Realistic and Fast Cloud Rendering

Niniane Wang
Microsoft Corporation (now at Google Inc.)
niniane@ofb.net

November 11, 2003

Abstract

Clouds are an important aspect of rendering outdoor scenes. This paper describes a cloud system that extends texture splatting on particles to model a dozen cloud types (e.g., stratus, cumulus congestus, cumulonimbus), an improvement over earlier systems that modeled only one type of cumulus. We also achieve fast real-time rendering, even for scenes of dense overcast coverage, which was a limitation for previous systems.

We present a new shading model that uses artist-driven controls rather than a programmatic approach to approximate lighting. This is suitable when fine-grained control over the look-and-feel is necessary, and artistic resources are available. We also introduce a way to simulate cloud formation and dissipation using texture splatted particles.

1 Introduction

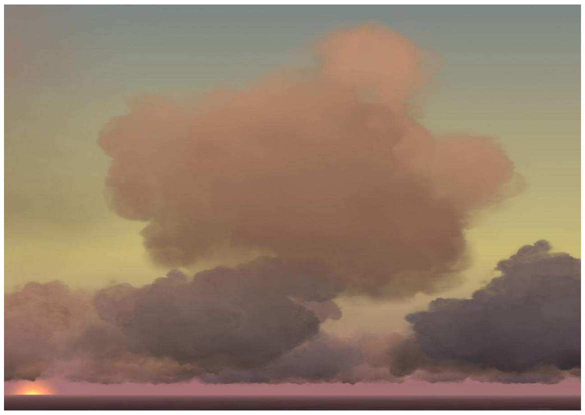

Figure 1: Scene of realistic clouds at sunset.

Clouds play an important role in simulating outdoor environments. Realistic-looking clouds can be one of the most compelling graphical components of outdoor scenes, especially for real-world applications such as flight simulators and movie productions.

In interactive real-world applications, cloud systems must look realistic from any camera position and must scale to handle the wide spectrum of cloud types (e.g., stratus, cumulus congestus) and coverages (e.g., sparse, broken, overcast) that exist in real life. For example, our cloud system ships with Microsoft Flight Simulator 2004: A Century of Flight, which allows users to download real-world weather and see current conditions reflected in the game graphics. One design constraint on our system was that it must yield high-quality results for all scenarios that occur in the real world, from wispy stratus clouds to overcast thunderstorm skies.

To be useful for practical interactive applications, our system must maintain aggressive performance. Using Flight Simulator as an example, the game operates at high framerates while performing computations for numerous systems besides cloud rendering, such as physics, AI, and terrain rendering. Therefore, our cloud modeling must maintain low render times of milliseconds. Additionally, it must achieve fast performance even on low-end PCs, which represent the hardware constraints for a large percentage of the user base.

The appearance of clouds is affected by the light cast by the sun and filtered from the sky, which must be reflected in the cloud shading. Our system targets applications that require fine-grained control over the look-and-feel and have artistic resources available, such as movie productions and computer games.

In the real world, clouds do not remain static. They move across the sky, form in areas of moisture and unstable air, and dissipate when these conditions abate. To simulate these changes, our cloud rendering method must include a mechanism for forming and dissipating clouds in a realistic manner over time.

In this paper, we present a cloud rendering system that addresses all of the above. We use texture splatting on particles to create ”puffs” in the cloud, which allows us to model a dozen cloud types such as nimbostratus and altocumulus, an improvement over previous systems that modeled one type of cumulus cloud. We use an octagonal ring of impostors to achieve fast performance, even for scenes of dense overcast cloud cover, which have traditionally proven to be a challenge. To approximate sky and sun light, we use a simple efficient shading model. Finally, our approach allows us to dynamically evolve the cloud into existence or fade parts of it away.

Section 2 reviews the background and previous work. Section 3 describes our technique, and section 4 explains how our system dynamically forms and dissipates clouds. Section 5 covers our shading methods, and section 6 presents performance metrics and results. Section 7 describes our experiences, algorithmic limitations, and directions for future work.

A short video demonstrating the cloud system is available online at the web address listed at the end of this paper.

2 Previous Work

Many techniques have been used to model, animate, and render clouds.

[Per85] modeled cloud volumes by filling them with procedural solid noise techniques, and [ES00] extended this to model clouds in real-time using textured ellipsoids. [DKY^{+}00] used metaballs — also called blobs — to realistically model cloud shapes based on atmospheric conditions. With programmatic approaches, it is often difficult to achieve an exact desired result, as the creation process involves tweaking equation parameters rather than directly adjusting the visual model.

