Polthier
Johannes von Lindheim
Institut für Mathematik
Arnimallee 6
Freie Universität Berlin
14195 Berlin
info@lukas-polthier.de
jovoli@gmx.de

Table of Contents

0 Abstract

This article is a tutorial to implement the "Stable Fluids" solver by Stam [2] which includes a nice user interface to interactively manipulate the fluid. We provide the mathematical idea, numerical computations and the concept of the framework which we implemented in JavaView [4]. The main features of this solver are real time computation and being unconditionally stable. Moreover it allows to compute a wide range fluids with i.e. different viscosities and diffusion attributes.

1 Introduction

For many years, building realistic fluid solvers has been a challenging problem in the VFX industry.
In computer graphics, in contrast to many physical applications, the visual appearance of the fluid is very important whereas physical accuracy is not essential. In order to animate the fluid, it is required to provide real time computation. Typically this results in lower bounds on physical accuracy.

The Stable Fluids Solver by Stam [2] first gave a method that provides unconditionally stable fluids which are computed in real time. Previous solvers are unstable, i.e. they require small time steps which contradicts real time computation, or they are not based on the full Navier-Stokes equations and thus do not reflect a realistic fluid simulation.

Stams method of solution has been implemented in this article in the framework of JavaView [4]. We added the representation of colours, the functionality to import fotos and a give the user a nice framework to interactively manipulate the fluid.

The general problem is to create a realistic looking fluid simulation that can be computed in real time. This yields to constructing a stable solver for the Navier-Stokes equations.

2 Mathematical Modelling

2.1 Basic Equations

Let $\Omega \subset \mathbb{R}^2$ be the domain of interest, e.g. $\Omega = (0,1)^2$. Now let $u \in C^2(\mathbb{R} \times \Omega, \mathbb{R}^2)$ denote the velocity vector field and $p \in C^2(\mathbb{R} \times \Omega, \mathbb{R})$ denote the pressure field. u describes the velocity of the "ambient fluid", whereas p denotes the pressure of the fluid. Both fields depend on the time $t \in \mathbb{R}$ and the position in space $x \in \Omega$. The evolution of these fields is given by the Navier-Stokes equations

$$\operatorname{div} u = 0 \tag{2.1}$$

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \Delta u + f, \tag{2.2}$$

where ν, ρ are constants that determine the viscosity of the fluid and the density respectively. In $\mathbb{R}^2$, equation 2.2 can be written out as

$$\begin{pmatrix} \frac{\partial u_1}{\partial t} \\ \frac{\partial u_2}{\partial t} \end{pmatrix} = -\begin{pmatrix} u_1 \frac{\partial u_1}{\partial x_1} + u_2 \frac{\partial u_1}{\partial x_2} \\ u_1 \frac{\partial u_2}{\partial x_1} + u_2 \frac{\partial u_2}{\partial x_2} \end{pmatrix} - \frac{1}{\rho}\begin{pmatrix} \frac{\partial p}{\partial x_1} \\ \frac{\partial p}{\partial x_2} \end{pmatrix} + \nu \begin{pmatrix} \frac{\partial^2 u_1}{\partial x_1^2} + \frac{\partial^2 u_1}{\partial x_2^2} \\ \frac{\partial^2 u_2}{\partial x_1^2} + \frac{\partial^2 u_2}{\partial x_2^2} \end{pmatrix} + \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}.$$

Here, u_1, u_2 denote the first and second component of u respectively. The pressure field and the velocity field that appear in the Navier-Stokes equations, are related. In fact, by combining equation 2.1 and 2.2 we obtain a singe equation as follows.

By the Helmholtz-Hodge decomposition theorem we have that any C^2 vector field $w = u + \nabla q + \text{res}$ uniquely decomposes into a divergence-free part u, a gradient field ∇q and a residual term depending on the genus of the surface. In our case, the residual term vanishes as we operate on $\Omega \subset \mathbb{R}^2$ and Ω is contractible.

