

Simulation of Cloud Dynamics on Graphics Hardware

Mark J. Harris William V. Baxter III Thorsten Scheuermann[†] Anselmo Lastra

Department of Computer Science, University of North Carolina at Chapel Hill, North Carolina, USA[‡]

Abstract

This paper presents a physically-based, visually-realistic interactive cloud simulation. Clouds in our system are modeled using partial differential equations describing fluid motion, thermodynamic processes, buoyant forces, and water phase transitions. We also simulate the interaction of clouds with light, including self-shadowing and light scattering.

We implement both simulations – dynamic and radiometric – entirely on programmable floating-point graphics hardware. We use “flat 3D textures” – 3D data laid out as slices tiled in a 2D texture – to implement 3D simulations on the GPU. This has scalability advantages over the use of traditional 3D textures. We exploit the relatively slow evolution of clouds in calm skies to enable interactive visualization of the simulation. The work required to simulate a single time step is automatically spread over many frames while the user views the results of the previous time step. This technique enables the incorporation of our simulation into real applications without sacrificing interactivity.

Keywords: Clouds; Graphics hardware, Physically-based simulation, Light scattering, Fluid dynamics.

1 Introduction

Clouds are a ubiquitous feature of our world. They provide a fascinating dynamic backdrop to the outdoors, creating an endless array of formations and patterns. They are also an integral factor in the behavior of Earth’s weather systems. The combination of physical and visual complexity has made them an important area of study for meteorologists, physicists, and even artists.

Clouds can form in many ways. Convective clouds form when moist air is warmed and becomes buoyant. The air rises, carrying water vapor with it, expanding and cooling as it goes. As the temperature and pressure of the air decrease, its saturation point – the equilibrium level of evaporation and condensation – is reduced. When the water vapor content of the rising air becomes greater than its saturation point, condensation occurs, which yields the microscopic condensed cloud water particles that we see as clouds in the sky. Condensation increases the drag on the air, causing it to slow its ascent, which creates a natural limit on the vertical extent of a cloud layer. Stratus clouds usually form when masses of warm and cool air mix due to radiative cooling or lifting of the air over terrain (Orographic lifting). An example of the formation of stratus clouds by mixing is the fog that often rolls into the city of San Francisco.

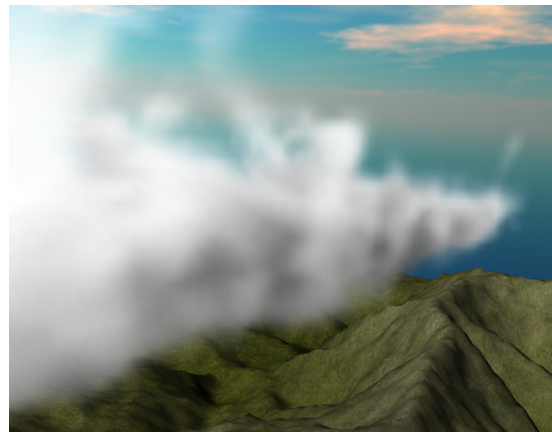


Figure 1: Simulated cumulus clouds roll above a valley.

We have developed a cloud dynamics simulation based on partial differential equations that model fluid flow, thermodynamics, and water condensation and evaporation, as well as various forces and other factors that influence these equations. We implement the discrete form of these equations using the programmable floating-point fragment processors in the latest graphics hardware. All computation and rendering is performed on the GPU; the CPU provides only high-level control. We describe two useful optimizations for this hardware: a representation of volume data in two-dimensional textures, and an efficient packing

[†] Now at ATI Research, Inc., Marlborough, MA. (thorsten@ati.com)

[‡] {harrism, baxter, scheuerm, lastra}@cs.unc.edu

of scalar fields to best exploit the vector operations of the fragment processor.

In order to incorporate dynamically simulated 3D clouds into existing 3D applications, the per-frame simulation cost must be small enough that it does not interfere with the application. We have implemented a method for amortizing the simulation cost over multiple frames allowing the application to budget time for the simulation each frame. This technique greatly improves interactivity, allowing us to view the results of a simulation at 60 frames per second or higher while the simulation progresses at several iterations per second.

For clouds to look realistic, we must also simulate their interaction with light. We use an illumination approximation that incorporates self-shadowing and multiple forward light scattering, and implement it on the GPU using dynamic 3D texturing. We have integrated all of the techniques described above in an interactive flight application [Harris and Lastra 2001] that amortizes the high cost of slice-based volume cloud rendering by using dynamically-generated impostors.

Others in computer graphics have researched methods of simulating clouds. [Kajiya and Von Herzen 1984] used a simple method based on PDEs to generate cloud data sets for their ray tracing algorithm. [Dobashi, et al. 2000] used a simple cellular automata model of cloud formation to animate clouds offline. [Miyazaki, et al. 2001] extended this to use a coupled map lattice model based on atmospheric fluid dynamics. [Overby, et al. 2002] described another physical model that, like ours, is based on the stable fluid simulation of [Stam 1999]. Of these, our



Figure 2: A sequence of stills (top-to-bottom, left-to-right) from our 2D cloud simulation, running on a 128x128 grid at greater than 30 frames per second.

work is most similar to the work by Kajiya and Von Herzen and Overby et al. However, there are several differences.

Both Overby et al. and Kajiya and Von Herzen use a buoyancy force that is proportional to potential temperature. Our model also accounts for the negative buoyancy effects of condensed water mass and the positive buoyancy effects of water vapor [Houze 1993]. This increases the realism of air currents. Overby et al. also assume that saturation is directly proportional to pressure, but they provide no information about how they model pressure in their system. Our system uses a well-known exponential relationship between saturation and temperature, and does not explicitly model pressure. In addition, they introduce two effects that are physically unrealistic. One is a computation meant to account for the expansion of rising air. The other is an artificial momentum-conservation computation. These computations are superfluous since the Navier-Stokes equations, which they solve, already account for these phenomena. Overby et al. were able to achieve rates of a few frames a second. However we are able to simulate on larger volumes at interactive rates due to the speed of the graphics hardware. Finally, none of the previous simulations have been integrated into truly interactive, high frame rate applications.

