A General Animation Framework for Gaseous Phenomena

Jos Stam
VTT, Information Technology
P.O. Box 1203, FIN-02044-VTT
Tekniikantie 4 B, Espoo, Finland

Abstract

This paper presents a new animation framework for the modeling of gaseous phenomena. We combine particle and grid based techniques in an innovative way. Based on this framework, our system allows for an incremental design of animations. In the first stage, an animator uses particle methods to model the evolution and appearance of gases. Grid based techniques are subsequently employed to compute high quality animations. Central to the success of our technique is a new algorithm to efficiently advect densities on grids. To demonstrate the effectiveness of our approach, we have included many animations.

Key Words & Phrases: Animation, Modeling and Rendering of Gaseous Phenomena, Turbulence, Advection of Densities, Volume Rendering.

Supported by an ERCIM (European Research Consortium for Informatics and Mathematics) Fellowship. This work was done while the author was visiting VTT from July 1996 to March 1997.

1. Introduction

The modeling and animation of natural phenomena has received much attention from the computer graphics community. Synthetic models of natural phenomena are required for such diverse applications as flight simulators, special effects, video games and architecture. The level of “accuracy” required of the animations depends greatly on the application at hand. In this respect, computer graphics applications differ from most other engineering applications. Often one single criteria cannot “validate” the animations produced by a given computer graphical system. In practice, the success of a given system depends on how easily it can translate an animator’s dream into a concrete animation. Therefore, one of the most important features of an animation system is the ability for the animator to *control* both the appearance and the evolution of a given phenomenon. In many applications, such as in the special effects industry, synthetic images are substituted or composited with real film sequences. In these cases the animations should behave “naturally” and have a “film-like” appearance. The use of physical models can greatly assist in producing such animations. See [Rus94] for a good review of related models of gaseous phenomena from other areas and their possible application to computer graphics. However, for many interesting phenomena, including gaseous phenomena, the physical equations describing such phenomena are *non-linear*. Therefore, simulations based on these equations are both hard to solve and more importantly hard to control. Instead, we provide the animator with an ensemble of modeling primitives. More precisely, we propose a new animation framework designed especially for the simulation of gaseous phenomena such as clouds, smoke, steam and fire. The emphasis of our model is on enabling an animator to easily produce a desired sequence. The core of our animation system was designed from basic observations of gaseous phenomena. We allow an animator to control the motion of the gas through the use of both motion fields and gas sources. From these primitives, our system can compute gas animations at various levels of quality, dependent on the computing power available. In our system, an animator uses fast particle-based solvers at the early stages of the design and high quality grid-based solvers at the final stages.We allow for a smooth transition between these stages. Similarly, the appearance of the gas is modeled incrementally. Aspects of the gas such as color and transparency are modeled from early on, while the computation of more subtle effects, such as scattering, are delayed until the end. Many of these ideas have been exploited in prior work. In one way, our system can be viewed as a synthesis of most existing models for gaseous phenomena. Therefore, we briefly discuss previous models next.

Broadly, computer graphical models for gaseous phenomena are either *particle-based* or *grid-based*. In particle-based methods, the gas is modeled as a set of primitives which are advected, i.e., moved, through a flow. Early models used point particles to model fire eruptions, for example [Ree83]. Later, the technique was enhanced by modeling each particle as a fuzzy *blob* of matter [Sim90,StFi93].To make these blobs appear less synthetic, the blobs can be texture mapped [Gar94] or deformed by the surrounding motion field [StFi95]. Many other primitives have been developed mainly in order to visualize certain flows. For example, models have been developed to compute the deformations of simple polyhedral elements, called “flow volumes”, as they move through a flow field [BLM95]. However, these methods cannot usually account for the arbitrary deformations that most gases undergo. Grid-based methods, on the other hand, allow the shape of the gas to be arbitrary within the accuracy of the grid. Early grid-based methods modeled the evolution of the gas by varying the parameters of a solid texture over time [EbPa90]. Alternatively, these parameters can also be derived from a statistical model of the gases’ density [Sak93]. More recently, however, many novel techniques have been developed to animate grid-based densities. For example, flows have been used to advect solid textures through a grid [Ebe93]. Motion fields are thus used in both methods to model the general evolution of the gases. These fields can be modeled either by simple primitives [Sim90,WeHa91] or as vector valued textures.The latter can be modeled as the gradient of a solid noise function [Per85], as an incompressible random vector field [ShFo92,StFi93] or as an ensemble of evolving point vortices [CMTM94,GLG95]. In short, the main advantage of particle-based methods is that the motion of the gas can be visualized in real-time and the main disadvantage is that the appearance of the gas often appears synthetic. On the other hand, grid-based methods can produce arbitrary gas distribution, but are expensive both to compute and to visualize. However, a considerable amount of research is devoted to making grid-based volume renderers more efficient, e.g., [LaLe94].

In this paper, we combine particle and grid-based methods into one system. Particle-based methods are used in the modeling phase, while grid methods are used to produce the final animation sequence. Although this paper has the appearance of a systems paper, it presents new algorithms which have not been published before. Specific contributions of this paper include a new and efficient grid-based motion solver (Section 5), a particle solver consistent with the grid solver (Section 5), a new method of animating flow fields (Section 4), an algorithm to generate particles from arbitrary sources (Section 3) and a concise implementation of a turbulence synthesis algorithm (Appendix A). We stress also that this is not a rendering paper but rather a research paper presenting a new animation framework for gaseous phenomena. The paper is organized as follows. In the next section we provide an overview of our system. Then, the following three sections focus on the different components of our model: gas sources, motion fields, motion solver and gas renderer, respectively. In Section 7, we describe several animations which we have created using our system. Finally, in Section 8 we conclude and outline promising directions for future research.

2. Overview of the System

The architecture of our system is based on a simple phenomenology of gaseous phenomena. We assume that the behavior of gases is determined entirely by a set of *sources* and *motion fields*. Sources introduce new matter into the environment, while the motion fields carry this matter through the environment. In addition, the evolution of the gas is controlled by a certain amount of *diffusion* and *dissipation*, both of which alter the distribution of gaseous matter over time.The appearance of the gas is a function of its color, of its transparency and of the way it absorbs and scatters incoming light and how it emits light. The parameters corresponding to these entities each have well defined physical units, which is desirable for many reasons. Most importantly, it ensures that the results from both our particle and grid solvers are consistent. The use of physically meaningful units is also suitable when physics based renderers are employed to visualize the gases or when physics based data, such as precomputed flow fields, are used. For reference, we give an overview of all our parameters with their units in Table 1. For simplicity, we have kept the number of parameters as small as possible. While more parameters increase the amount of control over the phenomenon, they also render this control more difficult. The number of such parameters is thus a tradeoff between control and *ease* of control. We intend to convince our reader that our set of parameters is nearly optimal in this respect. Complexity is introduced into our system mainly by multiplying our parameters by unitless texture maps. These maps can be either entirely synthetic or synthesized from an empirical stochastic model [FFC82]. We favor the latter since it usually produces “natural” behaviors in an almost automatic manner.

parameter	description	units (T=time, L=length, M=mass, I=radiance)
s_rate	rate of the source	T^{-1}
s_max	maximum value of the source	ML^{-3} (gas) LT^{-1} (field)
f_mag	magnitude of a motion field	LT^{-1}
diff	amount of diffusion	L^2T^{-1}
diss	amount of dissipation	T^{-1}
alb	albedo of the gas	no units, values beween 0 and 1
ex	exinction cross-section	$M^{-1}L^2$
em	emission of the gas	I
col	color of the gas	no units, each component between 0 and 1
size0	initial size of gas blob or field	L
T()	scalar texture map	no units, values are between 0 and 1
V()	vector texture map	no units, values of each component are between -1 and +1
n_new	rate of new blobs/fields	T^{-1}
time	current time of the simulation	T
step	time step of the simulation	T

Table 1: Parameters of our system controlling the behavior and the appearance of gases.