Let $P : C^2(\Omega, \mathbb{R}^2) \to \{f \in C^2(\Omega, \mathbb{R}^2), \operatorname{div} f = 0\}$ denote the projection operator onto the divergence free part. The operator P is implicitly defined by

$$\operatorname{div} w = \Delta q. \tag{2.3}$$

With Neumann boundary condition ($\frac{\partial q}{\partial n} = 0$ on $\partial\Omega$, n is the outward normal), equation 2.3 is a Poisson equation. Let q denote the solution, then P is defined by $Pw = w - \nabla q$. If we now apply P to both sides of 2.2, the Navier-Stokes equation compress into our fundamental equation 2.4. Equation 2.1 is no longer needed, as the projection ensures that the resulting field is divergence free. In particular the pressure field drops out as ∇p is a gradient field.

$$\begin{aligned} \frac{\partial u}{\partial t} &= P\left(-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \Delta u + f\right) \\ &= P\left(-(u \cdot \nabla)u + \nu \Delta u + f\right) \end{aligned} \tag{2.4}$$

2.2 Method of Solution and Discretization

Equation 2.4 consists of four parts, the **add force** term f, the **advection** term $(u \cdot \nabla)u$, the **diffusion** term $\nu\Delta u$ and the **projection** operator P. Both time and space are discretized with time step Δt and some equidistant grid points of distance $h = \frac{1}{n}$. The equation is solved from an initial state $u^0 = u(t, x)$ by marching through time with next iterate $u(t+\Delta t, x)$. Each of the four terms in equation 2.4 is applied successively to the initial state $u^0 \in C^2(\Omega, \mathbb{R})$. The general procedure is

$$u^0 \overset{\text{add force}}{\longrightarrow} u^1 \overset{\text{advect}}{\longrightarrow} u^2 \overset{\text{diffuse}}{\longrightarrow} u^3 \overset{\text{project}}{\longrightarrow} u^4.$$

The solution at time $t + \Delta t$ is then given by $u(x, t + \Delta t) = u^4(x)$.

2.2.1 Add force

The add force step incorporates additional force by the user and buoyancy force due to uplift of lighter or hotter gases, respectively downlift of heavier or cooler gases.

$$u^1(x) = u^0(x) + \Delta t \; f(t, x)$$

The buoyancy force is computed using Archimedes' principle. In a simplified approach, heaviness is equal to the density of the smoke. After computing the average temperature of the fluid, the upward force is determined for each pixel separately depending on the difference with respect to the average temperature:

foreach *Grid cell* **do**
  Cell.weight ← Sum of densities in this cell
  $f \leftarrow T_{amb} *$ Cell.weight $+ (1 - T_{amb}) *$ weight of one block of air
end

2.2.2 Advect

The advect step accounts for the advection or convection of the fluid itself, i.e. this step lets the fluid "flow" or move a little according to its own speed. The advection step is fundamental to this particular fluid solver. The design of the advect method is the reason why this method is called "Stable" Fluids, as this solver will never "blow up", independent of the size of the time step Δt.
The method can be understood intuitively: All particles in the fluid are moved by the velocity of the fluid itself. To obtain the velocity at the point x at time $t+\Delta t$ we backtrace the the point x through the velocity field at time t. This defines a path $p : (-\delta, \delta) \times \Omega \to \Omega$ corresponding to a streamline of the fluid. The velocity $u^2(x)$ is the set to be the velocity of $u^1(p(-\Delta t, x))$ at the previous time step:

$$u^2(x) = u^1(p(-\Delta t, x)).$$

Again, this step is unconditionally stable, independent of the step size.

[tba] include a figure that illustrates the approach. [tba]

In practice we backtrace each particle with some Runge-Kutta scheme, e.g. an Euler method. The we interpolate the vector at $p(-\Delta t, x)$ linearly from the values of the vector field on the four neighboring grid points.

