Fluid Flow Animation

Jakov Fustin, Zeljka Mihajlovic
University of Zagreb, Faculty of Electrical Engineering and Computing, Unska 3, 10000 Zagreb
Department of Electronics, Microelectronics, Computer and Intelligent Systems,
E-mail: jakov.fustin@zg.t-com.hr, zeljka.mihajlovic@fer.hr
phone: (+385) 1 6129 521; fax: (+385) 1 6129 653

Abstract – In this paper we briefly outline theoretical fundamentals of fluid dynamics as well as present a model for computer based simulation. Of special interest are target driven smoke simulations which don't require expert knowledge in order to achieve desired results. One of such methods was presented and implemented in two dimensions using programming language C++. The results obtained are presented alongside parameters that affect the progress of simulation.

Keywords: fluid dynamics, target-driven animation

I. INTRODUCTION

We are all witnessing the appearance of more and more special effects in movies and video games. They add to overall impression and make us talk about them for days. Sometimes, they are put up just to compensate for the lack of story and/or good acting, but that's not the subject here. What matters to us is what happens behind the scene and how these effects are created.

Of all special effects, the most challenging are natural phenomena, such as fluids in motion (e.g. smoke and water). A lot of work has been done to create realistic animations of fluids. The first models were operating with particles [1] and suffered from not being physically accurate. Navier-Stokes equations, which present a physical model for fluid flow, came into use later when Chen et al. [2] implemented them in two dimensions to animate water surface. Foster and Metaxas [3] went further to use the full Navier-Stokes equations in three dimensions and obtained many effects that were hard to key frame automatically. Their solver is explicit and becomes unstable for large time steps. Stam [4] introduced an unconditionally stable algorithm based on an implicit solver which lets users add smoke into the system and apply forces without fear of simulation blowing up. More recently, methods for controlling fluid flow have been developed. Treuille et al. [5] introduced a technique that causes a smoke simulation to optimally approximate a set of user specified key frames. While it produces good results, it is also computationally demanding and thus slow. Fattal and Lischinski [6] presented a method that does not guarantee the key frames to be optimally approximated but also demands less processor power.

The rest of this paper gives a brief introduction to fluid dynamics (Section 2) and presents Stam's model (Section 3) and later Fattal and Lischinski's method (Section 4) for simulation of fluid flow on computer. In Section 5 we describe our implementation of target-driven smoke solver and finally, in Section 6, we present results obtained by it.

II. FLUID DYNAMICS

Fluid mechanics is the branch of physics that studies fluids. It can be subdivided into fluid statics and fluid dynamics. The former concerns itself with fluids at rest and the latter studies fluids in motion. Since this paper talks about animating fluid flow, fluid dynamics will be of primary interest.

2.1 Euler and Navier-Stokes equations

The state of any moving fluid in 3-D space is completely determined with five physical quantities: three components of velocity and two thermodynamic quantities such as density and pressure. All these quantities are functions of coordinates and time.

$$\mathbf{u} = u_x(\mathbf{x},t)\mathbf{i} + u_y(\mathbf{x},t)\mathbf{j} + u_z(\mathbf{x},t)\mathbf{k}$$
$$p = p(\mathbf{x},t)$$
$$\rho = \rho(\mathbf{x},t)$$

To find out the state of a fluid at some time *t*, we must compute all of the five quantities using physical laws known as conservation of mass, conservation of momentum and conservation of energy. After combining these three laws, we get Euler equations for incompressible inviscid fluid flow

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{a} \qquad \textbf{(1)}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u}\cdot\nabla)\rho$$

$$\nabla\cdot\mathbf{u} = 0. \qquad \textbf{(2)}$$

Right hand side terms in (1) account for advection, hydrostatic pressure and external forces respectively, while (2) conserves the mass. To account for viscosity, additional considerations have to be made, the result of which are Navier-Stokes equations for incompressible homogeneous ($\partial\rho/\partial t = 0$) viscous fluid flow

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2\mathbf{u} + \mathbf{a} \qquad \textbf{(3)}$$

$$\nabla\cdot\mathbf{u} = 0.$$

The new term on the right hand side of (3) is $\frac{\mu}{\rho}\nabla^2\mathbf{u}$, where μ represents dynamic viscosity.

III. COMPUTATIONAL MODEL

Simulation of fluid flow on computer comes down to numerical integration of Euler (or, as in this paper, Navier-Stokes) equations. First, we must write them in a form that is suitable for computer implementation and then make discretizations along time and space axes.