Our system comprises three main modules: a modeler, a simulator and a renderer, as shown in Figure 1. The modeler allows an animator to specify the geometry of a particular scene. We opted for a simple single-view modeler inspired by the SKETCH system [ZHH96]. The modeler is also used to specify the locations and shapes of the gas sources and motion fields. Parameters and geometry are either inputted interactively with the mouse or entered manually. In addition, our modeler is able to save geometries and parameters in a script file.

The simulator is used to compute both the evolution of the gases and that of the motion fields. The gas simulator works in either particle mode or grid mode. In particle mode, the gas is represented as a superposition of spherical fuzzy blobs with a given mass and size. The sources are utilized to introduce new blobs into the environment, while the motion fields are used to move them around. Diffusion and dissipation alter the mass and size of each blob. In grid mode, the densities of the gas are defined on a user specified grid and are incrementally updated at each time step. New matter is introduced by evaluating the sources at each grid point. The motion is computed using our new grid motion solver (Section 5). The particle solver is much faster and therefore is used in the modeling phase. Although the grid solver is slower, it produces higher quality animations and is thus employed to produce the final animation. In the following sections, we describe how the results from both methods can be made consistent. The simulator can also be used to animate the motion fields in a manner similar to that of the particle gas solver. In this case, the sources create new field primitives instead of blobs of gas. Similarly, motion fields are used to move these fields through the environment. For example, as depicted on the left of Figure 2, a hot surface may generate a steam density and vortices, all of which are advected by a common heat field. As in the case of the gaseous blobs, the size and magnitude of each field can be varied by specifying a certain amount of diffusion and dissipation. Alternatively, single fields may be advected by other fields. For example,the turbulent field of smoke billowing from a smokestack can be modeled to follow the global directional field (wind) (see the right hand side of Figure 2). Thus, the role of the simulator is to compute the evolution of the gases and some of the motion fields. We describe the type of motion fields used in our system in Section 4.

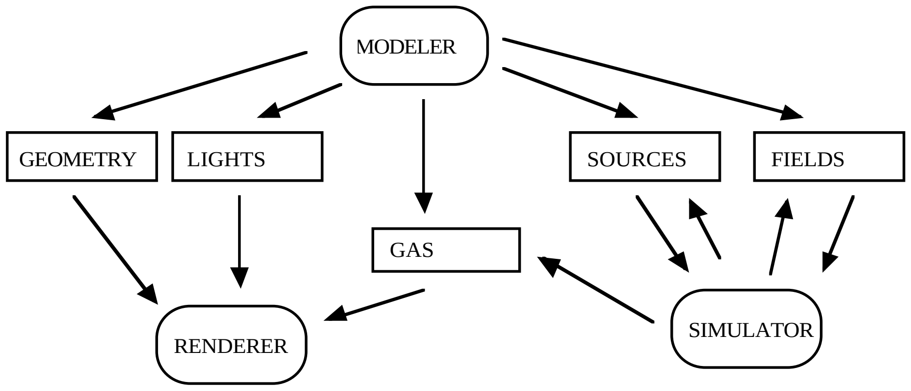

Figure 1: Overview of our system. There are three main modules: the modeler, the simulator and the renderer. Both the the gas sources and motion fields are allowed to evolve over time.

Our rendering model depicts the output of the particle solver in real time. Each blob is displayed either as a point or as a *splat*.. The latter is a polygonal approximation of the projection of the gaseous blob onto the viewing plane [Wes90]. Splats can be rendered in real-time on machines equipped with hardware polygon shaders and an accumulation buffer. The animator is thus able to view the motion and appearance of the gas while adjusting the parameters of our system. Once a desired behavior and look has been thus achieved, the grid solver is invoked and its output is rendered within a high quality renderer. In our case we added a grid-based volume renderer to a ray tracer. We describe both our interactive renderer and grid ray tracer in more detail in Section 6.

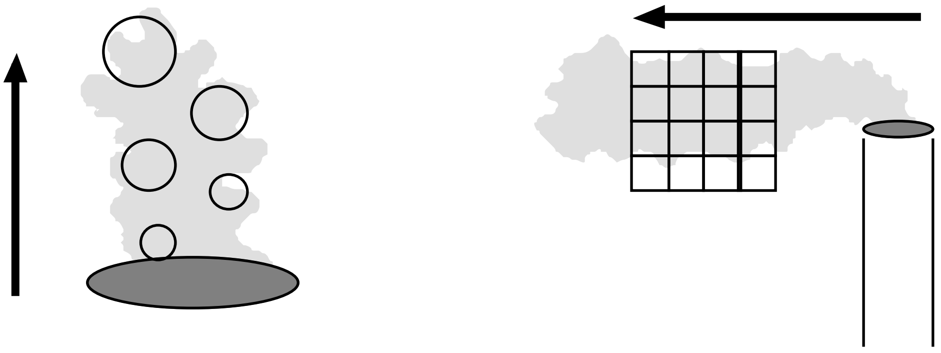

Figure 2: Our simulator computes the evolution of both gases and motion fields. In the picture on the left the disk source generates both gas densities and vortices. The vortices are advected by the single directional heat field. In the picture on the right a turbulence defined on a grid is advected by a directional wind field. In both example, the gas is advected by both fields.

3. Sources

Sources are used in our system to introduce new gaseous matter and fields into the environment. A source is defined by a region in space *S*, a rate `s_rate`, a maximum value `s_mag` and a texture map `T()` all of which are specified by the user. The region is defined by mapping the unit cube $[-1,+1]^3$ using an affine transform `M` (translation, scale and rotation). The transformation is specified by the animator, as is commonly done in computer graphical modelers. For the gas simulator, we need two routines: one that introduces new particles and one that evaluates the source at a given point in space. The latter is readily computed as

```
eval_src ( x ) {
      y = mmult(M^-1,x);
      if ( in_unit_cube(y) ) return step*s_rate*s_mag*T(y,time);
      else return 0;
}
```

The routine that generates particles from the source is used in the particle-based gas simulator and to introduce new fields into the environment. In the rest of this section we use a terminology consistent with a gas source. For field sources, “particle” should be replaced by “field” and “mass” by “mass-velocity”. The number and size of new particles are controlled by the parameters `n_new` and `size0`, respectively. At each time step, `n_new` particles are introduced in the environment, their location and mass depending on the parameters of each source. The basic idea is to compute the total mass to be released at the given time step for each source. This mass is then divided among the `n_new` particles, whose density is further determined by the initial size. The locations of the particles are a function of the distribution of the density within the source given by the texture map `T()`. The assignments of the locations are akin to generating random variables from a multivariate probability distribution.We first provide the process to generate new particles from a uniform source, i.e., `T()=1`, and then indicate how this process can be generalized to arbitrary distributions. For a uniform source, the total mass to be released for a given time step is equal to

```
m_src = step*s_rate*s_mag*vol(M)*8,
```

where the `vol()` function computes the volume corresponding to the unit cube transformed by `M`. For an arbitrary affine transformation, the volume is given by the product of the inverse of the square roots of the eigenvalues of the matrix, i.e., it is proportional to the scaling part of the transformation. The factor “8” accounts for the volume of the unit cube. Let `m_tot` be equal to the sum of the masses that each source must release. Then each source has to release `n_new*m_src/m_tot` new particles. In general, this number is not integral. Therefore, each source releases only the integral part of the number of particles. The rest is accumulated. More precisely at each time step:

```
n_acc += n_new*m_src/m_tot;
n = (int) n_acc;
n_acc -= n;
/* release n particles of mass m_src/n uniformly within S */
```

There are two alternative ways to model non-uniform source distributions. In the case when the texture map `T()` corresponds to a well known probability density function, we can borrow statistical techniques to generate random variates. In general, by discretizing the domain of the source we can bring ourselves back to the uniform source case. Indeed, a discretized source is equivalent to many uniform sources. The level of the discretization is a tradeoff between efficiency and spatial accuracy. Now that we have described the mechanisms that introduce matter in our system, we explain how this matter is moved further into the environment.

