

# Let There Be Clouds!

Fast, Realistic Cloud-Rendering in  
MICROSOFT FLIGHT SIMULATOR 2004:  
A CENTURY OF FLIGHT



**Y**ou're standing on rolling hills beneath a brilliant blue sky. You look up and see huge spherical white blobs suspended a few thousand feet in the air. What's wrong with this picture? Perhaps you could use a better cloud-rendering system.

In videogames that simulate outdoor reality, realistic clouds can be one of the most compelling aspects of the scene. Clouds can also set the mood — dark thunderheads for an ominous scene, light puffy clouds for a happy mood. Michelangelo spent years perfecting the heavens on the ceiling of the Sistine Chapel, but we need to render realistic clouds in milliseconds. Fortunately, we have more advanced tools to work with. This article describes the cloud modeling and rendering system that ships with MICROSOFT FLIGHT SIMULATOR 2004: A CENTURY OF FLIGHT.

Clouds in the real world consist of many types, such as altocumulus, stratus, and cumulonimbus, and cloud coverages ranging from a few sparse clouds to a dense, overcast sky. Our cloud system models this range of cloud types and coverages. MICROSOFT FLIGHT SIMULATOR allows users to download real-world weather and see current weather conditions reflected in the game graphics, which means we need to generate compelling visuals to match any scenario that could occur in the real world.

The interactive nature of games necessitates that clouds

must look realistic whether the camera is far away, next to the cloud, or traveling through the cloud. Another requirement is that we need to render at high framerates. MICROSOFT FLIGHT SIMULATOR supports a wide range of machines, from the latest PCs to those dating back several years, and the performance must scale to this spectrum of machines.

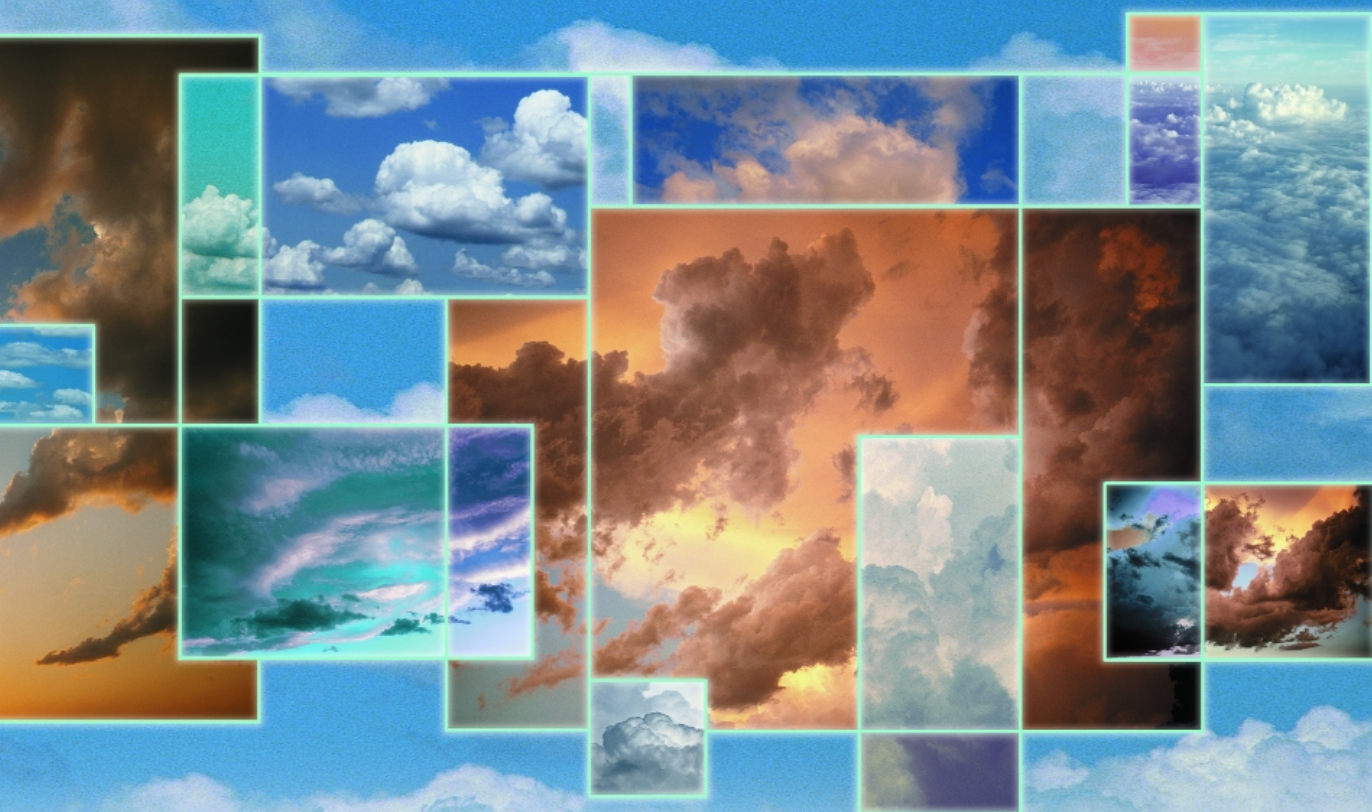
Clouds need to be shaded appropriately to emulate both sunlight and light reflected from the sky, especially for games such as MICROSOFT FLIGHT SIMULATOR, which take place over the course of day, spanning dawn, midday, dusk, and night. We model the dynamic aspect of clouds by introducing a method to form and dissipate them over time.