2.2.3 Diffuse

This step solves the diffusion of the fluid itself, i.e. the "friction" between parts of the fluid with different velocity. This effect is equivalent to the diffusion equation

$$\frac{\partial u^2}{\partial t} = \nu \Delta u^2. \tag{2.5}$$

The most straightforward way would be to discretize the Laplacian and solve the resulting sparse linear system. However, this approach is unstable when the viscosity is large. For our implicit approach we proceed by approximating $\frac{\partial u}{\partial t}$ with the backward difference quotient:

$$\frac{u(t+\Delta t, x) - u(t, x)}{\Delta t} = \nu \Delta u(t - \Delta t, x)$$

Finally, this yields

$$(I - \nu \Delta t) u^3(x) = u^2(x). \tag{2.6}$$

We now discretize 2.6 using a finite difference method and obtain

$$\left(\begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix} - \begin{bmatrix} -4 & 1 & & 1 & & \\ 1 & \ddots & & & \ddots & \\ & & & & & 1 \\ 1 & & & & & \\ & \ddots & & & \ddots & 1 \\ & & 1 & & 1 & -4 \end{bmatrix} \nu \frac{\Delta t}{h^2} \right) u^3 = u^2. \tag{2.7}$$

The resulting square matrix has $n \cdot m$ rows and columns, where n, m denote the number of pixels in the x- and y-axis respectively. Solving such a system can be done efficiently by iterative schemes, e.g. Gauß-Seidel. In particular, due to the symmetric, positive definite nature of the sparse matrix, iterative schemes are very effective.

[tba] make formatting nice, i.e. bigger brackets and identity bigger [tba]

2.2.4 Project

The last step makes the vectorfield mass preserving, i.e. divergence free. We already discussed in the derivation of our fundamental equation that the projection is obtained by solving

$$\operatorname{div} u = \Delta q.$$

When discretized, this equation becomes

$$\frac{1}{2h}\begin{bmatrix} 0 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 0 \end{bmatrix} u_1^3 + \frac{1}{2h}\begin{bmatrix} 0 & & 1 & \\ & \ddots & & \ddots \\ 1 & & \ddots & 1 \\ & 1 & & 0 \end{bmatrix} u_2^3 = \begin{bmatrix} -4 & 1 & & 1 \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ 1 & & 1 & -4 \end{bmatrix} q. \tag{2.8}$$

As in equation 2.7 we need to solve a sparse linear system which can be done analogously.

2.3 Moving Substances through the Fluid

Our solver enables us to compute the ambient fluid. However, we need to visualize the vector field. A substance, that is injected in the fluid and does not interact with it, will be advected by the vector field and diffuse at the same time. Let $d \in C^2(\mathbb{R}^+ \times \Omega, \mathbb{R})$ denote the density field of such a substance. The evolution of this scalar field is given by

$$\frac{\partial d}{\partial t} = -u \cdot \nabla d + \kappa \Delta d - \alpha d + S, \tag{2.9}$$

where κ is the diffusion constant, α the dissipation term and S is a source term. The dissipation term, which will be dropped in our modelling, describes the effect that kinetic energy is converted in thermal energy. The diffusion constant determines the effect of diffusion, i.e. the effect that density "interfuses" with nearby density and the overall picture becomes "softer".

The terms in equation 2.9 are quite similar to the terms in our fundamental equation 2.4. Thus the evolution of the density field can be computed analogously to the computation of the velocity field. In particular we need to perform the advect step and the diffuse step, which correspond to the movement of the density along the velocity field and the diffusion of the density itself.

2.4 Vorticity Confinement

For a sufficient number of grid points and infinite computational power, the solver described in the previous section 2 gives the fluid a realistic behaviour, quite similar to the real life experience. In practice, computational power is limited and we have to use a relatively small number of grid points to provide real-time computation. It turns out that small scale details,

in particular rotational turbulences are lost due to numerical dissipation.