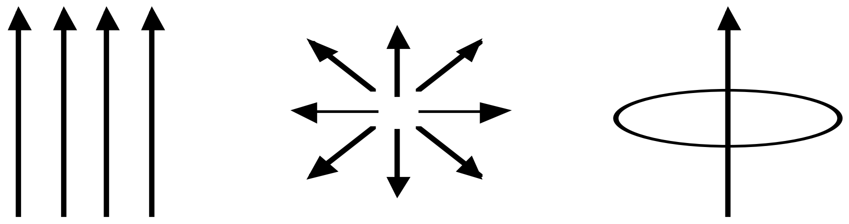

Figure 3: The simple motion fields used in our system. From left to right: directional, repulsive and vortex. In additio each field is allowed to decay in magnitude with the distance from the center of the field.

4. Motion Fields

Motion fields, like sources, are first defined within the unit cube and then interactively placed into the environment through an affine transformation by the animator. Figure 3 shows some of the simple field primitives found in our system. New primitives are easily added to our system. All that is necessary is the definition of the field within the unit cube. For example, a simple vortex field aligned with the z axis is defined as:

```
eval_vortex ( {x,y,z} ) {
      r = norm({x,y,z});
      if ( r>1 ) return {0,0,0};
      v = weigh_func(r)*{-y/r,x/r,0}
      return v;
}
```

The function `weigh_func()` is a blending function which controls the dropoff of the vorticity. The evaluation of all the motion fields is thus achieved by mapping the point back to the coordinates of the field. Care has to be taken to transform the returned velocity back to world coordinates. For each field the following commands are thus invoked:

```
eval_field(x) {
      y = mmult(M^-1,x);
      v = eval_primitive(y);
      mag = f_mag*norm(v);
      v = mmult(M,v);
      v *= mag/norm(v);
      return v;
}
```

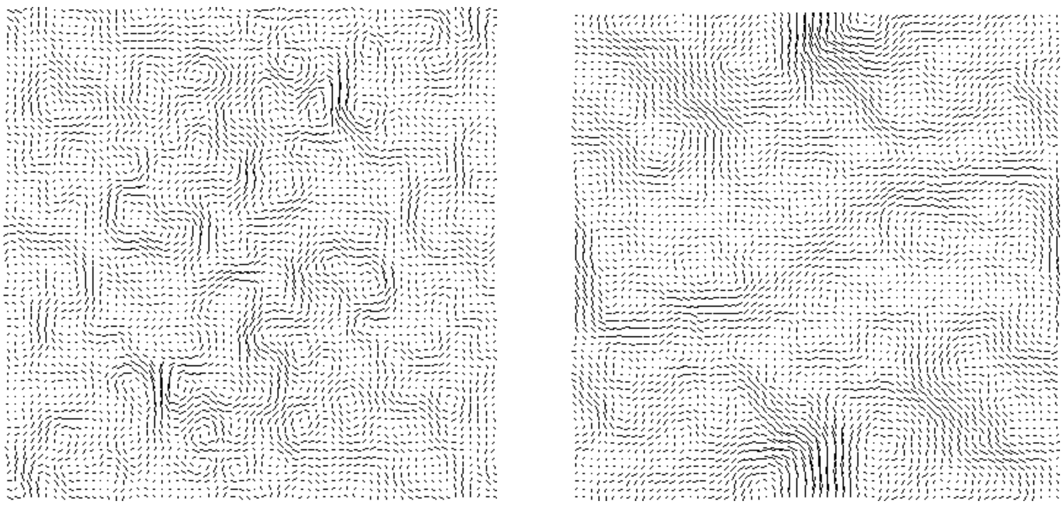

Figure 4: Two slices of our turbulences. The field on the left is a smooth turbulence field, while the field on the right depicts a fractal turbulence. Notice the "vortice-like" structures in both fields. These result from the assumption of incompressibility and are a key ingredient to producing "natural-looking" motions of gases.

To add complexity to the motion of the gas, we use vector valued texture maps which we call *turbulences* in this paper. Previous turbulence models include those derived from solid textures [Per85], those computed as a superposition of point vortices [CMTM94], and those precomputed using a spectral synthesis technique [StFi93]. We have adopted the latter class of fields because it is founded on a sound phenomenological basis ("turbulent cascade"), it is incompressible and it provides us with time-varying fields defined everywhere in space and time. The latter property is a consequence of the use of the fast Fourier transform. In [StFi93] and [ShFo92] only a "fractal" like spectrum corresponding to the "$k^{-5/3}$-law" was used to model the spatial structure of these fields. In our system we distinguish two classes of turbulences: "smooth" and "fractal". The smooth turbulences are generated from a Gaussian spectrum while the fractal turbulences are generated from an inverse power law spectrum. More precisely:

```
      S_smooth ( k ) { return ( exp(-k*k/(param*param))/param ); }
      S_fractal ( k ) { return ( pow(k,-param) ); }
```

where `param` is arbitrarily provided by the animator (units are irrelevant here since we normalize the resulting turbulence). Similarly, the temporal structure of our turbulences is modeled using either `S_smooth()` or `S_fractal()`. The resulting turbulences can thus be of a mixed type, e.g., fractal in space and smooth in time. We have experimented with other spectra, but with results which were too similar to either the smooth or fractal turbulences. Figure 4 compares the spatial structure of the smooth and fractal turbulences. Algorithms to generate turbulences with given spectra were given in both [ShFo92] and [StFi93]. The reader interested in the mathematics behind these algorithms is referred to these papers. In Appendix A we give a more concise algorithm based on a neat geometrical interpretation of the condition of incompressibility in the frequency domain. Indeed, in the frequency domain the vector velocity of an incompressible turbulence at each spatial frequency lies in the plane normal to that frequency [MoYa75]. The algorithm generates a turbulence defined on a four dimensional grid. The values of the field between grid points are obtained via linear interpolation.

5. Motion Solver

The simulator lies at the heart of our system and determines the evolution of both the shape of the gas and that of some of the motion fields. The simulator can run in two different modes. In particle mode the sources introduce new particles in the system, the fields move them around and the diffusion and dissipation control the size and mass assigned to each particle. In this mode the speed of the simulation is either in "real-time" or is run with a fixed time step. The duration of the simulation is provided by the animator in the form of a start time and an end time. In real time mode, the time step is updated after each simulation and display step in order to match the real time that has elapsed so far. This option is very useful since it provides the animator with a good approximation of what to expect in the final animation. While the simulator is running, the animator has the option of changing any of the parameters and is able to view their effect in real-time using the particle renderer (see next section). The effect of the diffusion is to increase the size of each particle proportional to the square root of the simulation time elapsed (this is consistent with the units of the diffusion parameter). The dissipation parameter accounts for an exponential decay of mass over time.The basic procedure of the simulator in particle mode is thus summarized as follows:

```
      time = start_time;
      while ( time < end_time ) {
            generate_new_particles ();
            for ( /* each particle p do */ ) {
                  v = eval_field(p->center);
                  p->center += step*v;
                  p->size += sqrt(step*diff);
                  p->mass -= step*diss*p->mass;
            }
            if ( real_time_mode ) step=clock()-time;
            time += step;
      }
```

The function `clock()` returns the real time passed since the start of the simulation. We now describe in detail our grid-based solver which is used to produce high quality simulations.

