REAL−TIME SIMUALTION OF SMOKE THROUGH PARALLELIZATIONS

Torbjørn Vik, Anne C. Elster & Torbjørn Hallgren
NTNU, Norway

Animation/visualization of smoke has long been a topic of interest in computer graphics, and following the increase in common processing power researchers have started using physical simulations to generate these animations. Smoke, and other fluids, is extremely hard to animate by hand, and simulation techniques producing realistic results are highly applicable and desirable in areas ranging from computer games to weather forecasts. To maintain a reasonable level of usability, these techniques should allow a certain degree of control, and not be prohibitively expensive computationally. However, since these tasks are extremely compute−intensive, powerful parallel hardware systems is needed to solve these tasks in real−time, thus elevating the programming challenges up to a higher level.

Our work combined mathematical, graphical and technical techniques which resulted in a real−time application exploiting dual−cpu hardware to visualize rolling and twirling smoke in three dimensions. One of the essential goals was for our simulation to run in *real−time*, producing satisfactory visual results. In order to have a basis of comparison, we defined *real−time* to approx 1 fps (frames per second).

Smoke Simulations

The modeling of natural phenomena such as smoke remains a challenging problem in computer graphics. This is not surprising since the motion of gases/fluids such as smoke is highly complex and turbulent. Also, building animation tools for such fluid−like motions is an important area with many obvious applications in the industry including special effects and interactive games. Physical models, unlike key frame or procedural based techniques, permit an animator to almost effortlessly create interesting, swirling fluid−like behaviors. Also, the interaction of flows with objects and virtual forces is handled elegantly. Ideally, a good smoke model should both be easy to use and produce highly realistic results.

Obviously the modelling of smoke and gases is of importance to other engineering fields as well. More generally, the field of computational fluid dynamics (CFD) is devoted to the simulation of gases and other fluids such as water. Only recently have researchers in computer graphics started to excavate the abundant CFD literature for

algorithms that can be adopted and modified for computer graphics applications. Unfortunately, many computer graphic smoke models are either too slow or suffer from too much numerical dissipation.] adapts techniques from the CFD literature specific to the animation of gases such as smoke, and the model used are stable, rapid and doesn't suffer from excessive numerical dissipation. This makes it possible to produce animations of complex rolling smoke even on relatively coarse grids (as compared to the ones used in CFD).

Our Smoke Simulation

Our smoke simulations is based on an approach described in [0] that exploits physics unique to smoke in order to design a numerical method that is both fast and efficient on the relatively coarse grids traditionally used in computer graphics applications (as compared to the much finer grids used in the computational fluid dynamics literature). The model uses inviscid Euler equations instead of Navier−Stokes, as the former usually are more appropriate for gas modeling and less computationally intensive.

Both simulation and visualizing algorithms needed to handle three dimensional domains. Compared to our original 2−dimentsional prototype, our 3D work introduces a more complicated data structures and rendering code while the simulation code stays more or less the same. The linear interpolation of off−grid values now involve eight reference points instead of four, and the matrix built for solving Poisson equations, of course, grows accordingly to the extra dimension. The relationship between rendering and simulation also changes as the model is extended to three dimensions. More processor intensive calculations are required to render the results, especially in respect to lighting as the effect of self−shadowing is a very important aspect when rendering volumes of smoke. This is currently achieved using a fast Bresenham line drawing voxel traversal algorithm [0] to trace rays of light through the volume.

Both simulation and rendering code needed to detect and exploit hardware capable of running multiple parallell instructionstreams (dual−cpu). Algorithms must were hence chosen and designed to make efficient use of such hardware.

Top level pseudocode for our simulation program is presented in Figure 1.

```
For each frame
1     Add forces to velocities
```

```
2     Solve advection term (Semi−Lagrangian)
3     Solve Poisson equations
4     Conserve mass by subtracting gradient
5     Advect density and temperature (Semi−Lagrangian)
6     Calculate lighting
7     Render
```

Figure 1 Top level pseudocode.

Step 1: Add forces to velocities

Step 1 updates the velocity field, according to different forces. This is done simply by multiplying the force of each voxel with the timestep Δt and adding the result to the voxel's velocity. The forces include user defined fields and the buoyancy force.

```
For each voxel
1.1   U = U + F_u * dt
1.2   V = V + F_v * dt
1.3   W = W + F_w * dt
```

Figure 2 Low level pseudocode for Step 1

In other words, these computations involve standard vector−vector additions. A simple data−parallel approach would do for this step of the algorithm. Some care had to be taken to avoid several threads trying to access the same cacheline at the same time.

Step 2: Solve advection term (Semi−Lagrangian)

This step solves for the advection term in , using a Semi−Lagrangian Semi−Implicit (SLSI) scheme. This builds a new grid of velocities from the ones already computed, using the trapezoidal method as integration method. The center of each voxel is traced through the velocity field, and the velocities $u(x,t)$ are linearly interpolated at the arrival points. The values are then transferred back to the cells the trajectories originated from. Simple linear interpolation is easy to implement, gives satisfactory results and is unconditionally stable as it never overshoots data. Higher order interpolation schemes are, however, desirable in some cases for high quality animations, but as our prototype aimed for a real−time environment, the processing cost of higher order interpolation was not considered worth the effort.