The key idea of Vorticity Confinement is to add these details artificially in a way that the properties of the fluid are preserved and the effect of numerical dissipation is damped. In a follow-up paper [3], Stam proposes this idea based on previous work of Steinhoff [5].

The vorticity of a 3d vector field u is defined as

$$\omega = \nabla \times u,$$

i.e. in our two dimensional case, we have

$$\omega = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{pmatrix} \times \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial}{\partial x_1} u_2 - \frac{\partial}{\partial x_2} u_1 \end{pmatrix}.$$

We identify ω with a 1d scalar field on Ω. The vorticity measures the amount of small scale detail of the fluid. The normalized gradient of the absolute vorticity

$$N = \frac{\nabla|\omega|}{|\nabla|\omega||}$$

is a vector field on Ω that points from regions of low vorticity to regions of high vorticity. Finally the small scale detail is added as follows

$$f = \varepsilon h(N \times \omega) = \begin{pmatrix} N_1 \\ N_2 \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ \omega \end{pmatrix} = \begin{pmatrix} N_2 \omega \\ -N_1 \omega \\ 0 \end{pmatrix}. \tag{2.10}$$

Again in equation 2.10 we identify 2d vector fields with 3d vector fields that are constant in the with respect to the direction of the x_3-axis. Here $\varepsilon > 0$ is a factor that determines the effect of vorticity confinement. The dependence on h gives that small scale detail is added proportional to the grid size, i.e. to the numerical dissipation. Hence the convergence of the solver is preserved.
In fact the construction provides that small scale detail is added, exactly where it is needed. The reader can convince himself of the effectiveness in figure 2.1.

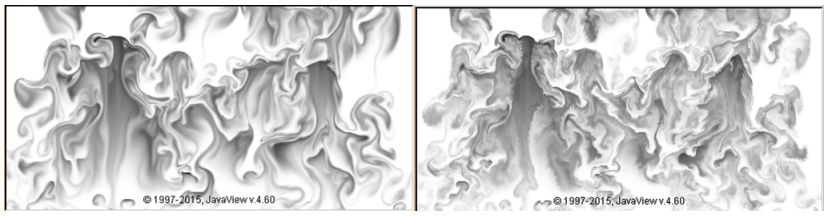


Figure 2.1: The left picture shows the fluid solver without vorticity confinement, whereas the right picture shows the same setting with medium vorticity confinement. Both pictures where computed with the same grid size and initial configuration. One can see clearly how the left picture misses small scale rotational turbulences. The right picture provides a more gaseous behaviour of the fluid.

2.5 Dependence on the Parameters

The solver is compatible with a wide range of parameters. Thus we can simulate different kinds of fluids, depending on viscosity, buoyancy force, diffusion, etc.

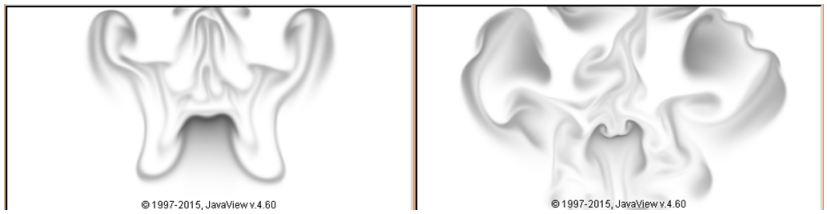


Figure 2.2: The left picture shows a fluid with high viscosity, e.g. honey or very thick water. The right picture shows a more liquid fluid like thin water or some gas. The left picture preser [tba] end the sentence [tba] Both pictures where created with identical initial condition and vary only in the viscosity constant. In both pictures there is no vorticity confinement to emphasize the effect of viscosity.