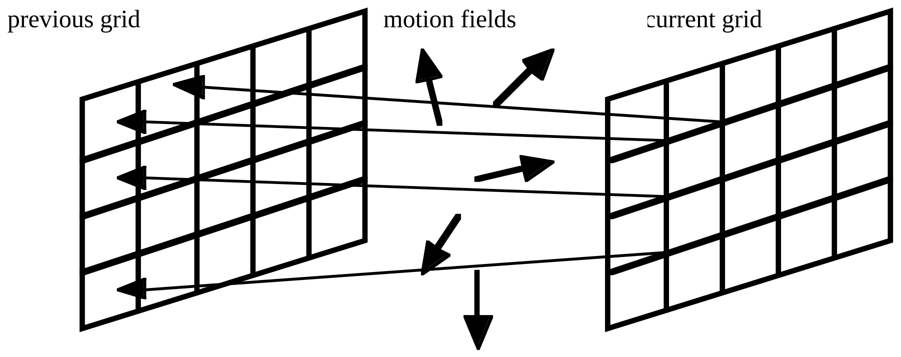

Figure 5: Two dimensional analogue of our grid motion solver. At each time step the grid points of the current grid are backtraced along the motion fields. The density at the destination is interpolated from the density values stored in the previous grid.

In grid mode, the density of the gas is defined on a three-dimensional grid with a user defined resolution. The location and size of the grid are placed interactively in the environment by the animator. Ideally, the grid size should be big enough to contain all the blobs when the simulator is run in particle mode. However, the grid size could be smaller when only part of the gas is visible in the final view. The resolution need not be the same for each coordinate. For instance, in the case of gases which are nearly confined in a plane, e.g., stratus clouds, it is more efficient to use a smaller resolution along the “thickness” of the plane. The resolution of the grid is of course also limited by the hardware available to the animator. In grid mode the simulation time step is set to the local video refresh rate (1/30 or 1/25). We require two grids to compute the evolution of the gas over time. At each step the *current grid* `dens1[]` is updated from the density values stored in the *previous grid* `dens0[]`. After each update, we alternate the grids. The sequence of updates for each time step is as follows: advection, add sources, dissipation and diffusion. The last three updates, being fairly straightforward, are described first. The effect of the sources is accounted for by evaluating the source at the current grid point and adding it to the value from the previous grid (see Section 3). Dissipation is solved by taking the density at each point of the current grid multiplied by the dissipation parameter and the time step and subtracting the result from itself, i.e.,

```
        dens1[i][j][k] = (1-diss*step)*dens0[i][j][k].
```

Diffusion is accounted for by solving the corresponding tridiagonal system in each coordinate. This is a fairly standard procedure and is fully explored in [NRC88].

To resolve the advection part, i.e., the influence of the motion fields on the gas, we propose a new method that incrementally updates the grid. One solution would be to treat each grid point as a particle and to move these particles along the motion field over a period equal to the time step of the simulation. From these new particles we could then reconstruct the density on the grid using a sparse interpolation scheme. The disadvantage of this method, however, is that it is prone to aliasing, since the distribution of points is very unlikely to be uniform. Instead, we trace each grid point *backwards* in time and then use an interpolation scheme to find the density at the destination. In this way, we guarantee that the density is always regularly sampled on the grid, and no aliasing can occur. In Figure 5, we illustrate our method. However, instead of aliasing, our method can suffer from numerical diffusion and dissipation. The importance of these effects depend on both the quality of the interpolant and the grid resolution. In practice though, dissipation and diffusion are present in real gases and as long as we can keep these effects relatively low, they do not affect the quality of the final animation. We obtained good results using a linear-interpolant and grids as small as 32^3. The method is summarized in the following algorithm. The transformation `M` models the location and size of the computational domain, while `Nx`, `Ny` and `Nz` are the resolutions in each coordinate of the grid.

```
advect ( dens1, dens0 ) {
    for ( i,j,k=0 ; i,j,k<Nx,Ny,Nk ; i,j,k++ ) {
        x = grid_to_unit_cube({i,j,k});
        y = mmult(M,x);
        v = eval_field(y);
        y = y - step*v;     /* higher order scheme can be used here */
        x = mmult(M^-1,y);
        {i0,j0,k0} = unit_cube_to_grid(x);
        dens1[i][j][k] = lin_interp(dens0,i0,j0,k0);
    }
}
```
In order to account for backtraces which leave the grid, we add a “boundary layer” around our grids with zero density. Matter is thus effectively lost outside the grid. During the particle-solver the animator can interactively modify the size of the computational domain, i.e., `M`, such that the particles remain within the domain.

There is some previous work related to our grid solver. Many researchers have used similar techniques to visualize flow fields by distorting texture maps. At the start of the simulation, the texture is undistorted. At later times, the point is backtraced along the field until the original texture is reached. Efficient schemes have been devised in two-dimensions for static motion fields by updating the vector displacement over time [Sim92]. For evolving fields, similar schemes have been devised using the Jacobian of the transformation [BLM95]. Three-dimensional textures were advected similarly by backtracing grid points to the original texture [MCW92]. But, this method becomes expensive for large time intervals. It is unlikely that our technique should prove to be useful in visualization, since information is lost due to numerical diffusion. However, our model is very successful in producing animations of gaseous phenomena. Similar methods were also used at the rendering stage in both [Ebe93] and [StFi95]. In these methods, the point to be rendered in the gas is backtraced through time, making these methods expensive for long time spans. Our method differs from these methods in that we incrementally update the gases’ morphology before the rendering. Different types of renderers can therefore be used in conjunction with our solver.

We now describe how we depict our gases at the various stages of the design process.

6. Rendering

Our renderer essentially works in two different modes: interactive and batch. Our interactive renderer is implemented using the OpenGL library [OGL92]. This library has the advantage of being available on many different architectures. The routines automatically take advantage of any graphics hardware available on each architecture. On an Iris Indigo, for example, we were able to visualize our simulations in real-time. The interactive renderer is able to display the geometry of our scene, taking into account shading due to light sources. The interactive renderer is used to display the results obtained from our particle simulator. The particles are either displayed as points or as transparent disks corresponding to the projection of each blob onto the viewing plane. The cumulative effect of all the blobs is automatically accounted for by the alpha blending capabilities of the OpenGL library. This method is known as “splatting” in the volume visualization literature and was first introduced in [Wes90]. We used a simple hexagonal polygonization for our splat.The color at the edges of the polygons is a function of the color, emission and albedo of the gas. Indeed, the polygon material ambient component is set to the emission multiplied by the color, while the albedo is used to set the reflection of the polygon. The transparency of the polygon is determined by the alpha value at the vertices. The alpha value is zero for the external points and equal to the opacity of a single blob at the center. The opacity at the center of the splat is a function of the distribution of the mass in the blob, the extinction cross-section (`ex`) and the size of the blob. If we assume a linear decay of mass from the center, the opacity is equal to

```
alpha = 1-exp(-ex*col*mass/(3*PI*size*size))
```

The `3*PI` factor comes from the normalization due to the total volume of a single spherical blob of radius one, i.e., the value of the integral of the linear ramp 1-r, over the unit ball. Using the interactive renderer, the animator is able to view the effect of all the parameters on the appearance of the gas. However, in this mode the gas looks artificial and blob-like. Also, the effects from more sophisticated scattering models and the effects of self-shadowing cannot be achieved. For this, a higher quality batch renderer which takes the density output from our grid simulator is required.