## Previous Work

**O**ver the past 20 years, graphics researchers have modeled clouds in many ways, including cellular automata, voxels, and metaballs. They also modeled cloud animation via fluid dynamics. There are two reasons that these research techniques have not been widely adopted by games. The first is performance. Many of these systems produced screenshots that were gorgeous but required multiple seconds to render. The second is lack of artistic control. Imagine that you create a cloud by running a set of fluid dynamics equations. You examine the results and decide you would like a wispy top on the cloud. You must then iterate through cycles of adjust-

**NINIANE WANG** | Niniane has five years of game development experience at Microsoft Game Studios. Most recently she was a software engineer lead on MICROSOFT FLIGHT SIMULATOR 2004: A CENTURY OF FLIGHT. She can be reached at [niniane@ofb.net](mailto:niniane@ofb.net).





ing variables such as air humidity and temperature, and recomputing the equations, which can require hours and still may not produce the visual effect you had in mind.

Realistic results in cloud shading have been achieved by simulating the scattering of light by particles as it passes through the cloud, known as anisotropic scattering. This produces accurate self-shadowing and interesting effects such as the halo when the cloud lies between the camera and the sun. We created a simple shading model for our system, forgoing these effects in exchange for fewer computations and higher artistic control.

Many flight simulation games have featured clouds, recent examples being FLIGHT SIMULATOR 2002, IL-2 STURMOVIK, and COMBAT FLIGHT SIMULATOR III. A common approach is to paint clouds onto the skybox texture, which has minimal performance overhead, but such clouds look two-dimensional and never get closer as the camera moves toward them. A better solution is to draw each cloud as a single facing sprite. This solution looks realistic from a stationary camera but produces anomalies as the camera rotates around it. A few recent games use clusters of textured particles, similar to our system. Some use unique textures for every cloud, which has a high video memory cost as the number of clouds in the scene increases. Other systems use small blurry textures, which results in clouds that look volumetric but lack definition. All of these systems also lack the ability to form and dissipate clouds.

Our system was inspired after hearing a GDC talk by Mark Harris, who developed Skyworks, a real-time system that created volumetric clouds from sprites. Harris dynamically gener-

ated an impostor for every cloud and achieved speeds of 1 to 500 frames per second. He also modeled the fluid motion behind cloud animation. The limitation of his system is that it cannot render large clouds, such as cumulonimbus, or dense scenes of overcast clouds, due to the prohibitively high video memory cost of generating large impostors. Our system is able to address this limitation. In addition, we tackle the problem of scaling to multiple cloud types.

## Cloud Modeling

**G**iven that we want immediate visual feedback and full control over the final result, how can we design the artistic pipeline for modeling clouds? We model each cloud as five to 400 alpha-blended textured sprites. The sprites face the camera during rendering and together comprise a three-dimensional volume. We render them back-to-front based on distance to the camera.

We wrote a plug-in for 3DS Max that creates cloud sprites based on a 3D model composed of boxes. The artist denotes a cloud shape by creating and placing a series of boxes, using default 3DS Max functionality. The artist can create any number of boxes of any size and can choose to overlap the boxes.

The plug-in UI contains an edit field to specify the number of sprites to generate. To create denser clouds, the artist would set a number which is proportionately higher than the size of boxes in the model. Wispy clouds would be created by setting a lower number. There are generally 20 to 200 boxes for each 16-square-kilometer section of clouds, and the number of

sprites per box can vary between 1 to 100, depending on the density. The UI also allows the artist to specify a range for the width and height of each sprite, and choose between categories (such as stratus and solid cumulus) that determine the textures that will be automatically placed by the tool onto the sprites. Figure 1 shows a screenshot of the tool UI.