3.1 Helmholtz-Hodge decomposition

Using Helmholtz-Hodge decomposition on (2) and (3), we obtain a single equation for the velocity

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}\left(-(\mathbf{u}\cdot\nabla)\mathbf{u} + \frac{\mu}{\rho}\nabla^2\mathbf{u} + \mathbf{a}\right), \tag{4}$$

where **P** represents an operator that projects a vector field onto its divergence free part (therefore, condition $\nabla\cdot\mathbf{u}=0$ is implicit). This form will be used in computer implementation.

3.2 Space and time discretization

The space is divided into equally sized parallelepipeds that are called cells. State variables can be positioned inside these cells in two ways, each of which has its advantages.

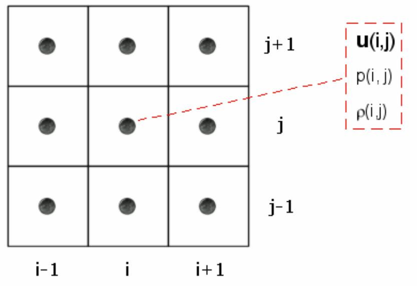


Fig. 1. All quantities defined at cell centers.

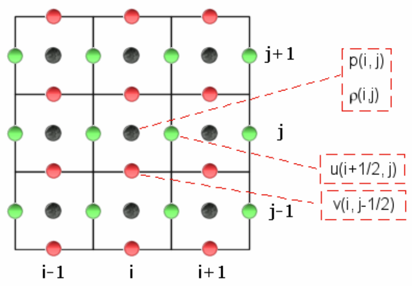


Fig. 2. A staggered grid.

The first way is to define all quantities at the center of each cell (easier to implement, but susceptible to checkerboard instability). The other way is to define velocity variables at the centers of cell faces and leave only pressure and density at the center of each cell (avoids checkerboard instability). Since the space is no longer continuous, operator ∇ changes from $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$ to $\nabla = (\Delta/\Delta x, \Delta/\Delta y, \Delta/\Delta z)$.

Finally, (4) is written in the following form to account for time discretization

$$\frac{\mathbf{u}^* - \mathbf{u}}{\Delta t} = \mathbf{P}\left(-(\mathbf{u}\cdot\nabla)\mathbf{u} + \frac{\mu}{\rho}\nabla^2\mathbf{u} + \mathbf{a}\right), \tag{5}$$

where **u** represents velocity at some time *t*, and **u*** represents velocity at *t*+Δ*t*.

3.3 Moving substances

Non-reactive substances moving through the fluid (e.g. a drop of ink in the water) are advected by it and satisfy the following equation

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u}\cdot\nabla)\rho + \kappa_\rho\nabla^2\rho - \alpha_\rho\rho + S_\rho, \tag{6}$$

where κ_ρ is diffusion constant, α_ρ is dissipation rate and S_ρ represents external sources. Like (4), this equation too has to be discretized over time and space.

3.4 Solving Navier-Stokes equations

Since the similarity of (4) and (6) is obvious, same methods can be used to find their solution. Stam describes this in detail in [4] so we'll skip it here.

IV. TARGET-DRIVEN SIMULATIONS

While the model described above provides the means for a user to affect the progress of a simulation (by modifying term **a** in (4) and term S_ρ in (6)), it is very hard to achieve desired results. This is due to nonlinearity of (4) and numerical dissipation. It would be nice if user only had to specify the desired state and the simulation evolved to it spontaneously. Such methods are crucial for animators because they let them create convincing special effects not seen before.

4.1 Modified Navier-Stokes equations

This paper focuses on the method developed by Fattal and Lischinski [6]. They make a few modifications to (5):

$$\frac{\mathbf{u}^* - \mathbf{u}}{\Delta t} = \mathbf{P}\left(-(\mathbf{u}\cdot\nabla)\mathbf{u} + v_f\mathbf{F}(\rho,\rho^*) - v_d\mathbf{u}\right), \tag{7}$$

where $\rho = \rho(\mathbf{x},t)$ is density of the substance moving through the fluid and $\rho^* = \rho^*(\mathbf{x})$ is the desired density. Term **a** is replaced by $v_f\mathbf{F}(\rho,\rho^*)$ which is a driving force, i.e. it drives the substance to the desired state ρ^* and:

$$\mathbf{F}(\rho, \rho^*) = \rho' \frac{\nabla \rho'^*}{\rho'^*}. \quad \textbf{(8)}$$

The term $-v_d \mathbf{u}$ attenuates momentum. Equation (6) is modified as follows:

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla) \rho + v_g \mathbf{G}(\rho, \rho^*), \quad \textbf{(9)}$$

where $v_g \mathbf{G}(\rho, \rho^*)$ is a smoke gathering term and

$$\mathbf{G}(\rho, \rho^*) = \nabla \cdot \left[\rho \rho'^* \nabla (\rho - \rho^*) \right]. \quad \textbf{(10)}$$

More information about ρ', ρ'*, $\mathbf{F}(\rho, \rho^*)$ and $\mathbf{G}(\rho, \rho^*)$ can be found in [6]. All of the three new terms are controlled by nonnegative parameters v_f, v_d and v_g.