2 Cloud Dynamics

The dynamics of cloud formation, growth, motion and dissipation are complex. In the development of a cloud simulation, it is important to understand these dynamics so that good approximations can be chosen that allow efficient implementation without sacrificing realism. In this section, we describe the equations of cloud dynamics that make up our model. For much more detailed information and analysis, we refer the reader to [Andrews 2000; Houze 1993; Rogers and Yau 1989].

The basic quantities necessary to simulate clouds are velocity, $\mathbf{u}=(u, v, w)$, air pressure, p , temperature T , water vapor, q_v , and condensed cloud water, q_c . These water content variables are *mixing ratios* – the mass of vapor or liquid water per unit mass of air. It is the condensed water, q_c , that makes clouds visible, so this is the desired output of our simulation. We require a system of equations that models cloud dynamics in terms of these variables. These equations are the equations of motion, the thermodynamic equation, and the water continuity equations.

2.1 Equations of motion

The motion of air in the atmosphere can be described by the incompressible Euler equations of fluid motion:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u} - \frac{1}{\rho} \nabla p + \mathbf{B} \hat{\mathbf{k}} + \mathbf{f} \quad (1)$$

$$\nabla \cdot \mathbf{u} = 0 \quad (2)$$

where ρ is the density of the fluid. Equation (1) is a statement of the conservation of momentum in which the first term on the right expresses how the velocity field transports, or *advects* itself, the second term is an

acceleration caused by the pressure gradient, \mathbf{B} is buoyant acceleration, and \mathbf{f} is acceleration due to other forces. Equation (2) is known as the *continuity equation*, because it enforces that the velocity field is divergence-free, and conserves mass. In addition to the advection of velocity, temperature and water (in both phases) are also advected by the flow, as will be described below.

2.2 Parcels and Potential Temperature

A conceptual tool used in the study of atmospheric dynamics is the *air parcel* – a small mass of air that can be thought of as “traceable” relative to its surroundings. The parcel approximation is useful in developing the mathematics that our simulation requires.

When a parcel changes altitude without a change in heat, it is said to move *adiabatically*. Since air pressure (and therefore temperature) varies with altitude, the parcel’s pressure and temperature will change. We can account for adiabatic changes of temperature and pressure with the concept of potential temperature. The potential temperature, θ , of a parcel of air can be defined as the final temperature that a parcel would have if it were moved adiabatically from pressure p and temperature T to pressure \hat{p} (standard pressure at sea level, ~ 100 kPa):

$$\theta = \frac{T}{\Pi} = \left(\frac{\hat{p}}{p}\right)^\kappa T, \quad (3)$$

$$\kappa = \frac{R_d}{c_p} = \frac{c_p - c_v}{c_p} \approx 0.286.$$

Π is called the Exner function. R_d is the gas constant for dry air ($287 \text{ J kg}^{-1} \text{ K}^{-1}$), and c_p and c_v are the specific heat capacities of dry air at constant pressure and volume, respectively. Potential temperature is convenient to use in atmospheric modeling because it is constant under adiabatic changes of altitude, while absolute temperature must be recalculated at each altitude.

2.3 Buoyant Force

Changes in the density of a parcel of air relative to its surroundings result in a buoyant force on the parcel. If the parcel’s density is less than the surrounding air, this force will be upward; if the density is greater, the buoyant force will be downward. The density of an ideal gas is related to its temperature and pressure. A common simplification in cloud modeling is to regard the effects of local pressure changes on density as negligible, so we can represent this buoyant force per unit mass with the following expression

$$\mathbf{B} = g \left(\frac{\theta_v}{\theta_{v,0}} - q_H \right), \quad (4)$$

where g is the acceleration due to gravity and q_H is the mass mixing ratio of *hydrometeors*, which includes all forms of water other than water vapor. In the case of the simple two-state bulk water continuity model to be given in Section 2.6, this is just the mixing ratio of liquid water, q_c . $\theta_v \approx \theta(1 + 0.61q_v)$ is the *virtual potential temperature*, which accounts for the effects of water vapor on air

temperature, and is defined as the potential temperature that dry air would have if its pressure and density were equal to those of a given sample of moist air. $\theta_{v,0}$ is the reference potential temperature, usually between 290 and 300 K. While the difference between virtual and potential temperature may seem negligible, it is possible to noticeably increase buoyant force by increasing only the water vapor content of the air.

2.4 Environmental Lapse Rate

The Earth’s atmosphere is in static equilibrium. The *hydrostatic balance* of the opposing forces of gravity and air pressure results in an exponential decrease of pressure with altitude:

$$p(z) = p_0 \left(1 - \frac{z\Gamma}{T_0} \right)^{g/(\Gamma R_d)} \quad (5)$$

Here, z is altitude, and p_0 and T_0 are the pressure and temperature at the base altitude. Typically, $p_0 = 100$ kPa and T_0 is in the range 280–310 K. The *lapse rate*, Γ , is the rate of decrease of temperature with altitude. In the Earth’s atmosphere, temperature decreases approximately linearly with height in the troposphere (sea level to about 15 km, the tropopause). Therefore, we can assume that Γ is a constant. A typical value for Γ is around 10 K km^{-1} . We can use (3) and (5) to compute the environmental temperature and pressure of the atmosphere in the absence of disturbances, and as we describe below, compare them to the local temperature and pressure to compute the saturation point of the air.