We implemented a simple volume ray-tracer similar to the ones described in [Lev90] and [Sak90]. The grids used in the renderer are bigger than those used in our solver. We then traverse our grid using a three-dimensional version of a Bresenham line renderer which requires only integer arithmetic. The opacity values at each grid point are precomputed from the densities as follows:

```
      alpha[i][j][k] = 1-exp(-ex*col*dens[i][j][k]*(l_x+l_y+l_z)/3),
```

where `l_x`, `l_y` and `lz` are the spacings of the grid in each spatial coordinate, respectively. Prior to the grid traversal, we precompute the intensity at each grid point by determining the contribution due to each light source and multiplying it by the albedo of the gas. To this we add the emission of the gas:

```
      intens[i][j][k] = ( alb*shadow(i,j,k,light)*intens(light) + em )*col.
```

The contribution of each light source is also multiplied by the transparency `shadow()` caused by either the gas (self-shadowing) or by other gases or objects (cast shadows). We have implemented neither the arbitrary scattering distributions (non-constant phase functions) nor the effects of multiple scattering in this version of the batch renderer yet. For a good review of models which handle such effects, see [Max95]. The state of the art technique for cloud rendering taking into account sky light and realistic scattering is [NDN96].

7. Results

We now present various animations created using our animation system. To make these animations we had at our disposal both a Sun Sparc 20 workstation and an Iris Indigo. All animations were created in a period of three weeks, including both the modeling and the rendering of the sequences. We believe that both our hardware and the time constraints are typical of a computer graphics animation production. Figure 6 depicts a typical cycle in the production of an animation using our system. The image on the left shows our system with the motion solver in particle mode and the display in wireframe mode. In these modes, we specify the geometry and most of the parameters of the sources and the fields. The middle image shows our system in solid drawing mode. In this phase, we set the parameters controlling the appearance of the gas. The right image shows an actual frame from the final animation. This image is generated using our volume ray tracer applied to the output of our grid motion solver. We used the Iris Indigo mostly for the modeling while raytracing our frames on the Sun Sparc. For this sequence the modeling is done in real-time, while the grid solver took approximately two-seconds per frame at a resolution of 32^3. The raytracer took about one minute to render each frame at half the video resolution and with a grid size of 150^3.

Our other animations are depicted in Figures 7 through 13. In each figure, we show three pictures of the final sequence and a screen dump. The screen dump displays the sources (in blue), the fields (in red) and the computational domain of the grid (in white). The exact values of the parameters used in the animations are provided in Appendix B.

To achieve the most visual detail possible from our grids, we have set the diffusion to zero throughout. However, once larger grids are used, diffusion can be a useful parameter when modeling the blurring of a gas over time. Also note that we have used the same parameters for both `s_mag` and `s_rate` in each of our animations. We think that it is wise to keep these parameters in the system to ensure compatibility between physical units when using physics based data, for example. We emphasize that only small grids were used in these animations, due to the time constraints in producing the animations. Our system and hardware can in fact handle larger grids and we plan to recompute some of these animations at a much higher resolution in the near future.

8. Conclusions and Future Work

In this paper we have presented a new animation framework for gaseous phenomena. We implemented a simple system that incorporates this framework. We are very satisfied with the ease with which we were able to produce high quality animations of gaseous phenomena. Central to the success of our methodology is the coupling of particle and grid methods, and the use of our grid motion solver. Despite the fact that only simple field and source primitives were used, we were able to generate complex animations of gaseous phenomena. One of the key ingredients to the success of our animations is the use of our turbulences which automatically add complex detail to the motion.

Our system can be further refined in many ways. First, we are planning to include our volume ray tracer into a physics-based renderer such as Radiance [War94]. In addition, we intend to supplement our volume tracer with a preprocessing pass which computes the effects of multiple scattering and skylight. The use of a better renderer we hope will enable us to produce "photorealistic" (indistinguishable from a photograph) pictures of gases. We also plan to work on different turbulences. Although our current turbulences give good results, they are limited by the fact that they are homogeneous, i.e., statistically identical in space. For example, we expect turbulences to be different near a wall or behind the wake of an object. Also, we have not fully explored the possibilities offered by our field motion solver. Automatically generating vortices such as in [CMTM94] could be a promising approach. However, vortices in three-dimensions are rarely as simple as a rotational field around a straight segment. Finally, our grid motion solver can be refined by using either higher order interpolation schemes, a better advecting scheme at each time step or non-uniform grids.

More generally, we hope to apply a similar methodology to the simulation of water flows. This type of flow is more complicated due to the presence of a "surface" and coherence between fluid elements (continuity).

Appendix A. Turbulences

To generate our turbulences, we first compute an uncorrelated set of random complex vectors in the frequency domain. Each vector is scaled by the spectral density `S(k,l)` and projected on to the plane normal to each spatial frequency (incompressibility). The spectral density is defined entirely by the spatial and temporal spectra (see Section 4), e.g., `S(k,l)=S_fractal(k)*S_smooth(l)`. The turbulence is obtained by inverse transforming the set of random vectors. In addition, we have to ensure that our turbulence in the frequency domain satisfies certain symmetry conditions in order for the resulting field to be real valued [StFi93,NRC88]. The following algorithm implements these ideas.

```
gen_turb ( v ) {
   for ( it=0 ; it<=Nt/2 ; it++ ) {
      l = it/Nt;
      for ( ix,iy,iz=0 ; ix,iy,iz<Nx,Ny,Nz ; ix,iy,iz++ ) {
         kx, ky, kz = ix/Nx, iy/Ny, iz/Nz;
         if ( ix > Nx/2 ) kx = kx - 1.0;
         if ( iy > Nx/2 ) ky = ky - 1.0;
         if ( iz > Nx/2 ) kz = kz - 1.0;
         k2 = kx*kx+ky*ky+kz*kz;
      /* generate random complex vector */
         tx, ty, tz = unif(0.0,2*PI);
         RE(Wx,Wy,Wz) = S(sqrt(k2),l)*cos(tx,ty,tz);
         IM(Wx,Wy,Wz) = S(sqrt(k2),l)*sin(tx,ty,tz);
```

```
        /* project onto plane normal to {kx,ky,kz} */
           Vx = (1-kx*kx/k2)*Wx -    kx*ky/k2 *Wy -    kx*kz/k2 *Wz;
           Vy =   -ky*kx/k2 *Wx + (1-ky*ky/k2)*Wy -    ky*kz/k2 *Wz;
           Vz =   -kz*kx/k2 *Wx -    kz*ky/k2 *Wy + (1-kz*kz/k2)*Wz;
        /* store and ensure "mirror" symmetries */
           w[ix][iy][iz][it] = {Vx,Vy,Vz};
           w[(Nx-ix)%Nx][(Ny-iy)%Ny][(Nz-iz)%Nz][(Nt-it)%Nt]=
              {CC(Vx),CC(Vy),CC(Vz)};
        }
     }
/* "mirror" symmetries at the axis of the symmetry */
     for ( it,ix,iy,iz = 0,0,0,0 and Nt/2,Nx/2,Ny/2,Nz/2 ) {
        IM(vel[ix][iy][iz][it]) = 0.0;
     }
     X(v) = invFFT4(X(w));
     Y(v) = invFFT4(Y(w));
     Z(v) = invFFT4(Z(w));
     /* normalize each component of  v in the range [-1,+1] */
}
```