4.2 Solving modified Navier-Stokes equations

Method of solution is similar to the one described in [4] and can be found in [6]. The main difference is in the approach to solve the advection term $-(\mathbf{u} \cdot \nabla)\mathbf{u}$. Stam uses the unconditionally stable semi-Lagrangian scheme, while Fattal and Lischinski use the second order hyperbolic solver based on Lax-Wendroff formula which is subject to a time step restriction.

V. IMPLEMENTATION

We implemented the method described in previous chapter in C++ programming language. Some fragments from Stam's code were used, but for the most part, the program was written from the scratch.

5.1 Data structures and algorithms

The base of the program is TDSmokeSolver class that solves (7) and discretized form of (9). An excerpt from the header file follows:

```
class TDSmokeSolver
{
  protected:
    // state variables
    std::vector <double> u, v, r, p;
    // ro star, ro tick, ro tick star, ro gather
    std::vector <double> rs, rt, rts, rg;
    // simulation parameters
    double vf, vd, vg, sigma;

    // six small steps that make one big step
    void applyDrivingForce ( );
    void attenuateMomentum ( );
    void advectMomentum ( );
    void project ( );
    void advectSmoke ( );
    void gatherSmoke ( );

public:
    // a big step
    void MakeStep ( );
};
```

Since this is a 2-D solver, there are only four (instead of five) state variables: u, v, r and p. Simulation parameters are denoted by vf, vd, vg and sigma variables. The key method here is MakeStep that computes state variables at a time $t+\Delta t$. A step consists of six smaller steps as shown below:

```
void TDSmokeSolver::MakeStep ( )
{
  void applyDrivingForce ( );
  void attenuateMomentum ( );
  void advectMomentum ( );
  void project ( );
  void advectSmoke ( );
  void gatherSmoke ( );
}
```

Before we describe every element of MakeStep method, a few things have to be said. We used a staggered grid, with variables positioned as shown in fig. 2. On this grid, discrete derivative and average operators are defined as follows:

$$D_x(u)_{i,j} = \frac{\left(u_{i+1/2,j} - u_{i-1/2,j}\right)}{\Delta x}$$

$$D_{xx}(u)_{i+1/2,j} = \frac{\left(u_{i+3/2,j} - 2u_{i+1/2,j} + u_{i-1/2,j}\right)}{(\Delta x)^2}$$

$$A_y(\rho)_{i,j+1/2} = \frac{\left(\rho_{i,j+1} + \rho_{i,j}\right)}{2}.$$

Now we'll show the pseudocode for the first step, which applies the driving force using (8). It is pretty much straightforward, so no explanations other than that already present in [6] are necessary.

```
void TDSmokeSolver::applyDrivingForce ( )
{
  // Fu    -> horizontal component of the force
  // D_rts -> derivative of ro tick star
  // A_rt  -> average of ro tick
  // A_rts -> average of ro tick star

  // horizontal component - u
  for ( every_cell u[i,j] )
  {
    double Fu, D_rts, A_rt, A_rts;

    D_rts = ( rts[i,j] - rts[i-1,j] ) / delta_x;
    A_rt  = ( rt [i,j] + rt [i-1,j] ) / 2.;
    A_rts = ( rts[i,j] + rts[i-1,j] ) / 2.;
    Fu = A_rt * D_rts / A_rts;

    u[i,j] += dt * vf * Fu;
  }

  // the same goes here for vertical component
}
```

The second step attenuates the momentum through $-v_d \mathbf{u}$ term and is the easiest to implement.

```
void TDSmokeSolver::attenuateMomentum ( )
{
  // horizontal component - u
  for ( every_cell u[i,j] )
    u[i,j] -= dt * vd * u[i,j];

  // vertical component - v
  for ( every_cell v[i,j] )
    v[i,j] -= dt * vd * v[i,j];
}
```