2.5 Saturation Mixing Ratio

Cloud water continuously changes phases from liquid to vapor and vice versa. When the rates of condensation and evaporation are equal, air is said to be *saturated*. The water vapor mixing ratio at saturation is called the *saturation mixing ratio*, denoted by $q_{vs}(T,p)$. When the water vapor mixing ratio exceeds the saturation mixing ratio, the air is *supersaturated*. Rather than remain in this state, condensation may occur, leading to cloud formation. A useful empirical approximation for saturation mixing ratio is

$$q_{vs}(T, p) = \frac{380.16}{p} \exp\left(\frac{17.67T}{T + 243.5}\right), \quad (6)$$

with T in Celsius and p in Pa. This is based on the formula for a curve fit to data in standard meteorological tables to within 0.1% over the range $-30^\circ\text{C} \leq T \leq 30^\circ\text{C}$ [Rogers and Yau 1989].

2.6 Water Continuity

We use a simple *Bulk Water Continuity* model as described in [Houze 1993] to describe the evolution of water vapor mixing ratio q_v and condensed *cloud water* mixing ratio, q_c . Cloud water is water that has condensed but whose droplets have not grown large enough to precipitate. The water mixing ratios at a given location are affected both by advection and by phase changes (from gas to liquid and

vice versa). In this model, the rates of evaporation and condensation must be balanced, resulting in the water continuity equation,

$$\frac{\partial q_v}{\partial t} + (\mathbf{u} \cdot \nabla) q_v = - \left(\frac{\partial q_c}{\partial t} + (\mathbf{u} \cdot \nabla) q_c \right) = -C, \quad (7)$$

where C is the rate of condensation.

2.7 Thermodynamic Equation

While adiabatic motion is a valid approximation for air that is not saturated with water vapor, the potential temperature of saturated air cannot be assumed to be constant. If expansion of a moist parcel continues beyond the saturation point, water vapor condenses and releases latent heat, warming the parcel. If latent heating and cooling due to condensation and evaporation are the only non-adiabatic heat sources, then the first law of thermodynamics results in

$$\frac{\partial \theta}{\partial t} + (\mathbf{u} \cdot \nabla) \theta = \frac{-L}{c_p \Pi} \left(\frac{\partial q_v}{\partial t} + (\mathbf{u} \cdot \nabla) q_v \right), \quad (8)$$

where L is the latent heat of vaporization of water, 2,501 J kg⁻¹ at 0° C (Changes by less than 10% within ±40°). Notice from (7) that we can substitute $-C$ for the quantity in parentheses above. This equation states that the change in local potential temperature is determined both by advection of potential temperature in and out of the local region, and by the latent heat of local phase changes.

2.8 Vorticity Confinement

Like the smoke that was the simulation goal of [Fedkiw, et al. 2001], convective clouds typically contain rotational flows at a variety of scales. As they explained, numerical dissipation caused by simulation on a coarse grid damps out these interesting features. Therefore like Fedkiw et al., we use *vorticity confinement* to restore these fine-scale motions. [Overby, et al. 2002] also used vorticity confinement in their cloud simulation.

Vorticity confinement works by first computing the vorticity $\omega = \nabla \times \mathbf{u}$, from which a normalized vorticity vector field

$$N = \frac{\eta}{|\eta|}, \quad \text{where } \eta = \nabla |\omega| \quad (9)$$

is computed. The vectors N point from areas of lower vorticity to areas of higher vorticity. From these vectors we can compute a force that can be used to replace dissipated vorticity back in:

$$\mathbf{f}_{vc} = \varepsilon h (N \times \omega). \quad (10)$$

Here ε is a user-controlled scale parameter and h is the grid scale.

3 Solving the Equations

From the description above, we see that we must solve the equations of fluid flow, (1) and (2), the water continuity equation, (7), and the thermodynamic equation, (8).

3.1 Fluid Flow

Our cloud model is based on the equations of fluid flow, so our simulator is built on top of a standard fluid simulator much like the ones described by [Fedkiw, et al. 2001; Stam 1999]. We solve the equations of motion using the stable two step technique described in those papers.

In the first step, we use the semi-Lagrangian advection technique that Stam described to compute an intermediate velocity field \mathbf{u}' , and add to it the buoyancy force, (4), and vorticity confinement force, (10). This step solves equation (1) without the pressure term. In the second step, the intermediate field \mathbf{u}' is made incompressible (so that it satisfies both (1) and (2)) using a projection method based on the Helmholtz-Hodge decomposition [Chorin and Marsden 1993]. The projection is performed by solving for the pressure using the Poisson equation

$$\nabla^2 p = \frac{1}{\delta t} \nabla \cdot \mathbf{u}' \quad (11)$$

with pure Neumann boundary conditions ($\partial p / \partial n = 0$), and then subtracting the pressure gradient from \mathbf{u}' :

$$\mathbf{u} = \mathbf{u}' - \delta t \nabla p. \quad (12)$$

Using the advection technique as for velocity, we also advect the temperature, θ , and water variables, q_v and q_c . During advection, we apply different boundary conditions for each of the variables. For the velocity, we use the no-slip condition ($\mathbf{u} = \mathbf{0}$) on the bottom boundary, and free-slip condition ($\partial \mathbf{u} / \partial n = \mathbf{0}$) on the top. At the sides, we set the vertical velocity to zero, and the horizontal velocities to the user-defined horizontal wind speeds. We set the top and side temperature boundaries to the user-defined ambient temperature. We use periodic side boundaries for q_v to simulate water vapor being blown in from outside our simulation domain, and we set all q_c boundaries and the top q_v boundary to zero. Finally, we specify input fields at the bottom boundary for both temperature and water vapor. These fields are randomly perturbed, user-specified constant values, and are the source of the temperature and water that cause clouds to form.