To create clouds that change dynamically, [DNYO99] used voxels with forms of cellular automata. [Ebe97] combined volume rendering and a-buffer rendering techniques to animate gaseous phenomena based on turbulent flow. These systems produced beautiful images but were not real-time.

Shading has also been explored in various systems. [Bli82] pioneered work on cloud shading by introducing a single scattering model of light reflected by uniform small particles. [NDN96, Har03] simulated multiple anisotropic scattering of light from particles in the cloud as well as sky light. These systems created accurate shading. When designing our system, we made the tradeoff of accepting small shading inaccuracies in exchange for fewer computations and higher artistic control.

Many flight simulation games feature clouds. Recent examples are Flight Simulator 2002, IL-2 Sturmovik, and Combat Flight Simulator III. In games, a common approach is to paint clouds onto the skybox texture. This places almost no overhead on performance but the clouds do not look three-dimensional and never get closer as the camera moves. Another solution is to use a single sprite per cloud, which looks realistic from a stationary camera but produces anomalies as the camera rotates around it. A couple of recent approaches use clusters of texture splatted 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 creates clouds that look volumetric but lack definition (small details that resemble wisps and eddies). These systems also lack the ability to form and dissipate clouds.

Our work is most closely related to [HL01, Har03], a real-time system that built volumetric clouds from texture splatted particles, with a uniform Gaussian blob texture. Harris internalciteHarris:2003 dynamically generated an impostor for each cloud, and achieved framerates of 1 - 500 frames per second. The system also modeled the fluid motion and thermodynamic forces behind cloud animation. Our system differs in that we can render large clouds such as cumulonimbus, as well as scenes of overcast clouds, which had prohibitively high video memory costs in Harris internalcite-Harris:2003's system, since large clouds require large impostors. We also tackle the additional problem of scaling to multiple cloud types.

3 Cloud Modeling

Our cloud rendering technique renders each cloud as 5 to 400 alpha-blended textured sprites. The sprites face the camera during rendering and together comprise a 3-dimensional volume. We render them back-to-front based on distance to the camera.

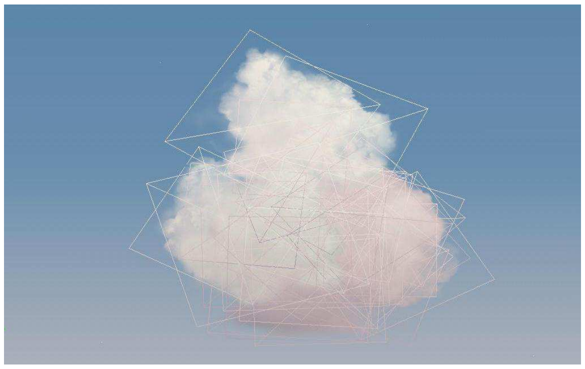

Figure 2: Frontal view of a single cloud with sprites outlined.

3.1 Cloud Creation Process

We designed our authoring process to give artists fine-grained control over the final look of the cloud model, with immediate visual feedback during editing. We chose not to use a more automated approach to programmatically generate the cloud models, as our experiences with them indicated that getting a particular desired effect was difficult since it involved tweaking parameters in equations whose effect on the visual model was less directly controllable. Our choice requires the availability of artists, which is true for our applications such as games and movie productions.

We wrote a plug-in to 3D Studio Max that allows artists to denote a cloud shape by creating and placing a series of boxes. When they press a button in our plug-in UI, the plug-in populates the boxes with randomly placed sprites. See Figures 3 and 4. The artist can control the number of sprites to create denser or wispier clouds. He also specifies ranges for the width and height of the sprites. In the clouds we created, the majority of sprites are square, though we used some wide, short sprites to create hazy areas since their texture gets stretched horizontally.

There are generally 20 – 200 boxes for each 16 square kilometer section of clouds, and 1 – 100 sprites per box depending on the cloud density. The UI allows the artist to specify the aforementioned sprite dimensions and density, along with the textures to use and other cloud properties.

After creating a list of randomly placed sprite centers, the tool traverses the list and eliminates any sprite whose 3-D distance to another sprite is less than a threshold value. This reduces overdraw in the final rendering, and also eliminates redundant sprites created from overlapping boxes. We have found that a cull radius of $\frac{1}{3}$ of the sprite height works well for typical clouds, and $\frac{1}{5}$ ... $\frac{1}{6}$ of the sprite height yields dense clouds.

After the artist edits the clouds, he 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.