The artist presses a button in the plug-in UI to generate the cloud sprites. The plug-in creates a list of randomly placed sprite centers, then traverses the list and eliminates any sprite whose 3D distance to another sprite is less than a threshold value (the “cull distance”). This process reduces overdraw in the final rendering and also eliminates redundant sprites created from overlapping boxes. We have found that a cull radius of 1/3 of the sprite height works well for typical clouds, and 1/5 to 1/6 of the sprite height yields dense clouds. Figure 2 shows screenshots of a cloud model made of boxes and its corresponding sprites.

The plug-in creates an initial model of sprites, and the artist can now edit them within 3DS Max. Having achieved the desired visual look, the artist uses a custom-written exporter to create a binary file containing the sprite center locations, rotations, width, and height, along with texture and shading information. These files are loaded during game execution and rendered.

## Textures

To create a dozen distinct cloud types, we mix and match 16 32-bit textures for both color and alpha (see Figure 3). The flat-bottomed texture in the upper right-hand corner is used to create flat bottoms in cumulus clouds. The three foggy textures in the top row are used heavily in stratus clouds and have a subtle bluish-gray tinge. The six puffy textures in the bottom two rows give interesting nuances to cumulus clouds, and the remaining six are wispy sprites that are used across all cloud types.

By creating interesting features inside the textures that resemble eddies and wisps, we are able to create more realistic looking clouds with fewer sprites. We place all 16 textures on

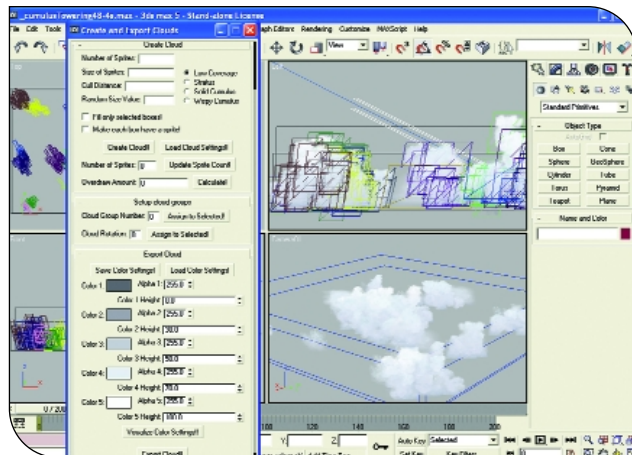


FIGURE 1. A custom tool within 3DS Max allows the artist to set properties and generate cloud sprites.

a single 512×512 texture sheet, which spared the cost of switching textures between drawing calls to the video card. We automatically generate mip-map levels for this texture from 512×512 down to 32×32. To create more variations from these 16 textures, the artist specifies a minimum and maximum range of rotation for each sprite. When the binary file is loaded into the game, the sprite is given a random rotation within the range.

## In-Cloud Experience

We would like a seamless in-cloud experience that looks consistent with the cloud’s appearance as viewed from the outside, which does not often come with the commonly used technique of playing a canned in-cloud animation. In our system, as the camera passes through a sprite, it immediately disappears from view. We encountered a problem because the sprites rotated to face the camera, and during the in-cloud experience, the camera was so close to the sprite center that

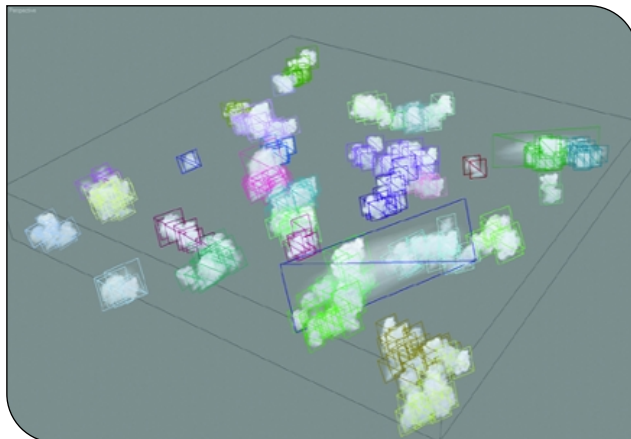
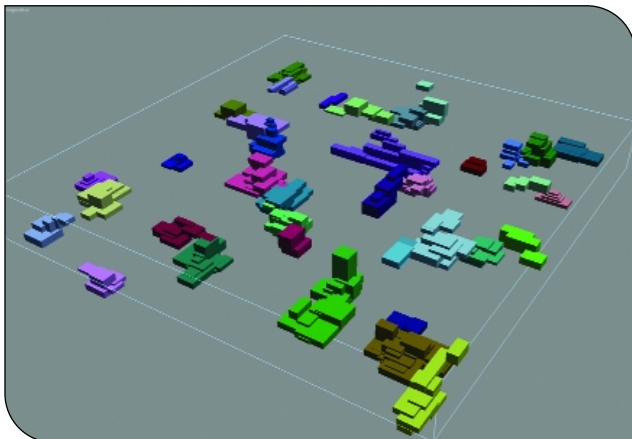