In the code we used the following macros:

`RE(c)` =	real part of complex variable
`IM(c)` =	complex part of complex variable
`CC(c)` =	complex conjugate of the variable
`X(v)` =	x component of a vector
`Y(v)` =	y component of a vector
`Z(v)` =	z component of a vector

Appendix B. Values of the Parameters Used in Our Animations

In all simulations `diff`=0, `s_mag` = 1.0, `s_rate` = 1.0. Dimension refers to the size of the primitive.

		computational grid					
Figure	`diss`	dimensions	resolution	`ex`	`alb`	`em`	`col`
7	0.0	1.55 1.48 1.34	32x32x32	11.6	0.9	0.0	(0.84 0.76 0.87)
8	0.0	1.95 1.83 1.88	40x40x32	2.1	1.0	0.0	(1.00 1.00 1.00)
9	0.5	4.13 3.71 1.68	32x32x32	11.8	0.9	0.7	(0.49 0.56 0.93)
10	0.0	1.00 0.96 1.51	16x16x32	6.5	1.0	0.0	(1.00 1.00 1.00)
11	0.5	4.14 3.71 1.68	32x32x32	11.8	0.7	0.2	(0.64 0.43 0.25)
12	0.0	1.00 0.96 1.30	40x40x40	18.7	1.0	0.0	(0.41 0.22 0.11)
13	0.0	1.00 1.00 1.00	40x40x40	1.3	1.0	0.0	(1.00 1.00 1.00)

Fields used in each animation along with their parameters (na=not applicable):

Figure	field type	`f_mag`	dimensions	time scaling	advected by
7	fractal turbulence	1.72	0.10 0.10 0.10	0.7	
	fractal turbulence	1.72	0.03 0.03 0.03	3.8	
8	directional	1.0	na	na	
	smooth turbulence	2.66	0.20 0.20 0.20	0.5	directional
	fractal turbulence	2.34	0.04 0.04 0.04	0.5	directional
9	directional	0.1	na	na	
	smooth turbulence	1.0	0.50 0.50 0.50	9.5	directional

	fractal turbulence	1.0	0.20 0.20 0.20	5.0	directional
10	directional	0.5	na	na	
	repulsive	6.4	0.40 0.40 0.40	na	
	fractal turbulence	2.66	0.15 0.15 0.15	8.2	directional
11	directional	2.0	na	na	
	repulsive	9.5	2.70 1.70 0.60	na	
	smooth turbulence	1.6	0.50 0.50 0.50	1.6	
	fractal turbulence	1.0	0.20 0.20 0.20	5.0	
12	directional	1.0	na	na	
	vortex	3.28	1.00 0.90 2.57	na	
	fractal turbulence	0.4	0.07 0.07 0.07	0.4	
13	directional	0.15	na	na	
	fractal turbulence	0.55	0.06 0.06 0.06	0.55	directional

References

[BLM95] B. Becker, D. A. Lane and N. Max. "Unsteady Flow Volumes". In *Proceedings of Visualization'95*. IEEE CS Press. Atlanta, Georgia, 1995.

[CMTM94] N. Chiba, K. Muraoka, H. Tajahashi and M. Miura. "Two-dimensional Visual Simulation of Flames, Smoke and the Spread of Fire". *The Journal of Visualization and Computer Animation*, vol. 5, 1994, pp. 37-53.

[Ebe94] D. Ebert. "Design and Animation of Volume Density Functions". *The Journal of Visualization and Computer Animation*, Vol. 4, 1993, pp. 1-20.

[EbPa90] D. Ebert and R. E. Parent. "Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques". *ACM Computer Graphics (SIGGRAPH'90)*, vol. 24, n. 4, August 1990, pp. 357-366.

[FFC82] A. Fournier, D. Fussell and L. Carpenter. "Computer Rendering of Stochastic Models", *Communications of the ACM*, 25 (1982), pp. 371-384.

[Gar94] G. Gardner, "Modeling Amorphous Natural Features". In *SIGGRAPH'94 Course Notes #8: Procedural Modeling, Texturing and Rendering Techniques*, July 1994.

[GLG95] M. N. Gamito, P. F. Lopes and M. R. Gomes. "Two-Dimensional Simulation of Gaseous Phenomena using Vortex Particles". In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*. Springer Verlag. 1995. pp. 3-15.

[LaLe94] P. Lacroute and M. Levoy. "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation". *Computer Graphics Proceedings (SIGGRAPH'94), Annual Conference Series*, 1994, pp. 451-458.

[MaBe95] N. Max and B. Becker. "Flow Visualization using Moving Textures". In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data.* September 1995.

[Max95] N. Max, "Optical Models for Direct Volume Rendering". *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, No. 2, June 1995.

[MCW92] N. Max, R. Crawfis and D. Williams. "Visualizing Wind Velocities by Advecting Cloud Textures". In *Proceedings of Visualization'92*, IEEE CS Press, Los Alamitos CA, October 1992, pp. 179-183.

[MoYa75] A. S. Monin and A. M. Yaglom. *Statistical Fluid Mechanics*. The MIT Press, Cambridge Massachusetts, 1975.

[NDN96] T. Nishita, Y. Dobashi and E. Nakamae. "Display of Clouds Taking Into Account Multiple Anisotropic Scattering and Sky Light". *Computer Graphics Proceedings (SIGGRAPH'96), Annual Conference Series*, 1996, pp. 379-386.

[NRC88] W. H. Press, B. P. Flannery, S.A. Teukolsky and W. T. Vetterling. *Numerical Recipes in C. The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1988.

[OGL92] J. Neider, T. Davis and M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL,* Release 1, Addison-Wesley, Reading Massachusetts, 1992.

[Per85] K. Perlin. “An Image Synthesizer”. *ACM Computer Graphics (SIGGRAPH’85),* vol. 19, n. 3, July 1985, pp. 287-296.

[Ree83] W. T. Reeves. “Particle Systems. A Technique for Modeling a Class of Fuzzy Objects”. *ACM Computer Graphics (SIGGRAPH’83)*, vol. 17, n. 3, July 1983, pp. 359-376.

[Rus94] H. Rushmeier. “Rendering Participating Media: Problems and Solutions from Application Areas”. In *Proceedings of the 5th Eurographics Workshop on Rendering.* Darmstadt, Germany. June 1994, pp. 35-56.

[Sak90] G. Sakas. “Fast Rendering of Arbitrary Distributed Volume Densities”. In *Proceedings of Eurographics ‘90,* Elsevier Science Publishers B.V. (North Holland), September 1990, pp. 519-530.

[Sak93] G. Sakas. “Modeling and Animating Turbulent Gaseous Phenomena Using Spectral Synthesis”. *The Visual Computer*, vol. 9, 1993, pp. 200-212.

[ShFo92] M. Shinya and A. Fournier. “Stochastic Motion - Motion Under the Influence of Wind”. In *Proceedings of Eurographics ‘92,* Elsevier Science Publishers B.V. (North Holland), September 1992, pp. 119-128.

[Sim90] K. Sims. “Particle Animation and Rendering Using Data Parallel Computation”. *ACM Computer Graphics (SIGGRAPH’90),* vol .24, n. 4, August 1990, pp. 405-413.

[Sim92] K. Sims. “Choreographed Image Flow”. *The Journal of Visualization and Computer Animation,* vol. 3, 1992, pp. 31-43.