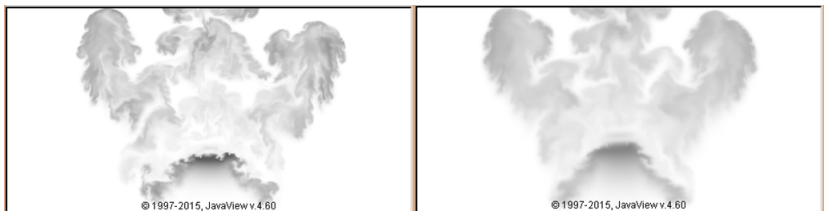


Figure 2.3: In the left picture there is no diffusion, whereas in the right picture we can clearly see how diffusion influences the nature of the fluid. There are less details and the picture looks overall smoother. Again both simulations vary only in the effect of diffusion and are computed with the same resolution.

2.6 Colors and Import of Images

There are several ways to represent colors. One of the most common priciples are RGB or CMY color schemes. In the black and white solver there is only one density array which evolves according to the velocity field. To represent colors, we use three separate density fields, each one of them representing either cyan, magenta or yellow and evolving according to the same velocity field, i.e. the same "ambient fluid".

We are now able to import picture, which behave like liquid or smoke, i.e. the picture can be manipulated interactively and behaves like the ambient fluid.
One can experiment with different kinds of buoyancy forces, e.g. only specific colors generate an uplift where some colors are heavier that others.

3 Implementation and Framework

We implemented our whole framework in the environment of JavaView. In the following, we want to show some features of our application, how they work and why we think they make sense for visualizing smoke.

3.1 Capabilities of JavaView

We want to give a brief overview over the features of JavaView that were used in the development of the project and also how they were used.

- To be able to show an image visualizing the smoke in the application, we use the common `PvDisplayIf` class. One can fix the camera over an x-y-coordinate system by selecting the correct camera: `m_disp.selectCamera(PvCameraIf.CAMERA_ORTHO_XY)`. Then one can set the image one is manipulating as the background image by `m_disp.setBackgroundImage(m_image)`.
- For handling mouse input, there are the two functions `pickInitial(PvPickEvent pickEvent)` and `dragCamera(PvCameraEvent cameraEvent)` which are invoked by JavaView, when the left resp. right mouse button is clicked. These methods were overwritten to add smoke resp. force. The mouse coordinates can be obtained from the corresponding events in the parameter list.
- The parameters of the fluid solver (the "mathematics object") and block size (see 3.3) are controlled by sliders in the application. One is able to add these for types like `PuInteger` or `PuDouble` by for instance adding `m_Project.m_vorticityConf` to some `PsPanel` in the application.
- Time can be handled by an animation, i.e. an instance of the class `PsAnimation`. Registering this animation in the project class' superclass makes JavaView invoke the method `setTime(PsTimeEvent timeEvent)` which was overwritten to call our `computeImage()`-function. To overcome the problem, that every animation has a maximum time, we are increasing the maximum by one timestep for every call of `setTime`.
- When the user drags though the display with the left mouse button, the corresponding `PvPickEvent`s yield some loose points in the canvas. The class `PgBezierCurve` is used to connect them with spline curves, such that there is not only some bumps, but a whole continuous trace of smoke added in the cancas, like one would expect for instance from a pen. For details, see 3.2.

3.2 Splines

As mentioned above, the user is able to "draw" smoke with the left mouse button like with a pen, even though JavaView just provides a handful of input points. If we just would add smoke there, just some single pixels would be provided with some smoke.

Figure 3.1: Some smoke drawn into the display by the user with the mouse.

To be able to achieve this, we connected the mouse input points from between two frames with a spline curve consisting of mostly cubic bezier curves or lower degree at the start or end where less information is available about the continuation of the curve. The resulting spline curve is C^1-continuous.

We also show the formula for the control points in the general case of cubic curves.