The advection term $-(\mathbf{u}\cdot\nabla)\mathbf{u}$ is solved using second order hyperbolic solver. Details for implementing such solver can be found in Appendix A of [6]. Protected methods hyperbolicSolveU and hyberbolicSolveV contain our implementation of the solver for x and y axis respectively.

```
void TDSmokeSolver::advectMomentum ( )
{
  // horizontal component - u
  for ( every_cell u[i,j] )
    u[i,j] -= dt * hyperbolicSolveU(…) / delta_x;

  // vertical component - v
  for ( every_cell v[i,j] )
    v[i,j] -= dt * hyperbolicSolveV(…) / delta_y;
}
```

The projection is done differently for inner and boundary cells because it depends on cell's neighbors. While all inner cells have four neighbors, boundary cells have three or even only two neighbors.

```
void TDSmokeSolver::project ( )
{
  // this term appears in all three cases below
  for ( every_cell div[i,j] )
    div[i,j] = ( u[i+1,j] - u[i,j] +
                   v[i,j+1] - v[i,j] ) * delta;

  // compute pressure
  for ( int k=0; k<20; ++k )
    for ( every_cell p[i,j] )
    {
      // cells with two neighbors
      if ( bottom_left_cell )
        p[i,j] = ( p[i+1,j] + p[i,j+1]
                               - div[i,j] ) / 2.;
      else if ( upper_left_cell )
      ...

      // cells with three neighbors
      else if ( left_boundary )
        p[i,j] = ( p[i+1,j] + p[i,j+1] +
                      p[i,j-1] - div[i,j] ) / 3.;
      else if ( right_boundary )
      ...

      // cells with four neighbors
      else p[i,j] = ( p[i+1,j] + p[i-1,j] +
           p[i,j+1] + p[i,j-1] - div[i,j] ) / 4.;
    }

  // subtract gradient of pressure from velocity
  for ( every_cell u[i,j] )
    u[i,j] -= ( p[i,j] - p[i-1,j] ) / delta_x;

  for ( every_cell v[i,j] )
    v[i,j] -= ( p[i,j] - p[i,j-1] ) / delta_y;
}
```

The term $-(\mathbf{u}\cdot\nabla)\rho$ is solved using second order hyperbolic solver. This step is similar to advection of momentum.

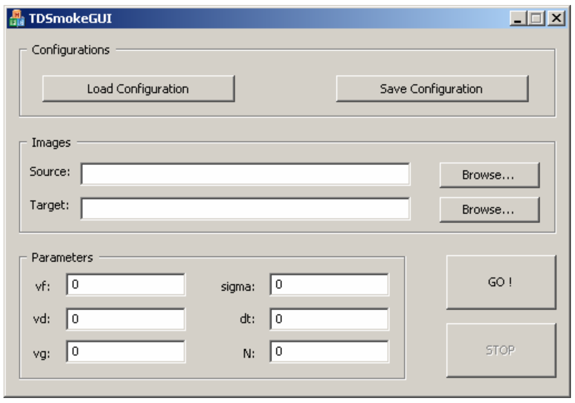
```
void TDSmokeSolver::advectSmoke ( )
{
  for ( every_cell r[i,j] )
    r[i,j] -= dt * hyperbolicSolveU(…) / delta_x
            + dt * hyperbolicSolveV(…) / delta_y;
}
```

Smoke gathering is done in five steps. The first four steps pertain to (10) and solve it starting from inside. The last step pertains to (9).

```
void TDSmokeSolver::gatherSmoke ( )
{
  // step one
  for ( every_cell rg[i,j] )
    rg[i,j] = r[i,j] - rs[i,j];

  // step two
  for ( every_cell rg[i,j] )
    rg[i,j] += dt * hyperbolicSolveU(…) / delta_x
            + dt * hyperbolicSolveV(…) / delta_y;

  // step three
  for ( every_cell rg[i,j] )
    rg[i,j] *= r[i,j] * rts[i,j];

  // step four
  for ( every_cell rg[i,j] )
    rg[i,j] += dt * hyperbolicSolveU(…) / delta_x
            + dt * hyperbolicSolveV(…) / delta_y;

  // step five
  for ( every_cell r[i,j] )
    r[i,j] += vg * rg[i,j];
}
```

5.2 Graphical user interface

Graphical user interface was created with MFC (Microsoft Foundation Classes). It is pretty simple and easy to use.



vf: 0
vd: 0
vg: 0
sigma: 0
dt: 0
N: 0

Fig. 3. The main window.