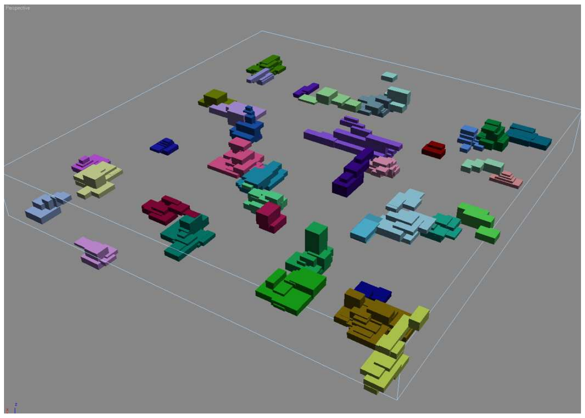

Figure 3: Boxes in 3D Studio Max representing a square region of clouds.

3.2 Textures

We mix-and-match 16 textures to create a dozen distinct cloud types. Three of our cloud types can be seen in Figure 5.

The 16 textures can be seen in Figure 6. They are 32-bit, and used for both color and alpha. The flat-bottomed texture in the upper righthand 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-grey 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 keep the video memory cost low by placing all 16 textures on a single 512x512 texture sheet, which spares the cost of switching textures between drawing calls to the video card. Economizing video memory is important for performance on low-end PCs running video cards with 8 or 16 megabytes of memory, and we achieve a significantly lower cost as compared to other cloud systems which use a different texture for every cloud.

To create more variation 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. We have found that it looks best to give the cumulus cloud bottoms a narrow range (-5 to 5 degrees) and all other sprites the full range of rotation (0 to 360 degrees). We do not apply a rotation to non-square sprites.

3.3 In-cloud Experience

Our approach generates a natural in-cloud experience where the camera appears to be passing through puffs of the cloud. As the camera passes through a sprite, the sprite immediately disappears from view. This ceates a consistent in-cloud experience, in

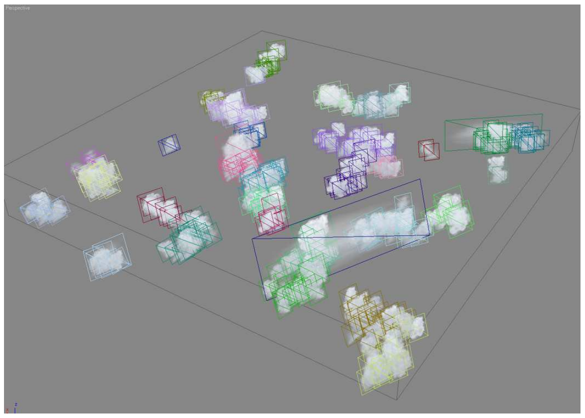

Figure 4: After running our script, the boxes from Figure 3 are filled in with textured sprites.

contrast to the sometimes jarring transition that occurs with the technique of playing a canned animation when the user flies into the cloud.

When we first implemented our solution, we had the sprites always face the camera so that they would not be seen edge-on. However, during the in-cloud experience, the camera was so close to the sprite center that 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.

Our solution is to lock the facing angle of the sprite when the camera comes within half of the sprite radius. This removes the Red Sea effect, but creates a new problem when the camera moves close to the sprite, causing it to lock, and then pivots around until the sprite can be seen edge-on. We experimented with detecting this situation and shifting the sprite to a new locked orientation, but it causes a noticeable jump right as the user passes through the sprite. Our solution is to detect the angle between the sprite's locked orientation and the vector to the camera, and adjust the transparency of the sprite, which also produces the side effect of making that section of the cloud appear less opaque.

4 Cloud Formation and Dissipation

Enabling the formation and dissipation of clouds adds another level of realism to the visual experience. We control the evolution of a cloud by adjusting the transparency level of sprites. To do this, we calculate a transparency factor and multiply it into the alpha from the shading equations for each vertex.

We decide the transparency factor based on the sprite's 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 they have reached a threshold opacity,

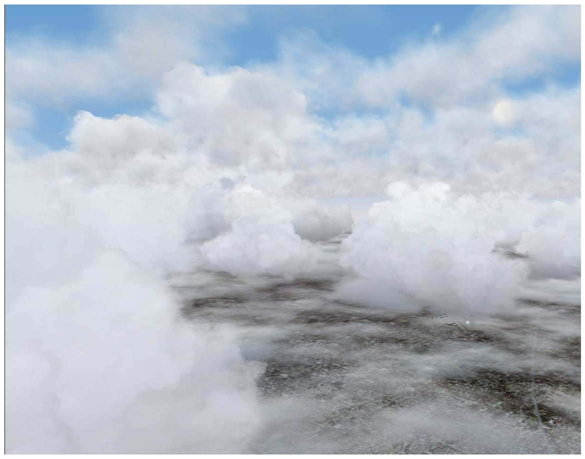

Figure 5: Scene showing 3 of our 10 cloud types: a lower layer of stratus, middle layer of cumulus congestus, and high layer of altocumulus.

we begin to render sprites whose center is over half of the cloud radius from the cloud center.