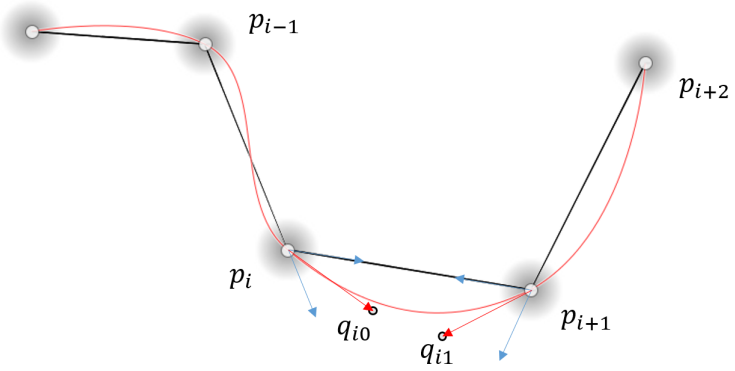


Figure 3.2: Scheme how control points are computed.

Then we compute the control points of the connecting cubic bezier curve to be

$$q_{i0} = p_i + \alpha|p_i - p_{i+1}| \cdot ((p_i - p_{i-1})^\star + (p_{i+1} - p_i)^\star)^\star$$
$$q_{i1} = p_{i+1} + \alpha|p_i - p_{i+1}| \cdot ((p_i - p_{i+1})^\star + (p_{i+1} - p_{i+2})^\star)^\star$$

where $p^\star$ is a notation for $\frac{p}{|p|}$. In our case we chose $\alpha = \frac{2}{5}$.

3.3 Block Size

Since for a large enough canvas sizes interaction with the smoke in real time is not possible because of the large amount of computations, the user should be able to choose lower resolutions on the same canvas size to make the application run faster. The corresponding (larger) pixels are called *blocks*.

If the user wants to change the size of a block, the color value of the new blocks are taken to be the average of the previous color of all pixels, that lie in that block.

3.4 Code Files

[tba]

3.5 Framework Structure

We now want to give a (brief and slightly simplified) overview over the structure of the implementation of the framework.

Every time when `setTime()` is invoked (when the animation has moved forward in time), the thread dives into the heart of the program. The structure there is as follows:

```
setTime()
    computeImage()
        Add Input Force
        Add Input Density
        if isFrozen == false then
            Solve ambient fluid
            Solve densities
        end
        while incorrecSizes() do
            wait()
        end
        Copy solution into density array of canvas
        computeColors()
    end
end
```

Since JavaView runs multithreaded, the user could possibly change the BlockSize or the size of the canvas, while a calculation of the Fluid Solver is running. But then the dimensions of the Fluid Solver and the display's density array do not match, so as seen above, the thread first has to `wait`.
Eventually a `componentResized`-event or the `update()`-method of the `blockSize`-variable is triggered. In the first case, the `Painter`'s procedure `resizeImage()` is invoked or in the latter

case, `update()` in turn calls `changeBlockSize()`. Both adjust the dimensions of the Fluid Solver (see below).
Only now can we go on in `computeImage()` and copy the solution of the differential equations into the own density array of the size of the canvas. The two adjustment functions look like this:

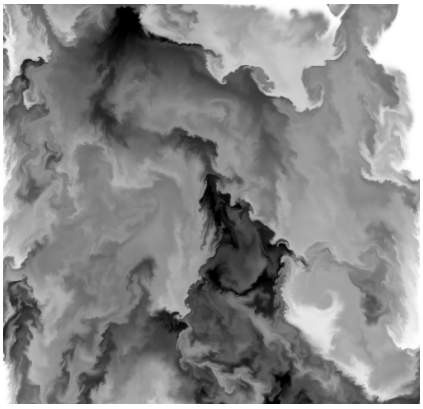
```
resizeImage()
    if this.size == display.size then
        this.size ← display.size
        resolution ← this.size/blockSize
        Resize FluidSolver
        Resize density arrays
        Resize images
        Initialize and start animation
    end
    notifyAll()
```