3.2 Water Continuity

The solution of the water continuity equations (7) is straightforward. The equations state that the changes in q_v and q_c are governed by advection of the quantities as well as by the amount of condensation and evaporation. We solve them in two steps. First, we advect each using the semi-Lagrangian technique mentioned before, resulting in intermediate values q'_v and q'_c . Then, at each cell, we compute the new mixing ratios as follows:

$$\begin{aligned} \Delta q'_v &= -\Delta C = \min(q_{vs} - q'_v, q'_c) \\ q_v &= q'_v + \Delta q'_v \\ q_c &= q'_c - \Delta q'_v, \end{aligned} \quad (13)$$

where ΔC is the amount of condensation over the time step, and q_{vs} is computed using equation (6) as described in Section 2.5. We compute T using equation (3) with the

current potential temperature θ , and the local environmental pressure computed with equation (5).

3.3 Thermodynamics

The left-hand side of the thermodynamic equation, (8), shows that like the other quantities, potential temperature is advected by the velocity field, so we compute an intermediate value, θ' via the semi-Lagrangian advection scheme. As mentioned before, we can substitute $-C$ for the quantity in parentheses on the right-hand side of the thermodynamic equation. This means that the temperature increases by an amount proportional to the amount of condensation, and we can update it as follows:

$$\theta = \theta' + \frac{L}{c_p \Pi} \Delta C \quad (14)$$

4 Implementation

We solve the equations presented in the previous section on a grid of voxels. We use a *staggered grid* discretization of the velocity and pressure equations as in [Fedkiw, et al. 2001; Foster and Metaxas 1997; Griebel, et al. 1998]. This means that pressure, temperature, and water content are defined at the center of voxels while velocity is defined on the faces of the voxels. Not only does this method reduce numerical dissipation as mentioned by Fedkiw et al., but as Griebel et al. explain, it prevents possible pressure oscillations that can arise with collocated grids (in which all variables are defined at cell centers). Our experiments with collocated grids have indeed shown some undesirable pressure oscillations when buoyant forces are applied. Section 5.2 describes our implementation of voxel grids using textures.

Overall, our method for solving the equations of cloud dynamics at each discrete time step is as follows.

1. Advect θ , q_v , and q_c and velocity, \mathbf{u} .
2. Compute vorticity confinement force, \mathbf{f}_{vc} .
3. Compute buoyant force, \mathbf{B} .
4. Compute $\mathbf{u}' = \mathbf{u}_{advected} + (\mathbf{B} + \mathbf{f}_{vc}) \cdot \delta t$.
5. Update q_v and q_c according to (13).
6. Update θ according to (14).
7. Compute the divergence $\nabla \cdot \mathbf{u}'$.
8. Solve the Poisson-pressure equation, (11).
9. Compute $\mathbf{u} = \mathbf{u}' - \nabla p$.

Our implementation of steps 1, 2, 7, and 9 follows [Fedkiw, et al. 2001] nearly exactly. We refer the reader to the appendix of that paper for the discrete form of the equations. Step 3 can be implemented directly from equation (4). However we find that providing the user with a scale factor applied to q_H provides useful control over the buoyancy of clouds. In our implementation, steps 5 and 6 are performed in a single fragment program (See Section 5.1), since we store the water and temperature variables in a single texture. We solve the remaining step, the Poisson-pressure equation, using a standard iterative relaxation solver applied to equation (11).

[Fedkiw, et al. 2001] use the conjugate gradient method with an incomplete Choleski preconditioner to solve the Poisson-pressure equation. While this is a straightforward solver to implement and run on a CPU, our implementation uses fragment programs that run on the GPU. Implementation of conjugate gradient on the GPU is feasible ([Bolz, et al. 2003; Krüger and Westermann 2003]), but many passes are required just to compute the large vector inner products required by the algorithm ($O(\log_2 N)$ passes, where N is the grid resolution). Because of this, we chose to use a simple solver such as Jacobi or Red-Black Gauss Seidel relaxation [Golub and Van Loan 1996]. These solvers can be implemented to run in one and two render passes, respectively, using only a few lines of Cg code. Therefore we can run more iterations in a given amount of time than with a more complex method, which helps make up for the slower convergence of our chosen solvers. Section 5.3 discusses the efficient implementation of these solvers.

5 Hardware Implementation

As mentioned before, we perform all of the numerical computation for our cloud simulator in the programmable, floating point fragment unit of a graphics processor. State fields, such as p and \mathbf{u} , are stored in textures. For efficiency, we store θ , q_v , and q_c in different channels of the same texture. Computation is performed much as in [Harris, et al. 2002]. The difference is that newer GPUs provide much more power in terms of precision, flexibility and instruction count, allowing us to tackle much more complex simulations, such as clouds. State textures are updated using a render pass that draws a quadrilateral fit to the viewport. We implement computations using fragment programs written in the Cg shading language [Mark, et al. 2003]. The fragment programs implement the steps described in the previous section using texturing operations to read data from the grids.

At the end of a render pass, the state texture is updated. This update can be performed via a copy from the frame buffer to the texture, or via “render to texture”. Render to texture requires that two textures be kept for each state field, and swapped after each update.

5.1 Interior and Boundary Computation

In a typical CPU implementation of fluid simulation, the simulation domain is represented in an array. Many simulation steps require different computations on the interior of the simulation domain than on its boundaries, so usually a single row of cells on the outside of the domain is reserved to store boundary values [Griebel, et al. 1998]. To perform a given simulation step, such an implementation will typically iterate over the domain using a set of nested loops, and then update the boundary values separately.