Cloud dissipation is simulated by reversing the process. We first increase the transparency of sprites whose centers are greater than half the cloud radius from the cloud center. When they have faded away completely, we increase the transparency of the sprites within half the cloud radius, with a greater transparency for sprites nearer the cloud edges. A sequence can be seen in Figures 7, 8, 9.

5 Cloud Shading

Previous research on cloud shading have calculated single and multiple scattering of light reflecting off particles within the cloud. We chose to use simpler calculations based on artist settings that yield a reasonable approximation, foregoing some lighting effects in exchange for more artistic control and fewer runtime computations.

The two factors that go into our cloud shading system are sky light and sunlight.

5.1 Approximation of Scattered Sky Light

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 3D Studio max to specify 5 ”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. (See Figure 10).

The artists can also use these ambient color levels to model the distinct cloud types. They set the color levels on stratus clouds to light gray, and nimbostratus a darker gray for a more ominous appearance. Cumulonimbus clouds receive a dark look from top

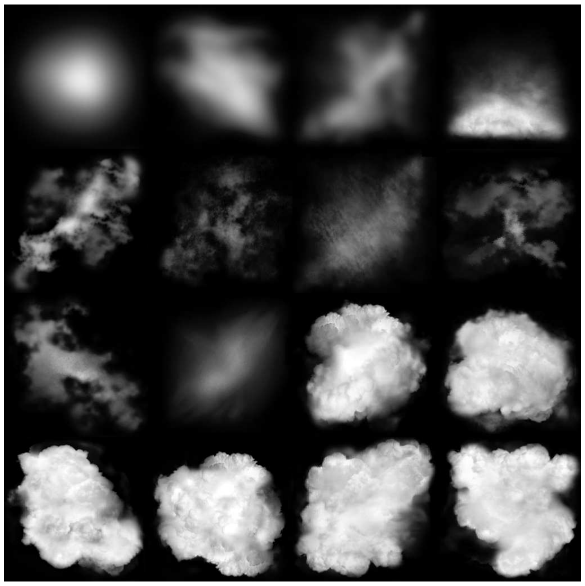
Figure 6: Sheet of 16 cloud sprite textures.

to bottom, while small cumulus humilis puffs are nearly uniformly white. The artist can also adjust the alpha values to make some clouds types more transparent, such as stratus.

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. We allow more samples around dawn and dusk.

When rendering a cloud during game execution, we calculate the 4 corners of each sprite after pivoting it to face the camera and determining locking due to close proximity to the camera. For each corner vertex, we take the vertical component of the position and use it to interpolate between the vertical color levels to get an RGBA color. Independently, we use the time of day to interpolate between the array of percentages over the day. We multiply the RGBA color by the percentage to get the ambient color for this sprite vertex at this time of day.

Given a vertex (V_x, V_y, V_z) at a time of day T between time T_0 with color C_{T0} and time T_1 with color C_{T1}, let $A_T = \frac{T_1 - T}{T_1 - T_0}$. If V_y is between vertical color level V_0 with color T_{V0} and level V_1 with color C_{V1}, let $A_V = \frac{V_1 - V_y}{V_1 - V_0}$. Then the ambient color calculation is given by:

$$C_{amb} = (A_V * C_{V0} + (1\text{–}A_V) * C_{V1}) * (A_T * C_{T0} + (1\text{–}A_T) * C_{T1}) \tag{1}$$

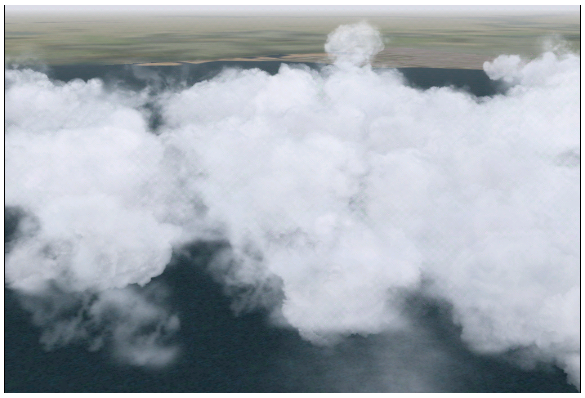

Figure 7: Cloud prior to dissipation.