The main window is divided into four logical sections: Configurations, Images, Parameters and Actions. The Configurations section provides user with the options to save current and/or load previously saved configuration. Both actions are done through a Windows Common File Dialog. In the Images section, user can enter paths to source and destination images, both of which must be raw with one byte per pixel. Parameters can be easily adjusted using controls in Parameters section and, finally, buttons GO and STOP start and stop the simulation respectively.

Fig. 4. A window showing the substance in the fluid.

Once the simulation is started, every control except STOP button is grayed out until STOP is pressed. Another window pops up, in which the substance moving through the fluid is shown.

VI. RESULTS

Let us now present the results obtained by our implementation of the fluid solver.

6.1 A few examples

The first simulation we ran is the one also used in [3] in which Greek letter psi tries to transform into omega. Simulation parameters are $v_f = 1.5$, $v_d = 3$, $v_g = 0$, $\sigma = 2.5$. Size of the grid is 64 x 64 and time step equals $dt = 0.005$.

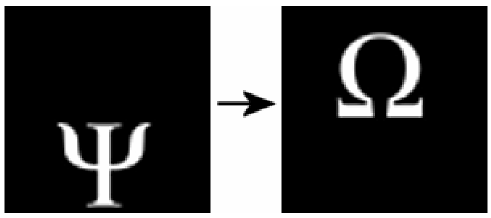


Fig. 5. Substance at the beginning of simulation (psi) and the desired state (omega).

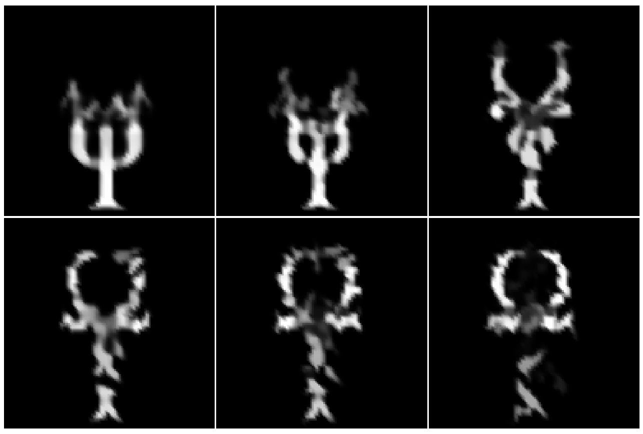


Fig. 6. Substance evolving to the desired state.

It is obvious that the target is not optimally matched, which is ok since this method doesn't guarantee that will happen anyway. Next we moved on to larger grids (128 x 128) and tried to transform a cross into a tick. Simulation parameters are $v_f = 1.25$, $v_d = 2.5$, $v_g = 0$, $\sigma = 1$ and time step equals $dt = 0.005$.

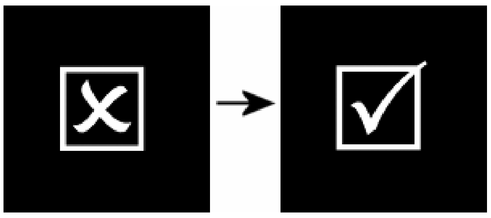


Fig. 7. A cross and a tick.

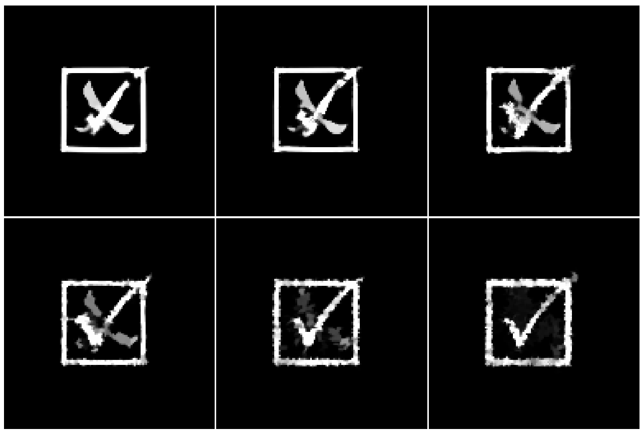


Fig. 8. A cross evolving into a tick.

Compared to fig. 6, the substance in fig. 8 looks finer. This is due to the four times larger grid. The larger the grid, the finer the result and, unfortunately, the less stable the simulation. The last example features letters FER evolving into FER logo on a 128 x 128 grid. Simulation parameters are $v_f = 20$, $v_d = 6.5$, $v_g = 5 \cdot 10^{-5}$, $\sigma = 1$ and time step equals $dt = 0.0002$.