Hardware simulation is very similar. The SIMD nature of the fragment processor means that the render-pass idiom described above is equivalent to the nested-loops-over-an-array idiom of CPU simulation. The borders of a texture contain boundary values, and different computations (thus different fragment programs) must be executed on the

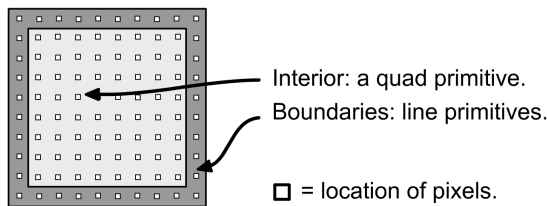


Figure 3: Updating a state field involves rendering a quad for the interior and a line for each border. Different fragment programs are applied to interior and border fragments.

border and interior. To implement this, we activate a boundary fragment program and render line primitives over the edges of the view port. Then we activate an interior fragment program and render a quadrilateral that covers all but the outer single-pixel border. This is illustrated in Figure 3.

5.2 Flat 3D Textures

Previous methods for 3D simulation on GPUs have used 3D textures or a stack of 2D textures to represent the grid [Harris, et al. 2002]. To apply a simulation operation to the grid – for example to compute the buoyancy force – the volume must be updated slice by slice. At each slice, the operation is applied, and the texture for that slice is updated, requiring a texture copy or a context switch associated with render to texture.

We instead represent our grids using what we call a “flat” 3D texture. A flat 3D texture is actually a 2D texture that contains the tiled slices of a 3D volume, as shown in Figure 4. In the figure, the dark grey borders represent the boundary cells of each slice, and the light grey boxes in the lower left and upper right represent the boundary slices of the volume along the slicing axis. Updating a flat 3D texture is much like the 2D texture update described in the previous section. We render the interior of each slice (the colored squares) as a quad primitive, and we render the boundaries using line primitives. One fragment program is used for all of the interior quad primitives, and another is used for the boundary lines and the two “end cap” (lower left and upper right) quads.

The advantage of true 3D textures over flat 3D textures is that addressing them is easy, since the GPU supports it. With flat 3D textures, however, we must convert the R texture coordinate into a 2D offset in order to do a texture lookup. We do this in our fragment programs. In practice, we precompute a 1D lookup texture that contains the offsets for each slice, and use this as an indirection table indexed by the Z coordinate.

Flat 3D textures can be updated in a single render pass – only one texture update is required for the entire volume. This means that a 3D simulation can be implemented in the same number of passes (ignoring changes in the computation itself) as an equivalent 2D simulation. We find that flat 3D textures provide a performance advantage over true 3D textures on current hardware. While the amount of data copied or updated is the same, we find that the total slice update overhead is much greater for true 3D

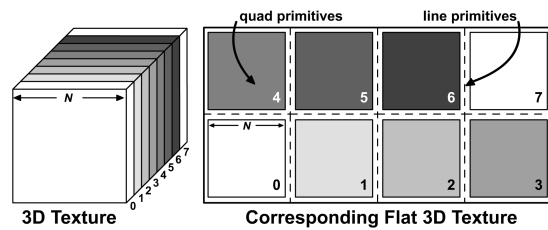


Figure 4: Flat 3D textures tile the slices of a 3D volume onto a 2D texture. This allows all slices of the volume to be updated in a single rendering pass.

textures. Also, flat 3D textures provide a quick, inexpensive way to preview the results of a 3D simulation, since they can be easily rendered as a 2D image.

5.3 Vectorized Iterative Solvers

Iterative solution of the Poisson equation for pressure is one of the most expensive operations in our numerical cloud simulation. For simulation on the GPU, the choice of solver is limited by the inability of fragment programs to both read and write the same memory in the same pass. This rules out Gauss-Seidel and Successive Over-relaxation (SOR), which have been used in many previous graphics applications of fluid simulation. We have, however, implemented several solvers and conducted an investigation to determine the most efficient of these given the constraints of graphics hardware. The results are given in Table 1.

Our Jacobi solver stores pressure as a single-channel floating point texture. The Jacobi fragment program also takes a divergence texture as input, and computes an updated pressure value as output for each fragment by sampling neighboring pressure values and subtracting the input divergence. After this single pass the output texture is used as input for the next iteration of the solver.

Red-Black Gauss-Seidel is a variation on Jacobi that splits the cells into two sets such that new red cell values only depend on black cell inputs, and vice versa (see Figure 5). All the red values can be updated using only the old black values, and then the black values can be updated using the new red values. Using the more recent red values for half of the texel updates improves the convergence rate.

To implement Red-Black efficiently we pack four pressure values into a single RGBA texel, as shown in Figure 5. This allows us to reduce the overall number of texture lookups required for each half-pass of Red-Black. Without packing, the Red pass would require 5 texture lookups per pressure update (corresponding to the 5 cells of a 5-point discrete Laplacian) in 2D, and 7 lookups in 3D. By packing pressure in a vectorized format the same 5 or 7 texture lookups enable us to update 4 pressure values instead of just one, as can be seen in Figure 5. With the 5 samples shown the 4 pressure values in $g_{i,j}$ can be updated. This is a significant savings; however, we incur extra overhead since the Black cells must be explicitly passed through on a Red pass and vice versa, so that both Red and Black are written every pass, and we also incur the overhead of two rendering passes for one solver iteration.

Poisson Solver	Convergence per msec ¹	Convergence ²	Time ³ (msec)
Jacobi 2D	0.50	0.078	45.9
Red-Black 2D	0.85	0.124	45.3
Vectorized Jacobi 2D	1.0	0.079	17.3
Jacobi 3D	N/C	N/C	110 [†]
Vectorized Jacobi 3D	N/C	N/C	49.0

¹Normalized convergence per millisecond achieved with a 17 ms time budget. ²Relative convergence after 100 iterations. ³Time for execution of 100 iterations. [†]Estimated from register usage, program length, and fragment count for 3D Jacobi program.

Table 1: A comparison of the convergence rates of various iterative solvers running on the GPU. All grids were 128x128 or 32x32x16. ("N/C" = "not computed".)

Still, Red-Black converges faster than a basic Jacobi solver given a fixed time budget.