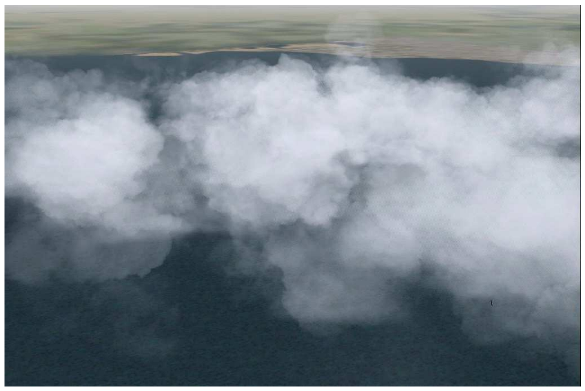

Figure 8: Cloud edges are fading out.

5.2 Approximation of Sunlight

The sun casts directional light on a cloud, which generates dramatic scenes, especially around dawn and dusk. We simulate the effect that areas of the cloud facing the sun receive more directional light while areas facing away from the cloud receive less. We do not simulate clouds casting shadows on themselves, other clouds, or other objects in the scene.

Our artists specify shading groups, sections of 1 – 30 sprites that are shaded as a unit, when they build the clouds in 3D Studio 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. See Figure 11.

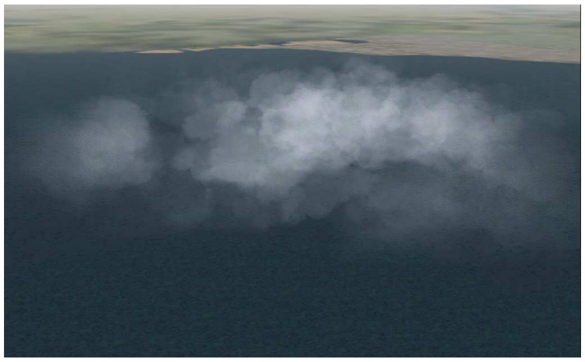

Figure 9: Cloud is mostly faded out.

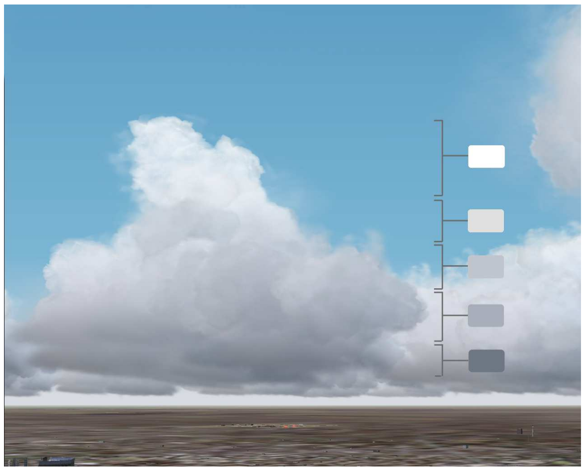

Figure 10: Ambient shading through vertical color levels.

We want to map the dot product from the range of $[-1,1]$ to $[C_{min}, C_{max}]$, biasing the result so that $[-1,0]$ maps to $[C_{min}, C_{median}]$ and $[0,1]$ maps to $[C_{median}, C_{max}]$. The reason for biasing the result is to avoid a sharp transition from light to dark down the middle of the cloud. The result from the mapping function determines the percentage of the maximum directional color for that vertex. The colors $C_{min}, C_{median}, C_{max}$ are decided by artists.

Artists specify directional colors for various times throughout the day, and we interpolate between them to get the maximum directional color at a given time of day. We multiply this color by the percentage computed above.

Given a vertex at time of day T between time T_0 with color C_{T0} and time T_1 with color C_{T1}, let $A_T = \frac{T_1 - T}{T_1 - T_0}$ as above. If V_{vc} denotes the normalized vector from the vertex to the cloud center, and V_{cs} is the normalized vector from the cloud center to the sun, then the directional color calculation is given by:

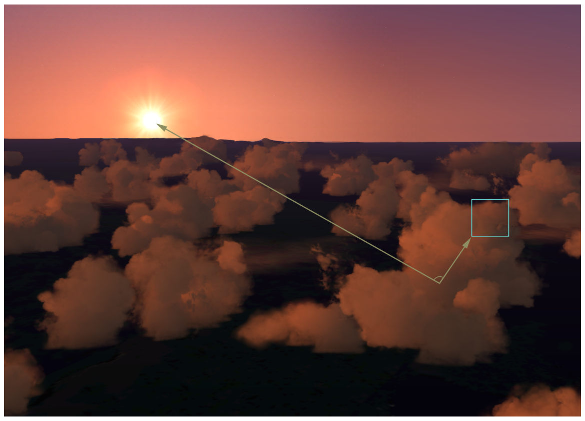

Figure 11: Directional shading.