[StFi93] J. Stam and E. Fiume. “Turbulent Wind Fields for Gaseous Phenomena”. *Computer Graphics Proceedings (SIGGRAPH’93), Annual Conference Series,* 1993, pp. 369-376.

[StFi95] J. Stam and E. Fiume. “Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes”. *Computer Graphics Proceedings (SIGGRAPH’95), Annual Conference Series*, 1995, pp.129-136.

[War94] G. Ward. “The RADIANCE Lighting Simulation and Rendering System”. *Computer Graphics Proceedings (SIGGRAPH’94), Annual Conference Series.* 1994, pp. 459-472.

[WeHa91] J. Wejchert and D. Haumann. “Animation Aerodynamics”. *ACM Computer Graphics (SIGGRAPH’91),* vol. 25, n. 4, July 1991, pp. 19-22.

[Wes90] L. Westover. “Footprint Evaluation for Volume Rendering”. *ACM Computer Graphics (SIGGRAPH’90),* vol. 24, n. 4, August 1990, pp.367-376.

[ZHH96] R. C. Zeleznik, K. P. Herndon and J. F. Hugues. “SKETCH: An Interface for Sketching 3D Scenes”. *Computer Graphics Proceedings (SIGGRAPH’96), Annual Conference Proceedings,* 1996, pp. 163-170.

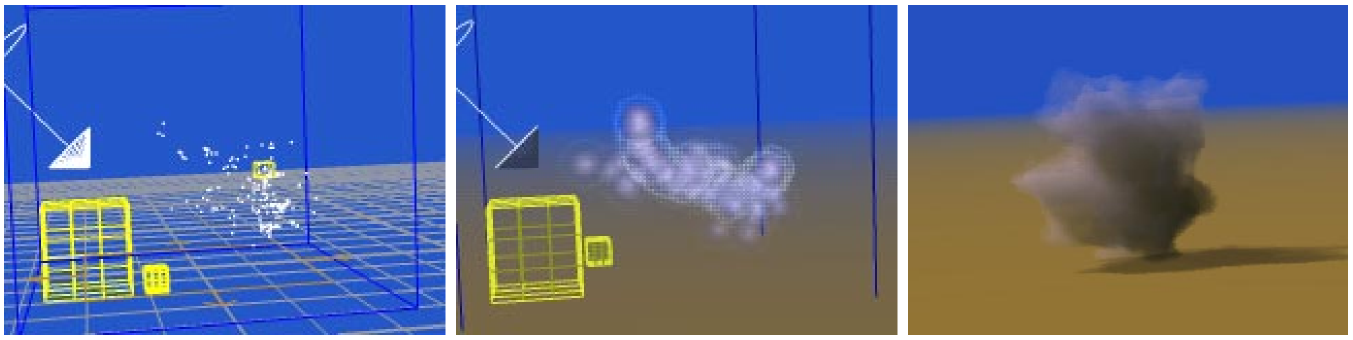

Figure 6: Three images showing the incremental design of an animation. The left image is a screen save of our system running in wire frame mode. The middle image is a screen save of our system working in solid drawing mode. Each gaseous blob is displayed as a "splat". The image on the right is created using our voxel tracer. The density of the gas is computed using our grid gas solver.

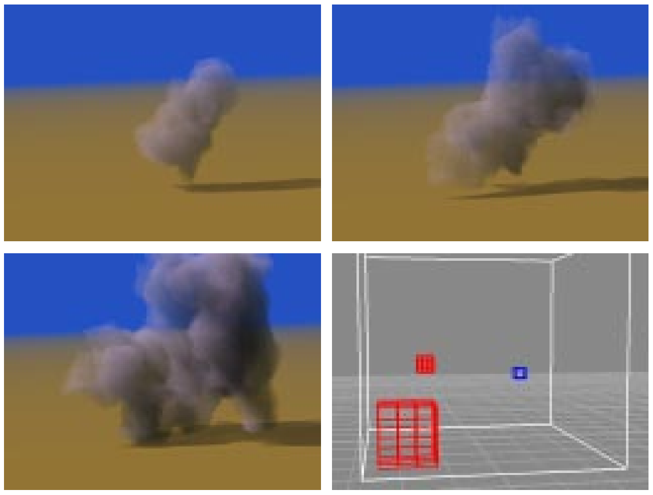

Figure 7: The gas emanating from a simple square source,in this example, is affected only by two turbulences with different scalings. The transparency of the gas is very low.

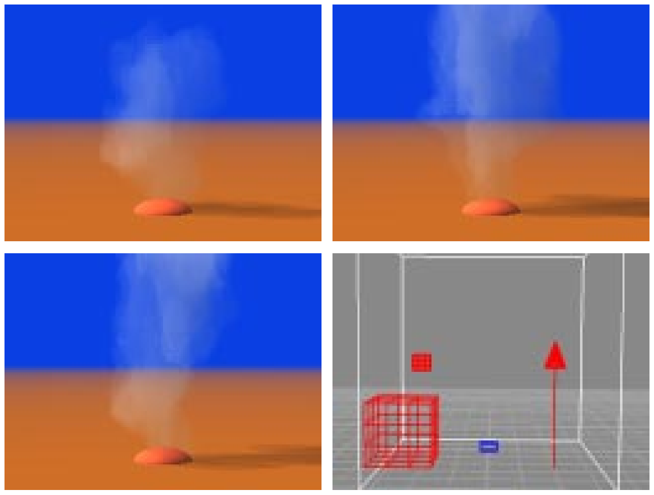

Figure 8: Steam−like gas. The heat field is approximated by a single directional field. Transparency of the gas is high.

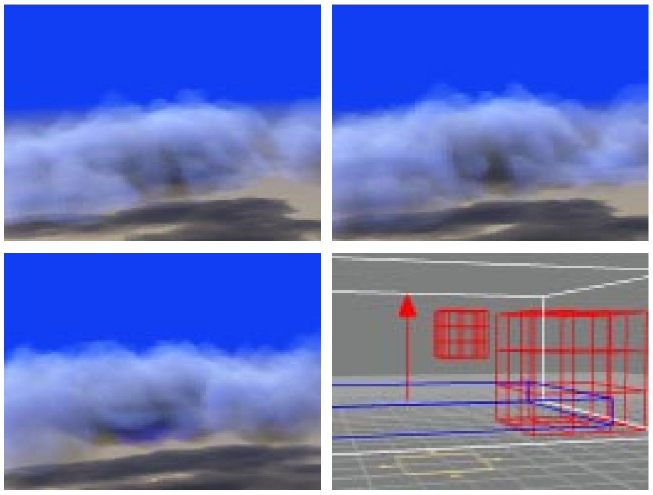

Figure 9: Animation of evolving clouds. The clouds are generated using a time−varying source. The clouds rise due to a global directional field. The albedo is high and the transparency is low. Multiple scattering is emulated by adding some blueish self−emission.

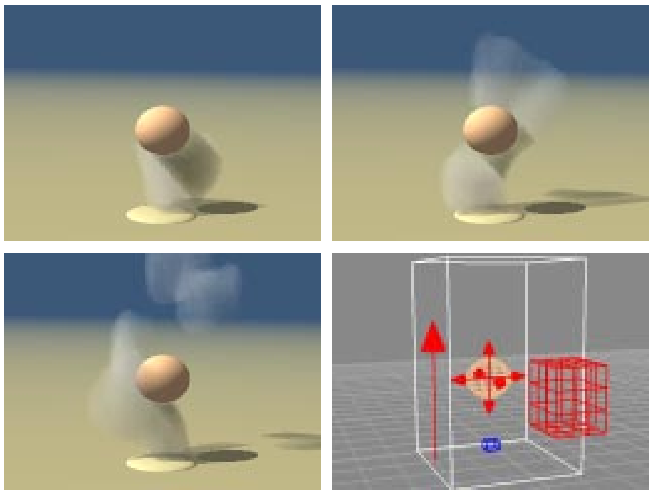

Figure 10: This example depicts a gaseous flow around a simple spherical object. The repulsive influence of the sphere is emulated by centering a repulsive point field at the object.