But the same vectorization technique used to accelerate the Red-Black solver can also be used to accelerate the Jacobi solver. With vectorization, two full Jacobi iterations take less time than a full Red-Black iteration, and give better convergence. The result is that vectorized Jacobi gives the best convergence under a fixed time budget of any of the hardware solvers we have tested (See Table 1).

6 Interactive Applications

Cloud simulation is a very computationally intensive process, and is therefore usually done offline. But simulations of phenomena such as clouds have the potential to provide rich dynamic content for interactive applications, so one of our goals in this work has been to create a simulation that will work well online.

As a test of this, we have integrated our cloud simulation into our *SkyWorks* cloud rendering engine [Harris and Lastra 2001]. *SkyWorks* was designed to render scenes full of static clouds very fast. It precomputes the illumination of the clouds, and then uses this illumination to render the clouds at run time. To amortize the cost of rendering the clouds, it uses dynamically generated impostors [Schaufler 1995].

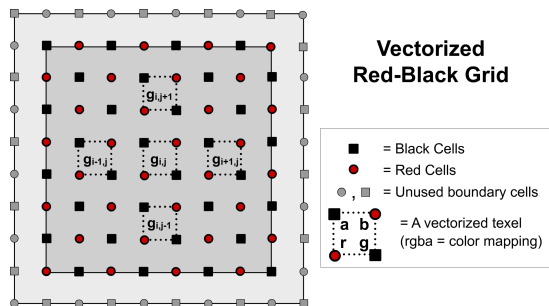


Figure 5: We implement efficient iterative solvers by packing multiple scalar values into each texel. The cross pattern demonstrates the sampling for vectorized Jacobi and Red-Black iteration.

©The Eurographics Association 2003.

6.1 Simulation Amortization

In order to incorporate dynamically simulated 3D clouds into existing 3D applications such as *SkyWorks*, the per-frame simulation cost must be small enough that it does not interfere with the application. If we were to perform a complete simulation time step every frame, our application's frame rate would drop below the frame rate of the cloud simulation. As an example, with a volume of resolution 64^3 , we simulate at under four iterations per second. This is not an acceptable frame rate for a flight simulator.

To avoid this problem, we have built into our simulation system a method for automatically dividing the work of a simulation time step over multiple frames. This is fairly straightforward to do with a GPU simulation because each operation is a render pass. We have instrumented our simulator with the ability to measure the time taken by any render pass. Every so often (usually just at startup and at the user's request) we run a complete simulation step with these timers active, and we record the time for each step. Then, in each frame of the application, the application budgets a certain amount of time for the simulation, and the simulator attempts to stay as close to that budget as possible.

We have found that this technique makes a tremendous difference in the performance of our application. We can fly around and through dynamic clouds at 40-80 frames per second while the simulation updates 1-5 times per second. Since our simulation time step can be set at a few seconds, we can have clouds that update in approximately "real time".

Still, this system is not perfect, because it is very difficult to accurately time an operation in the graphics pipeline. In order to get the best interactivity, we must be sure the simulator rarely goes over budget. To do this, we try to get worst case timings for each operation by forcing the pipeline to flush before we stop the timer. But this is not realistic, because under normal conditions (i.e. without forced flushes) there is more parallelism in the GPU. Therefore, a better method – perhaps with hardware support – of timing GPU operations would be useful.

7 Cloud Rendering

To render simulated clouds in *SkyWorks*, we convert the simulation's current cloud water texture into a true 3D texture, which is then used to render the cloud for multiple frames. We found that rendering directly from the flat 3D texture is too expensive, because of the added cost of the fragment program required to read the texture with 3D coordinates. Since a simulation time step does not complete every frame anyway, we find that the conversion is overall much faster. Also, the generation of the 3D texture (which is performed entirely on the GPU) is included in the simulation amortization, so that it doesn't affect our interactive frame rates. We use the impostors provided by *SkyWorks* as-is (one impostor for the entire simulation grid), and found that the rendering speed advantage impostors provide for static clouds transfers well

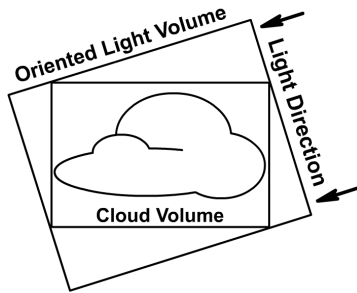


Figure 7: We use a 3D texture oriented along the light direction to compute and cloud illumination in hardware.

to our dynamic clouds, since the clouds remain static for several frames at a time.

7.1 Cloud Illumination

To create realistic images of clouds, we must account for the complex nature of their interaction with light. Light that reaches your eyes from a cloud has been scattered many times by the tiny water droplets in the cloud. This is what gives clouds their soft, diffuse appearance. A full simulation of multiple scattering requires the solution of a double-integral equation. However, cloud water droplets scatter most strongly in the direction of travel of the incident light, or forward direction. [Harris and Lastra 2001] used this fact to derive a computationally inexpensive model that simulates *multiple forward scattering*. Their algorithm, based on the shadowing algorithm of [Dobashi, et al. 2000], applied the approximation to precompute illumination of cloud particles using frame buffer blending and read back.

We use an extension of this algorithm that works with 3D textures instead of particles. Like the previous algorithms, it is a two pass algorithm that computes a 3D illumination texture, and then uses this illumination texture to illuminate a 3D density texture. The algorithm works as follows.

Tightly fit a bounding box to the bounding box of the cloud density volume, oriented so that the Z axis of the bounding box is aligned with the forward light direction. Traverse the light volume, rendering N slices, where N is the resolution of the light texture. Set the blending function and polygon colors as in [Harris and Lastra 2001] to compute shadowing and forward light scattering. Enable automatic texture coordinate generation so that rendering a quad along the current slice will be correctly textured by the 3D cloud density texture. At each slice, render a quad, and then copy the resulting frame buffer to the current slice of the 3D illumination texture. The result, shown in the middle of Figure 6, is a 3D texture that represents volumetric illumination.