$$C_{dir} = \text{mapping func}\,(V_{vc} \bullet V_{cs}) * (A_T * C_{T0} + (1\text{–}A_T) * C_{T1}) \tag{2}$$

5.3 Combining the Shading Components

To get the final vertex color, we add the ambient and directional color to the color from the sprite texture. At this point, we also multiply by the alpha value representing formation or dissipation of the cloud ($Alpha_{morph}$).

$$C_{vertex} = (C_{amb} + C_{dir}) * C_{texture} * Alpha_{morph} \tag{3}$$

6 Performance

To be useful in real-time systems, our cloud rendering must achieve fast performance. Our system is used within Microsoft Flight Simulator 2004, which maintains framerates of 15 – 60 frames per second on a consumer PC, including physics and AI computations and all other rendering. Our requirements within Flight Simulator necessitated that we render a 100km radius of thick cumulonimbus clouds within 5 – 40 milliseconds. An additional requirement was that we maintain high framerates on low-end PCs with older video cards, since a large percentage of the user base run the product on these types of machines.

The two main bottlenecks are vertex calculations on the CPU and fill rate due to overdraw on the GPU. To address the former, we cache vertex calculations, and for the latter, we employ impostor textures.

6.1 Caching Vertex Calculations

Because our sprites turn to face the camera, a change in the camera position causes recalculation of the sprite vertices, which in turn requires recomputing the ambient and directional shading values. Recalculating a scene of 500 clouds can take up to 3 milliseconds.

We mitigate this cost by caching computations and recalculating only when the time elapsed or the camera position delta exceeds threshold values. These values are based on the distance of the cloud to the camera, because rotational changes in nearby sprites are more visually noticeable than changes in distant sprites. We tweaked the thresholds to ensure they were not large enough to cause visual popping. We also recompute more frequently when the cloud is forming or dissipating.

6.2 Reducing Overdraw with Impostors

The heavy amount of overdraw in the clouds presents an opportunity to improve performance. We use the impostor technique [Sch95] of dynamically rendering multiple clouds into a texture that we then display as a billboard. (See Figure 12.) This reduces overdraw as well as the number of triangles being rendered.

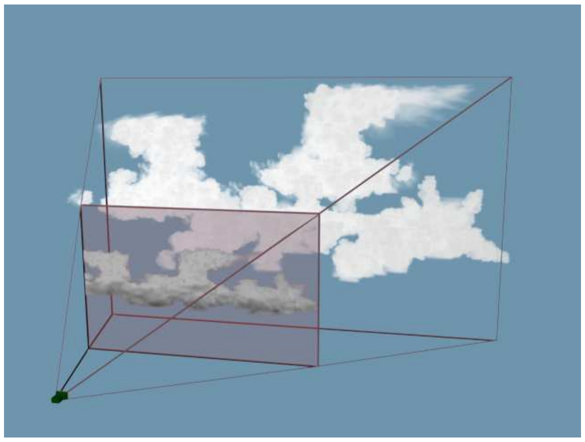

Figure 12: Rendering clouds onto an impostor.

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 blocks beyond the radius into impostors. Cloud blocks within the radius are rendered as individual sprites. (See Figure 13.) This reduces video memory requirements since creating impostors for nearby clouds would require larger textures to maintain the level of detail.

Our system allows the user to set the radius for the ring, which poses a tradeoff. A smaller ring radius signifies that more clouds are rendered into the impostors, which boosts framerate, but there are more noticeable visual anomalies. (See section 7.) A larger ring radius means that there is less performance gain from the impostors since fewer clouds are rendered into them, but there are fewer anomalies, and the impostors can be updated less frequently.

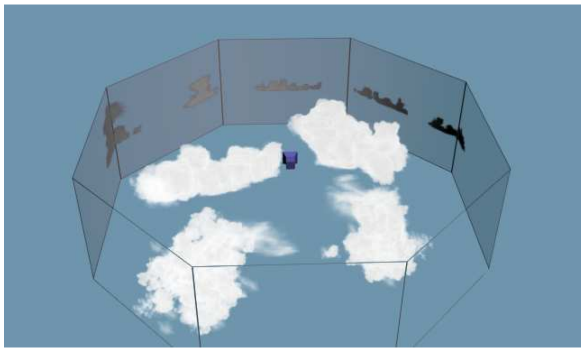

Figure 13: Ring of impostors around the camera. Clouds within the ring are rendered in 3-d.

We render the 8 impostors in fixed world positions facing the center of the ring, and recreate them when the camera or sun position has changed past threshold values. We recalculate all 8 rather than a lazy recomputation, so that the impostors are available if the user suddenly changes the camera orientation. Empirical results show that the user can move through 15% of the impostor ring radius horizontally or 2% of the ring radius vertically before recalculation becomes necessary.