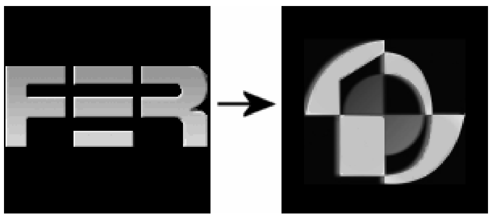


Fig. 9. FER and FER logo.

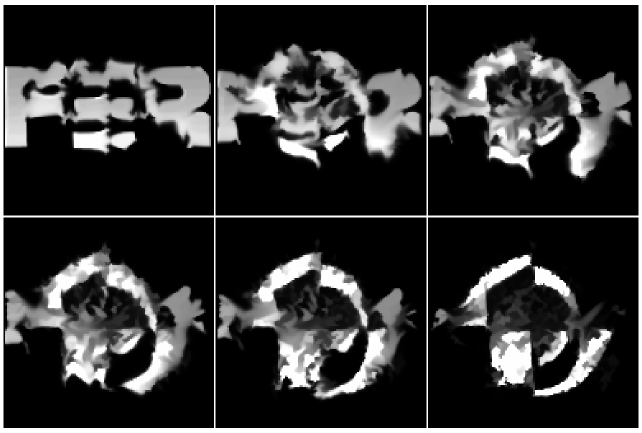


Fig. 10. FER evolving into FER logo (σ = 1).

One can also play with parameters and see what happens. If we change parameter σ to 7 and invert colors, we get the following result:

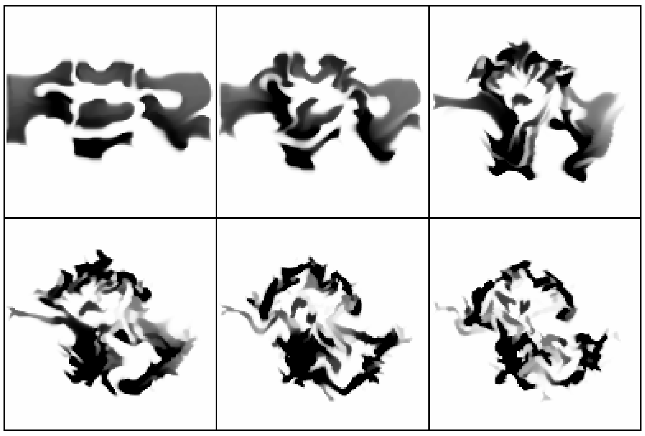


Fig. 11. FER evolving into FER logo (σ = 7).

6.2 Performance

Testing was done on three different configurations, each of which had the same amount of RAM (512 MB). Both 64 x 64 and 128 x 128 grids were used. We counted how many times MakeStep method got called within five seconds. If we divide that number by the time that passed, we get frames per second.

TABLE I
COMPARISON OF THREE CONFIGURATIONS

Configurations	FPS	
	64×64	128×128
Athlon 64 3000+ (1800 MHz)	95.14	23.06
Pentium IV 2400 MHz	65.6	15.5
Pentium IV 1800 MHz	44.81	11.91

VII. Conclusion

The implemented method produced more or less satisfying results. The only problem was setting parameters, because Fattal and Lischinski do not mention their referent values so we had to guess them. The key factor to achieving the best results using this method is experience.

Graphical user interface made the process of repeating simulations with different values easy. This is important for animators who want their tools to be powerful yet intuitive and simple to use.

References

[1] K. Sims, "Particle Animation and Rendering Using Data Parallel Computation", *ACM Computer Graphics (SIGGRAPH '90)*, p. 405, 1990.

[2] J. X. Chen, N. da Vittoria Lobo, C. E. Hughes and J. M. Moshell, "Real-Time Fluid Simulation in a Dynamic Virtual Environment", *IEEE Computer Graphics and Applications*, p. 52, 1997.

[3] N. Foster and D. Metaxas, "Modelling the Motion of a Hot, Turbulent Gas", *Computer Graphics Proceedings, Annual Conference Series*, p. 181, 1997.

[4] J. Stam, "Stable Fluids", *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*, p. 121, 1999.

[5] A. Treuille, A. McNamara, Z. Popovic and J. Stam, "Keyframe Control of Smoke Simulations", *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, p. 716, 2003.

[6] R. Fattal and D. Lischinski, "Target-Driven Smoke Animation", *SIGGRAPH 2004 Conference Proceedings, Annual Conference Series*, p. 441, 2004.