At run time, we render slices oriented to the viewer. We bind the cloud density texture and the light texture to the first two texture units, and we again use automatic texture coordinate generation. Finally, we set the texture matrix on the second texture to be the transformation matrix from the cloud space (usually world space) into the oriented light volume's space. This transform is determined by the fitting of the oriented bounding box. By using this texture matrix,

we transform texture coordinates for each lookup into the correct position in the light volume, so that the cloud is correctly illuminated.

For efficiency, we typically use a light volume texture that is one half the resolution of the cloud density volume. This allows us to very quickly create the light volume. Our algorithm is an alternative implementation of traditional two-pass volumetric illumination algorithm based on [Kajiya and Von Herzen 1984]. It is similar to the "shadow buffer" algorithm of [Levoy 1988]. The advantage is that since our light volume is oriented along the light direction, it is fast to compute in hardware. The disadvantage is that part of the resolution is wasted unless the light direction matches one of the cloud volume's axes, as can be seen in Figure 7.

8 Results and Conclusion

We have demonstrated a method for fast, physically-based cloud simulation implemented on programmable graphics hardware. Our GPU simulation system provides fast simulation of clouds on larger volumes than has been seen previously. On a volume of resolution 32^3 , we achieve an update rate of approximately 27 iterations per second on an NVIDIA GeForce FX Ultra. Flat 3D textures improve the scalability of the computation, since the number of render passes for simulations of *any* resolution is the same. Thus, a volume of resolution 64^3 updates at about 3.6 iterations per second, which means that the efficiency is increasing with the increase in volume. With traditional 3D textures, we doubt we would see the same scalability in our simulations.

Our simulation amortization technique has proven very valuable for visualizing the results of our simulations. Combined with our efficient illumination algorithm and the use of impostors for rendering, this technique allows the user to move around and through clouds like the ones in Figure 1, Figure 8 and **Figure 9** at high frame rates (the grid resolutions in both figures were $64 \times 32 \times 32$ and $64 \times 32 \times 64$).

The advection portion of our simulation is one of its bottlenecks. This is especially true in the case of staggered grid advection. We have made a comparison of the performance of advection under various conditions, shown in Table 2. We found that in large 3D staggered grid simulations, the advection could cause a large, regular jump in our smooth frame rate when performing the amortized simulation. We solved this problem by splitting advection into three less expensive passes (one for each dimension). This increase in granularity enables a more balanced per-

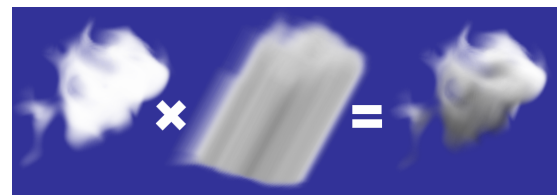


Figure 6: An unlit density volume (left), an oriented light volume (middle), and the resulting illuminated volume (right).



Figure 8: Simulated clouds in our interactive flight application, *SkyWorks*.

frame simulation cost, since the velocity computation can be spread over more than one frame. Because velocity is stored in three separate color channels of a texture, we use the color mask functionality of OpenGL to ensure that each advection pass writes only a single color channel. This way, only one texture update is necessary for all three passes. Splitting advection has another advantage. Fragment program performance on GeForce FX decreases with the number of registers used. Therefore even though the total instruction count for the split version is slightly higher, the shorter fragment programs execute faster since they use fewer registers. Therefore the total cost of split advection is lower, as shown in Table 2.

In the future, we hope to enhance the performance of our simulation. One optimization we have not taken much advantage of is the use of textures as lookup tables. While fragment programs provide computational flexibility, this comes at a cost. Lookup tables are important just as they are in CPU computation. Also, as our simulation grid sizes increase, we think that a more sophisticated linear solver will be needed to achieve good convergence. The multigrid method shows promise for accurate large-grid simulation on the GPU [Bolz, et al. 2003; Goodnight, et al. 2003].

Advection quantity	Average time per pass (ms)	
	Collocated Grid	Staggered Grid
Scalar 2D	0.71	0.87
Velocity 2D	0.67	2.40
Total 2D	1.39	3.28
Scalar 3D	N/C	2.34
Velocity 3D	N/C	8.28
Total 3D	N/C	10.61
Velocity 3D Split (3 passes)	N/C	5.90
Total 3D Velocity Split	N/C	8.24

Table 2: Cost comparison for various implementations of the advection operation. (N/C = “not computed”.)

©The Eurographics Association 2003.

We also plan to improve the visual results of our simulation. One major improvement will come from animation blending. Currently, the low simulation update rate causes visual “popping”. Linear interpolation of the current and next time step will help. A possibly better idea would be to use the current velocity field to perform a partial advection at each time step. This advection would use the collocated grid advection operator for efficiency, since the goal would be visual smoothing, not accuracy.

Acknowledgements

The authors would like to thank Jeff Juliano and others at NVIDIA for their generous help in tracking down driver bugs and other problems. We are also grateful to Mike Bunnell for the original idea behind flat 3D textures. This work was supported in part by NVIDIA Corporation, The Link Foundation, NSF grant number ACI-0205425, and NIH National Institute of Biomedical Imaging and Bioengineering, P41 EB-002025.