For each voxel at position $x = [x, y, z]$

2.1 Calculate $x^* = x - \frac{\Delta t}{2}\left[u(x,t_n) + u\left(x - \Delta t u(x,t_n), t_n - \Delta t\right)\right]$

```
2.2   Set temporary velocity u* for each voxel to
      u(x*,t)
```

Figure 3 Low level pseudocode for Step 2

Step 2.1 involves linear interpolations of two velocity vectors. The first, $u(x,t)$, is the current velocity at the voxel to be updated. The other, $u(x-\Delta t u(x,t_n), t_n - \Delta t)$, is an off−grid velocity vector needed in the trapezoidal method. At first glance, step 2 seems just as suitable for a simple data−parallel approach as step 1. Unfortunately, the off−grid interpolation involves several other voxels in addition to the current voxel. Depending on the length of the velocity vector, the additional voxels need not even be neighbouring voxels but could be located far away from the voxel to be updated. If a data−parallel approach is chosen, this fact may cause race conditions as several threads may try to access a given voxel at the same time. Restricting the user defined velocities/forces would reduce the possibility of this condition to occur, but such limitations would also reduce the simulator's useability and hence isn't very desirable. In general, these race conditions may be reduced by giving the different threads working areas located sufficiently far away from each other in memory. The distance between the memory locations depends on the relation between total number of threads and number of voxels in the grid. More threads/less voxels will result in smaller working areas, thus locating them closer to each other and higher possibility of race conditions.

Step 3: Solve Poisson Equations

In step 4, the velocity field is forced to conserve mass, and this requires the solution of a Poisson equation for the pressure in Step 3. A matrix of coefficients is constructed using the five−point formula approximation to Laplace's equation, and the system of equations is solved using the Conjugate Gradient method (CG).
Details of this step is show in Figure 4 (next page).

PETSc implements a number of different preconditioners for CG, together with a few direct solvers. Table 1 shows a comparison of the different PETSc preconditioners we tested. Two of the most promising preconditioners (Incomplete Cholesky and Multigrid) were not implemented for our test program's matrix format, and hence not included in the table.

3.1 Build solution vector $\vec{b}$. For each voxel $[i, j, k]$: $\vec{b}_{i,j,k} = \nabla^2 p_{i,j,k} = \frac{1}{\Delta t} \nabla \cdot u_{i,j,k}$ *

3.2 Solve equations (CG):

$i \Leftarrow 0$
$\vec{r} \Leftarrow \vec{b} - A\vec{x}$
$\vec{d} \Leftarrow \vec{r}$
$\delta_{new} \Leftarrow \vec{r}^T \vec{r}$
$\delta_0 \Leftarrow \delta_{new}$
while $i < i_{\max}$ *and* $\delta_{new} > \varepsilon^2 \delta_0$ *do*
  $\vec{q} \Leftarrow A\vec{d}$
  $\alpha \Leftarrow \frac{\delta_{new}}{\vec{d}^T \vec{q}}$
  $\vec{x} \Leftarrow \vec{x} + \alpha \vec{d}$
  If i *is divisible by* 50
    $\vec{r} \Leftarrow \vec{b} - A\vec{x}$
  else
    $\vec{r} \Leftarrow \vec{r} - \alpha \vec{q}$
  $\delta_{old} \Leftarrow \delta_{new}$
  $\delta_{new} \Leftarrow \vec{r}^T \vec{r}$
  $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
  $\vec{d} \Leftarrow \vec{r} + \beta \vec{d}$
  $i \Leftarrow i + 1$

Figure 4 Low level pseudocode for step 3

Preconditioner	Time	Iterations	Error
None	13.25 s	475	0.000944
Jacobi	13.59 s	475	0.000944
SOR	38.92 s	1000	34101857.585735
Block Jacobi	7.40 s	142	0.000690
Eisenstat	11.49 s	168	0.000747
Redundant	8.66 s	142	0.000690
Asm	9.89 s	142	0.000690
Incomplete LU	8.50 s	142	0.000690

Table 1 Comparison of PETSc's preconditioners

Direct solvers			
SLES	55.49 s	2	0.000000
LU	4.75 s	1	0.000000

Table 2.

The PETSc solver was tested with a set of equations with 40.000 unknowns, built out of a standard 2D Laplace problem. All benchmarks were run with a suggested solution vector. This vector was exported from the prototype, hence representing an actual situation in the simulator.

Step 4: Conserve mass by subtracting gradient

Step 4 is a standard vector−vector subtraction:

For each voxel $[i, j, k]$

4.1 Update velocity $u_{i,j,k} = u_{i,j,k}{}^{*} - \Delta t \nabla p_{i,j,k}$

Step 5: Advect density and temperature (Semi−Lagrangian)

For each voxel at position $x = [x, y, z]$

5.1 Calculate $x^{*} = x - \frac{\Delta t}{2}\left[u(x, t_n) + u\left(x - \Delta t u(x, t_n), t_n - \Delta t\right)\right]$

5.2 Set temporary velocity u^{*} for each voxel to $u(x^{*}, t)$

Figure 5 Low level pseudocode for step 5

Final Comments:

Numerically, we have so far focused on selecting an appropriate preconditioner for the Conjugate Gradient method, based on "benchmarks" for the different alternatives. Alternatives to CG solving the equations of fluid flow should also be investigated.

Our final paper will include performance results and resulting visual images.

References

[0] R. Fedkiw, J. Stam, and H. W. Jensen. Visual Simulation of Smoke. In *SIGGRAPH 2001 Conference Proceedings, Annual Conference Series* , pages 15−22, August 2001

[0] J. D. Fowley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice. Second Eidtio*n. Addison−Wesley, Reading, MA, 1990.