To prevent variability in framerate from the overhead of 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 8 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 crossfade 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. In this situation, the clouds are further away and take up less space on the screen, so there is less overdraw and performance only suffers slightly from 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 8 impostors, each a 256 x 256 texture with 32-bit color, adds up to a video memory cost of $8 * 256 * 256 * 4 = 2$ megabytes uncompressed. When crossfading, both impostor rings are rendered, which adds another 2 megabytes during the transition.

6.3 Performance Results

Our cloud system is implemented using the DirectX API on Windows PC systems. We found that framerate is correlated to both the number of sprites and their sizes. We created a metric that we call the ”cloud block overdraw”, which we calculate by summing the sizes of all sprites in a 16 square kilometer block and diving by the two-dimensional area of that block.

We ran three cloud scenarios in Microsoft Flight Simulator: sparse clouds with an overdraw of 170%, scattered clouds with an overdraw of 200%, and overcast skies with an overdraw of 475%. We ran each scenario with and without impostors, at a resolution of 1024 x 768 with 32-bit color.

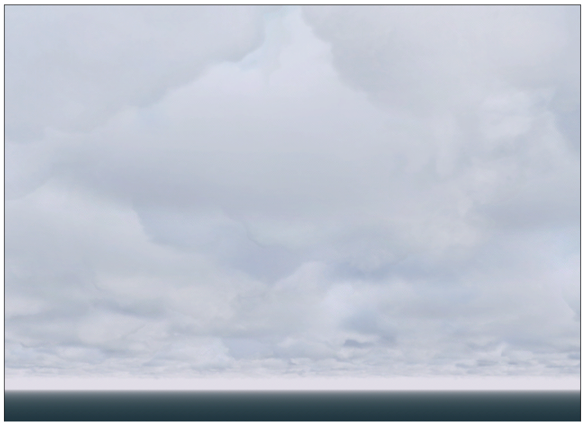

Figure 14: Thick overcast layer covering the sky.

A graph of the results is shown in Figure 15. As we can see, impostors dramatically improve performance on lower-end systems, more so than on faster machines where fill rate is less of a limiting factor. Across both machines, we are able to achieve 15 to 60 frames per second with impostors, maintaining high framerates even in overcast scenarios, which have traditionally been challenging from a performance standpoint. We show a screenshot of the overcast scene we used in Figure 14.

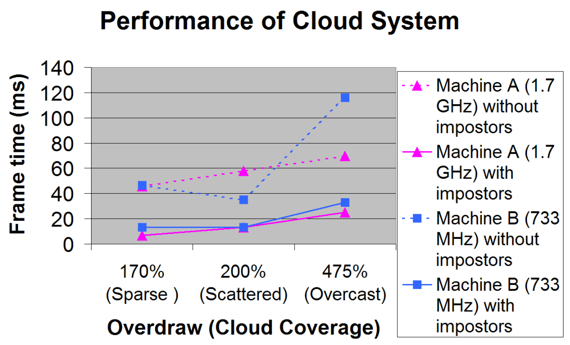

Figure 15: Chart comparing performance with and without impostors, on two systems.

7 Limitations and Extensions

We separate this section into cloud modeling, animation and shading, performance, and extensions to other visuals.

7.1 Modeling

Our system is well suited for creating voluminous clouds but less suited for creating flat clouds. Of the four basic cloud types – cumulus, stratus, cumulonimbus, and cirrus, our system easily handles the first three but has difficulty with cirrus clouds because they are so flat as to be almost two-dimensional. To replicate cirrus clouds in our system, we would need a large number of sprites, which would cause a performance hit. Instead, our system represents cirrus clouds with flat textured rectangles.

Because sprites within each cloud are sorted back-to-front based on distance to the camera, moving the camera can occasionally result in popping as sprites switch positions in the draw order. This is more noticeable at dawn and dusk when directional shading plays a greater role. One potential solution would be to save the previous ordering and fade between sprites when their order changes. In our experience, the popping was not jarring enough to necessitate this solution.

Our in-cloud experience sometimes does not appear as dense as one would expect. One scenario is when the user flies through wispy clouds compromised of relatively few sprites. When the camera is at the cloud's center, only half of the sprites lie in front of the camera, so that the cloud is half as opaque as when the user viewed it from the outside. However, in real life, often the cloud appears most opaque at its center.

Another situation in which the in-cloud experience appears too non-opaque is when the user changes the camera orientation. Suppose the user is flying parallel to the ground. When he moves close enough to the sprites, they lock in an orientation perpendicular to the ground. If he looks down, he will now see large gaps between the sprites. A potential solution is to detect when the user has entered into the core of the cloud and combine an additive fogging effect with our existing in-cloud experience.