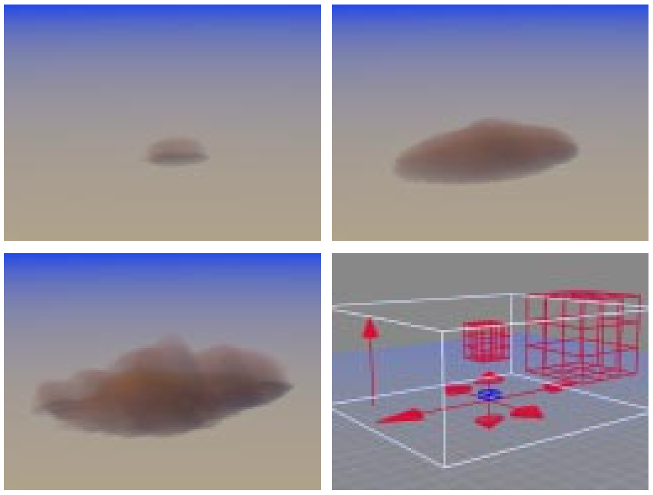

Figure 11: Animation of a very simple "explosion−like" behavior. The source is modulated by a short impulse. The spread is modeled using a single repulsivepoint field.

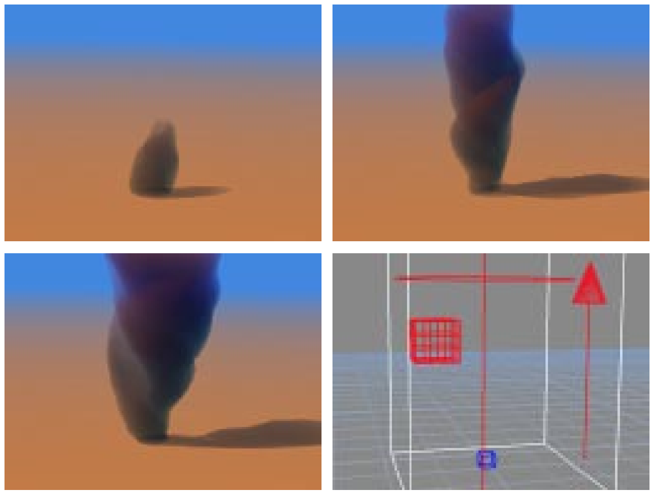

Figure 12: Animation of a "baby twister". The twisting of the gas is modeled using a single directional vortex field centered on the source.

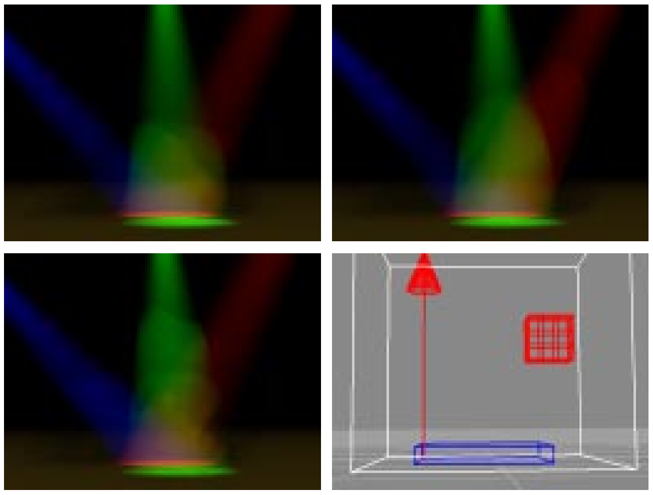

Figure 13: Rising smoke illuminated by three spotlights with different colors.

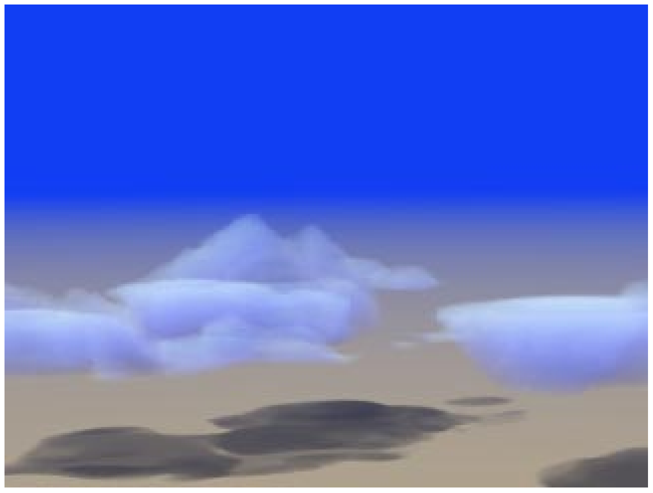

Figure 14: Frame from a higher resolution animation of the clouds.

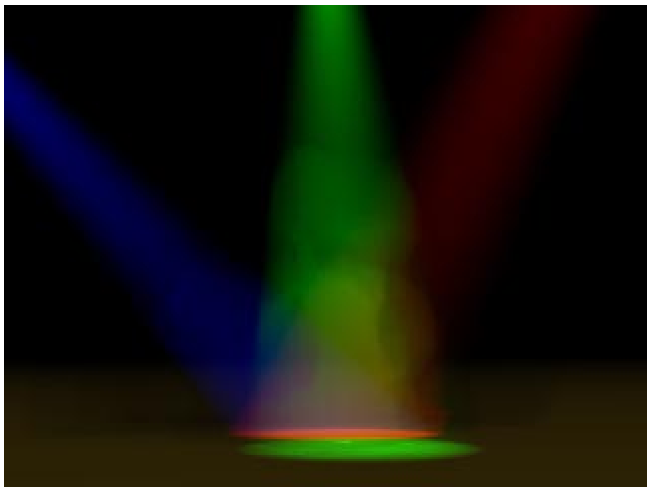

Figure 15: Higher resolution picture of the colored gas.

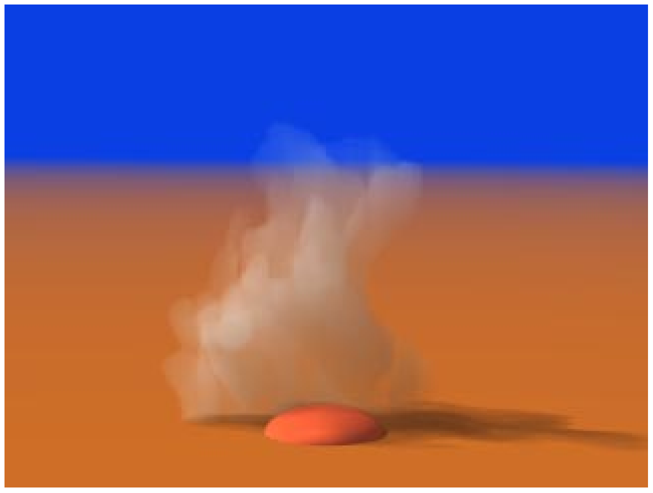

Figure 16: Higher resolution picture of the steam animation.