FIGURE 2. Artists use boxes to build the shapes of clouds (left), and then let a custom tool in 3DS Max populate the boxes with sprites (right).





FIGURE 3. The 16 textures used to create a dozen cloud types (left) and two cloud varieties: stratus cumulus (middle) and cumulus congest (right).

small movements in the camera position caused large rotations of the cloud sprite. This resulted in a “parting of the Red Sea” effect as sprites moved out of the way of the oncoming camera.

We locked the facing angle of the sprite when the camera came within half of the sprite radius. This removed the Red Sea effect but caused sprites to be seen edge-on if they locked and the camera then pivoted around them. Our solution was to detect the angle between the sprite’s locked orientation and the vector to the camera, and to adjust the transparency of the sprite. The negative side effect is that the section of the cloud near the camera appears less opaque.

## Performance

**M**ICROSOFT FLIGHT SIMULATOR 2004 maintains framerates of 15 to 60 frames per second on consumer PCs, even in overcast scenes. We achieve this across a range of machines with CPU speeds from 700MHz through 3.0GHz. Predictably, the machines with older CPUs and video cards pose a particular challenge.

The heavy amount of overdraw in the clouds presents an opportunity to improve performance. We use the impostor technique invented by Gernaut Schaufler (see For More Information) of dynamically rendering multiple clouds into a texture that we then display as a billboard. We create an octagonal ring of impostor textures around the camera, each with a 45-degree field of view. We can render hundreds of clouds into a single impostor. Our system compares clouds in 16-square-kilometer blocks against the ring radius and renders only the sections beyond the radius into impostors. Cloud blocks within the radius are rendered as individual sprites (see Figure 4).

We allow the user to set the ring radius. A smaller ring gives better performance but more visual anomalies, which I’ll discuss later. A larger ring means fewer anomalies but less performance gain from the impostors, since fewer clouds are rendered into them.

We render the eight impostors in fixed world positions facing the center of the ring, and re-create them when the camera or sun position has changed past threshold values. We recalculate all eight rather than a lazy recomputation, in case the user suddenly changes the camera orientation. Empirical results show that the user can move through 15 percent of the impostor ring radius horizontally or 2 percent of the ring radius vertically before recalculation becomes necessary.

To prevent variability in framerate when rendering to impostors, we spread out the impostor calculation over multiple frames. For video cards that support it, we do a hardware render-to-texture into a 32-bit texture with alpha over eight frames, one for each impostor. For the other video cards, we use a software rasterizer to render into the texture over dozens of frames, one 16-square-kilometer cloud block per frame. When we update to a new set of impostors, we cross-fade between the two sets.

We translate the impostor texture vertically up or down based on the angle between the clouds and the camera. When the clouds are displaced more than 10,000 feet vertically from the camera, we stop rendering them into impostors because the view angle is too sharp. The overdraw is much less when

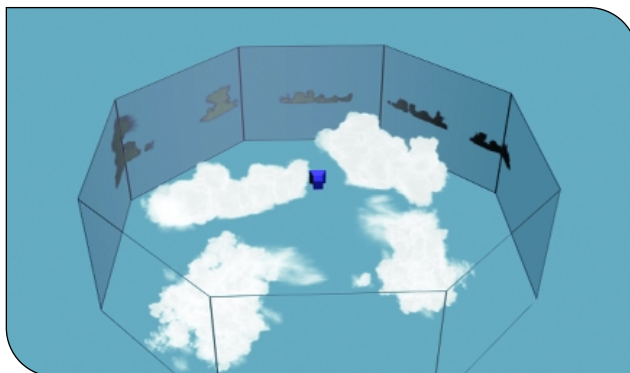


FIGURE 4. An octagonal ring of impostors around the camera eyepoint helps improve performance.