References

- [Andrews 2000] Andrews, D.G. *An Introduction to Atmospheric Physics*. Cambridge University Press. 2000.
- [Bolz, et al. 2003] Bolz, J., Farmer, I., Grinspun, E. and Schröder, P. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *Computer Graphics (Proceedings of SIGGRAPH 2003)*, ACM Press. 2003.
- [Chorin and Marsden 1993] Chorin, A.J. and Marsden, J.E. *A Mathematical Introduction to Fluid Mechanics*. Third. Springer. 1993.
- [Dobashi, et al. 2000] Dobashi, Y., Kaneda, K., Yamashita, H., Okita, T. and Nishita, T. A Simple, Efficient Method for Realistic Animation of Clouds. *Computer Graphics (Proceedings of SIGGRAPH 2000)*, ACM Press, 19-28. 2000.
- [Fedkiw, et al. 2001] Fedkiw, R., Stam, J. and Jensen, H.W. Visual Simulation of Smoke. *Computer Graphics (Proceedings of SIGGRAPH 2001)*, ACM Press / ACM SIGGRAPH. 2001.

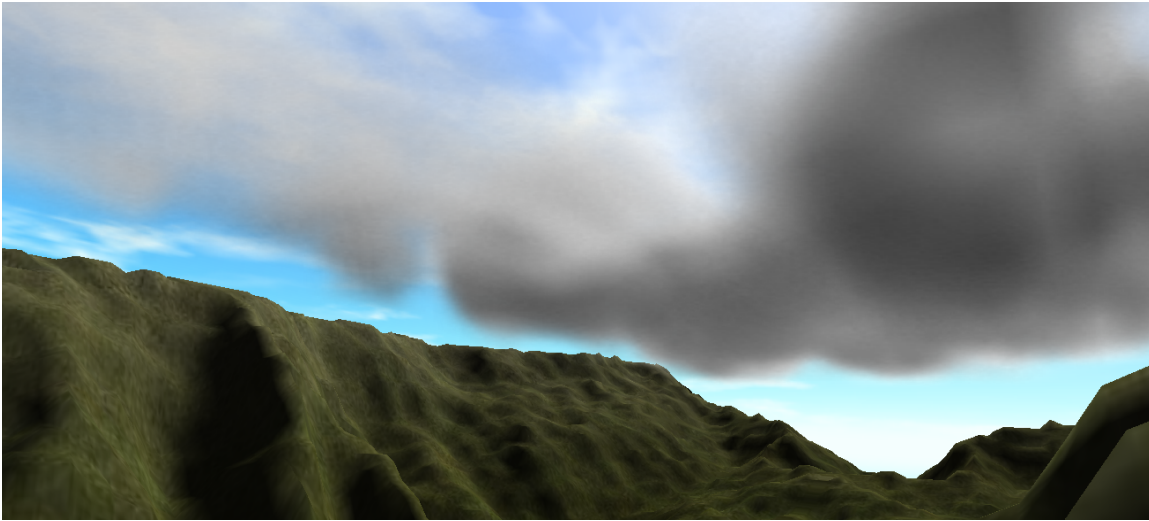


Figure 9: A view of simulated clouds from the ground. (The far clouds are part of the sky texture.)

[Foster and Metaxas 1997] Foster, N. and Metaxas, D. Modeling the Motion of a Hot, Turbulent Gas. *Computer Graphics (Proceedings of SIGGRAPH 1997)*, ACM Press, 181-188. 1997.

[Golub and Van Loan 1996] Golub, G.H. and Van Loan, C.F. *Matrix Computations*. Third Edition. The Johns Hopkins University Press. 1996.

[Goodnight, et al. 2003] Goodnight, N., Woolley, C., Lewin, G., Luebke, D. and Humphreys, G. A Multigrid Solver for Boundary Value Problems using Graphics Hardware. *Proceedings of Graphics Hardware 2003*, 2003

[Griebel, et al. 1998] Griebel, M., Dormseifer, T. and Neunhoeffler, T. *Numerical Simulation in Fluid Dynamics : A Practical Introduction*. Society for Industrial and Applied Mathematics. 1998.

[Harris, et al. 2002] Harris, M.J., Coombe, G., Scheuermann, T. and Lastra, A. Physically-Based Visual Simulation on Graphics Hardware. *Proceedings of SIGGRAPH / Eurographics Workshop on Graphics Hardware 2002*, 2002

[Harris and Lastra 2001] Harris, M.J. and Lastra, A. Real-Time Cloud Rendering. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, Blackwell Publishers, 76-84. 2001.

[Houze 1993] Houze, R. *Cloud Dynamics*. Academic Press. 1993.

[Kajiya and Von Herzen 1984] Kajiya, J.T. and Von Herzen, B.P. Ray tracing volume densities. *Computer Graphics (Proceedings of SIGGRAPH 1984)*, ACM Press, 165-174. 1984.

[Krüger and Westermann 2003] Krüger, J. and Westermann, R. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *Computer Graphics (Proceedings of SIGGRAPH 2003)*, ACM Press. 2003.

[Levoy 1988] Levoy, M. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(3). 29-37.1988.

[Mark, et al. 2003] Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M.J. Cg: A System for Programming Graphics Hardware in a C-like Language. *Computer Graphics (Proceedings of SIGGRAPH 2003)*, ACM Press. 2003.

[Miyazaki, et al. 2001] Miyazaki, R., Yoshida, S., Dobashi, Y. and Nishita, T. A Method for Modeling Clouds Based on Atmospheric Fluid Dynamics. *Proceedings of Pacific Graphics 2001*, IEEE Computer Society Press, 363-372. 2001

[Overby, et al. 2002] Overby, D., Melek, Z. and Keyser, J. Interactive Physically-Based Cloud Simulation. *Proceedings of Pacific Graphics 2002*, 469-470. 2002

[Rogers and Yau 1989] Rogers, R.R. and Yau, M.K. *A Short Course in Cloud Physics*. Third Edition. Butterworth Heinemann. 1989.

[Schauffler 1995] Schauffler, G. Dynamically Generated Impostors. *Proceedings of GI Workshop "Modeling - Virtual Worlds - Distributed Graphics"*, infix Verlag, 129-135. 1995

[Stam 1999] Stam, J. Stable Fluids. *Computer Graphics (Proceedings of SIGGRAPH 1999)*, ACM Press, 121-128. 1999.