Note the very important call of `notifyAll()`, which is the sign for the possible other thread sitting in `computeImage()` to stop `wait`ing and further process the calculation results of the Fluid Solver.
The `changeBlockSize()`-function also works rather straighforward:

```
changeBlockSize()
    Copy current FluidSolver
    Resize FluidSolver
    foreach Block of FluidSolver do
        Store average of pixels lying in this Block
    end
    notifyAll()
```

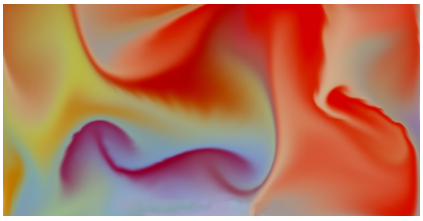
4 Results

Our solver could be used for many purposes requiring smoke-like motions, e.g. in graphic design or computer games. It can also be used to create purely artistical effects. Some possitbilities generated with the solver are shown in the following.

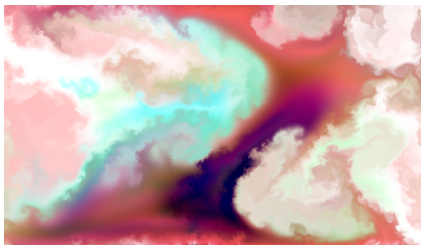


(a) Smoke generated with a single density array.

(b) Colored smoke generated with three density arrays for the CMY-colors



(c) Fluid with high viscosity.



(d) Still standing fluid, which has been diffusing for a while.

(e) Distorted Marilyn Monroe.

(f) Distorted pineapple.

5 Possible Extensions

All in all the fluid solver provides a very realistic simulation which can be used to simulate various phenomena and fluids. In particular the extension with Vorticity Confinement results in an overall realistic and intuitive behaviour of the fluid.

Possible extensions are listed as follows:
As mentioned in chapter 2, the solver can be implemented in three dimensions without further adaption of the mathematical model. This gives interesting possible extensions e.g. light sources which makes it possible to animate more complex phenomena such as clouds that are based mainly on the appearance of light and darkness. However due to the increase in grid points and more entries in the sparse matrices, we are not able to compute a 3d version in real time. Also a 3d implementation requires a renderer and the interactive user interface becomes more complicated.

Our implementation uses Gauss-Seidel to solve the sparse linear systems. While this is easy to implement and relatively fast for the sparse matrices we consider, it does not reach the speed of multigrid methods.

6 Outlook

Bezier curves
Reference List
Blocksize
Capabilities of Javaview
Details vom Solver, Wie funktioniert ein Schritt genau? e.g. Beschreibe den Advect-Schritt detailliert (Pseudocode)
Physikalische Beschreibung der Navier-Stokes
Genaue Formel der buoyancy force
Bewertung der Results -¿ Was ist gut, was ist schlecht
Anleitung um hübsche Bilder zu generieren, Tutorial um den Solver zu verwenden
Verlgleich der einzelnen Schritte vom Solver mit alternativen Lösungsansätzen -¿ Warum ist der Solver STABLE
Darstellung der Farben mit Density-Arrays
Gauß-Seidel
Boundary Conditions
Abstract (Zusammenfassung)
Introduction (Einlietung, Historie.

Reference List

[1] H. Alt: “Lineare Funktionalanalysis”, fifth edition, Springer, 2006.

[2] J. Stam: “Stable Fluids”, Siggraph, 1999.

[3] R. Fedkiw, J. Stam, H. W. Jensen: Visual Simulation of Smoke, Siggraph, 2001.

[4] K. Polthier: JavaView, program refer to javaview.de.

[5] J. Steinhoff, D. Underhill. Modification of the euler equations for “vorticity confinement”: Application to the computation of interacting vortex rings. Physics of Fluids, 6(8):2738– 2744, 1994.

[6] J. Stoer: “Numerische Mathematik 1”, ninth edition, Springer, 2005.

[7] J. Stoer, R. Bulirsch: “Numerische Mathematik 2”, fifth edition, Springer, 2005.