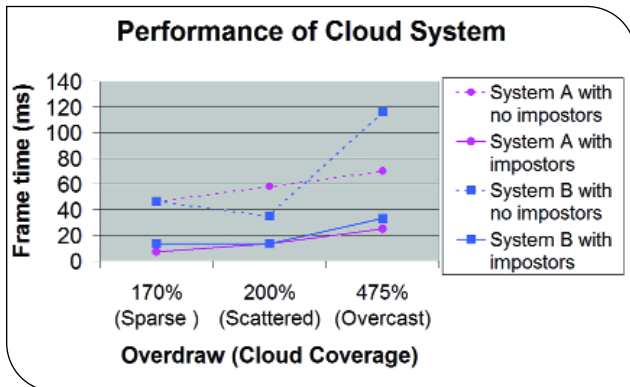


FIGURE 5. Performance comparison, with and without imposters, between System A (1.7GHz Intel Pentium, 768MB RAM, GeForce2 GTS) and System B (733MHz Intel Pentium III, 128MB RAM, Riva TNT2).

the clouds are far away, so performance in this situation only suffers slightly by not rendering into impostors.

Since video memory is frequently a tight resource on consumer PCs, we designed the impostor system to have low video memory usage. Our ring of eight impostors, each a  $256 \times 256$  texture with 32-bit color, adds up to a video memory cost of  $8 \times 256 \times 256 \times 4 = 2$  megabytes. When crossfading, both impostor rings are rendered, which adds another 2 megabytes during the transition.

Figure 5 shows our framerate for three scenes of varying coverage, on two machines. We chose older machines for this experiment to show that our results scale to machines with slower CPUs and lower video memory.

System A is a 1.7GHz Intel Pentium with 768MB of RAM and a GeForce2 GTS video card. System B is a 733MHz Intel Pentium III with 128MB of RAM and a Riva TNT2 video card.

On older systems, such as 450MHz machines with 8MB video cards, the fill rate is so expensive that even with the impostor ring radius at eight kilometers, rendering the cell block within the ring radius as individual sprites produces framerates that fall below 20 frames per second for denser cloud coverage. We created a simple LOD scheme that uses a single sprite per cloud. Since our rendering method scales to any number of sprites in the model, a single-billboard cloud is merely a degenerate case of the model, and we needed no special-case code to render or shade these models.

## Shading: Every Cloud Has a Silver Lining

**W**e chose not to simulate scattering of light from cloud particles, instead using simple calculations based on artist settings that yield a reasonable approximation. This means we do not simulate clouds casting shadows on themselves, other clouds, or other objects in the scene. The two factors that go

into our cloud-shading system are skylight and sunlight.

As rays of light pass from the sky through the cloud, they are scattered and filtered by the particles within the cloud. As a result, clouds typically have darker bottoms. To simulate this, our artists use a color picker in 3DS Max to specify five “color levels” for each cloud. The color level consists of a height on the cloud with an associated RGBA color. These levels are exported in the cloud description file.

Separately, for a set of times throughout the day, the artist will specify a percentage value to be multiplied into the ambient color levels at each particular time of day. This allows the ambient contribution to decrease approaching night.

The sun casts directional light on a cloud, which generates dramatic scenes, especially around dawn and dusk. We simulate the effect so that areas of the cloud facing the sun receive more directional light while areas facing away from the cloud receive less.

Our artists specify shading groups, sections of one of 30 sprites that are shaded as a unit, when they build the clouds in 3DS Max from boxes. On each box, they set a custom user property with a shading group number, and sprites generated for that box will belong to that shading group. These shading groups simulate clumps on the cloud. We calculate the directional component of shading for a given vertex in the cloud by first computing the vector to that point from the shading group center. We also find the vector from the group center to the sun and compute the dot product of the two vectors after normalization.

The artist specifies a maximum directional color and minimum and maximum directional percentages for a range of times throughout the day. We multiply the resulting percentage from the mapping function with the maximum directional color specified by the artist, to get the directional color of the vertex.

To get the final vertex color, we add the ambient and directional colors to the color from the sprite texture. At this point, we also multiply by the alpha value representing formation or dissipation of the cloud.

## Formation and Dissipation

**T**he clouds look more realistic when they can form and dissipate. We control the evolution of a cloud by adjusting the transparency level of sprites.

We multiply a transparency factor into the alpha value for each sprite vertex based on its position within the cloud. When a cloud is beginning to form, we render only the sprites whose center is within half of the cloud radius from the cloud center, and we render them with a high transparency level that we decrease over time. After these sprites have reached a threshold opacity, we begin to render sprites whose center is more than half of the cloud radius from the cloud center. Cloud dissipation is simulated by reversing the process (see Listing 1).

**LISTING 1.** Calculating the alpha value of a vertex when forming and dissipating the cloud.

vertex is the (x, y, z) with respect to cloud center.  
cloud\_radius is the radius of the cloud bounding box.  
alpha\_at\_edges controls how much to fade out the edges;  
this increases from 0 to 255 over time.  
alpha\_of\_cloud controls the transparency of the entire cloud;  
this starts out at 255.  
time\_delta is the time that passed since the last frame.

```
float alpha;
if (fade_out_to_edges)
{
    float radial_magnitude = cloud_radius -
        vector_magnitude (vertex);

    radial_magnitude = max (0, radial_magnitude);

    alpha = (alpha_of_cloud - radial_magnitude * alpha_at_edges /
        cloud_radius) / 255.0f;
}

alpha = min(1.0f, max(0.0f, alpha));

if (alpha_at_edges < 255)
    alpha_at_edges += 255 * time_delta / time_to_fade_cloud_edges;
else
    alpha_of_cloud -= 255 * time_delta / time_to_fade_cloud_core;
```

## Limitations and Extensions

Our system is well suited for creating voluminous clouds but less suited for flat clouds. Of the four basic cloud types — cumulus, stratus, cumulonimbus, and cirrus — our system easily handles the first three. However, cirrus clouds are so flat as to be almost two-dimensional, and it would require a large number of small sprites to model them using our approach, which would cause a performance hit. Instead, we represent cirrus clouds with flat textured rectangles.

Because sprites within each cloud are sorted back-to-front to the camera, moving the camera can occasionally result in popping as sprites switch positions in the draw order. This effect is more noticeable at dawn and dusk, when directional shading plays a greater role, but has not been jarring enough in our experience to necessitate a solution such as caching the previous sort order and crossfading.


As mentioned previously, our shading model does not include cloud shadows, self-shadowing, or the halo effect when the cloud is between the sun and the camera. One potential solution is to precalculate the lit and shadowed regions of the cloud for a set of sun angles. We can load this

information at run time and interpolate based on sun angle.

The ring of impostors can create visual anomalies. The clouds in the impostor do not move relative to each other, which means the parallax looks wrong. Also, distant objects must be drawn either in front of or behind all the clouds in the impostor, instead of in front of some and behind others. We could mitigate this by adding additional rings of impostors, but that increases video memory usage.

In the future, we would like to implement some of our techniques into vertex shaders, as more of our user base upgrades to video cards that support hardware vertex shaders. Our system can be extended to other gaseous phenomena, such as fog, smoke, and fire. Fog is a natural candidate, since it is essentially a stratus layer placed close to the ground. The problem is getting rid of the hard lines where sprites intersect the ground polygons. We can either split the sprites along the ground or multiply by a one-dimensional alpha texture based on the altitude.

Research into fluid simulation has produced realistic animations of clouds as they change shape. This creates more extensive morphing than our system of formation and dissipation, but it frequently does not yield enough control to the artists over the final result. It can be difficult to tweak the humidity and other parameters just right to have a cloud form over three minutes, or to ensure the cloud that forms looks a particular way.

A solution more appropriate for games and movies may be one that combines our artistic modeling with fluid simulation by using simple rules for cloud morphing in combination with our system of textured particles. For example, we could use weather variables such as humidity, airflow, and temperature to rotate, translate, and adjust transparency on individual sprites within the cloud to change the overall shape of the cloud. It would give the impression that wisps are being blown by the wind, or that clouds are condensing or breaking into several pieces. 

## FOR MORE INFORMATION

Download a video describing the cloud-rendering system at [www.gdmag.com](http://www.gdmag.com).

Mark Harris and Anselmo Lastra. "Real-Time Cloud Rendering." Computer Graphics Forum, vol. 20, issue 3. Blackwell, 2001.

Gernaut Schaufler. "Dynamically Generated Imposters." Modeling Virtual Worlds — Distributed Graphics, ed. D. W. Fellner, MVD Workshop, 1995.

## ACKNOWLEDGEMENTS

Thanks to Adrian Woods and John Smith, two very talented artists from Microsoft, without whom this work would not have been possible.