7.2 Animation and Shading

Currently clouds do not change shape except for formation and dissipation. In real life, air flow causes clouds to morph over time. We can simulate this by rotating or translating sprites within the cloud, giving the impression that wisps are moving and tumbling with the wind. As groups of sprites are translated, the cloud could condense into itself or break apart into several pieces. We could also fade individual sprites in and out to alter the overall shape of the cloud.

Because our shading model does not simulate the scattering of light, clouds do not cast shadows on themselves or neighboring clouds. Another shading inaccuracy is that directional shading is measured based on angle to the sun rather than density of cloud mass that the light has passed through. For example, we do not get a halo effect when the cloud is between the sun and the camera. One potential solution that fixes both of these problems is to pre-calculate the lit and shadowed regions of the cloud for a set of sun angles. We can load this information at runtime and interpolate based on sun angle.

7.3 Performance

Overdraw accounts for much of the cost of rendering our clouds, so the framerate can vary based on how much of the scene is taken up by clouds. When the camera moves into a cloud, there is a higher amount of overdraw, and framerate tends to drop. One way to alleviate this is to detect when the camera has moved inside a cloud using the bounding box of the sprites, and to add a fog effect. This would allow us to draw fewer clouds in the distance.

Using a ring of impostors can create visual anomalies. Suppose that beyond the impostor ring radius lies a mountain surrounded by clouds. Those clouds are rendered into a single ring of impostors, so that the mountain must draw either in front of or behind all the clouds, rather than behind some clouds and in front of others. This can be mitigated by adding more rings of impostors, but that increases video memory usage.

Impostors can also look wrong due to lack of parallax. The clouds in the impostor do not move relative to each other. Their motion also looks inaccurate relative to the clouds within the impostor ring radius that are rendered as individual sprites.

An area for future work is to take advantage of newer hardware and implement some of our techniques using vertex shaders. We have not yet done so because our application supports a user base with a wide spectrum of machines, many of which have video cards that do not support hardware vertex shaders.

7.4 Extensions to Other Visuals

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 that hard lines can be seen where the sprites intersect the ground polygons, which can be alleviated either by splitting the sprites along the ground or multiplying by a one-dimensional alpha texture based on the altitude.

To extend our system for simulating smoke, we would use darker, wispier sprites. Since smoke is more fluid and moves faster than clouds, we would also need to enhance the movement, potentially by adding some air flow simulation.

8 Acknowledgements

This work was made possible by the artistic contributions of John W. Smith and Adrian Woods. We also thank Jason Waskey, Carl Edlund, Eric Haines, Chris Prince, Ian Pieragostini, Jim Blinn, Mark Harris, and the Microsoft Flight Simulator team.

9 Web Information

The figures in this paper, along with additional screenshots and a link to a short video demonstrating the techniques, are available online at http://www.acm.org/jgt/papers/Wang03.

References

[Bli82] J Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH 82)*, Computer Graphics Proceedings, Annual Conference Series, pages 21–29. ACM, ACM Press / ACM SIGGRAPH, 1982.

[DKY$^+$00] Y Dobashi, K Kaneda, H Yamashita, T Okita, and T Nishita. A simple, efficient method for realistic animation of clouds. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 19–28. ACM, ACM Press / ACM SIGGRAPH, 2000.

[DNYO99] Y Dobashi, T Nishita, H Yamashita, and T Okita. Using metaballs to modeling and animate clouds from satellite images. 15(9):471–492, 1999.

[Ebe97] D S Ebert. Volumetric modeling with implicit functions: A cloud is born. In *Visual Proceedings of ACM SIGGRAPH 1997*, Computer Graphics Proceedings, Annual Conference Series, page 147. ACM, ACM Press / ACM SIGGRAPH, 1997.

[ES00] P Elinas and W Stuerzlinger. Real-time rendering of 3d clouds. *Journal of Graphics Tools*, 5(4):33–45, 2000.

[Har03] Mark Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina at Chapel Hill, 2003.

[HL01] M Harris and A Lastra. Real-time cloud rendering. In *Computer Graphics Forum*, volume 20, pages 76–84. Blackwell Publishers, 2001.

[NDN96] T Nishita, Y Dobashi, and E Nakamae. Display of clouds taking into account multiple anisotropic scattering and sky light. In *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 379–386. ACM, ACM Press / ACM SIGGRAPH, 1996.

[Per85] K Perlin. An image synthesizer. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, Computer Graphics Proceedings, Annual Conference Series, pages 287–296. ACM, ACM Press / ACM SIGGRAPH, 1985.

[Sch95] G Schaufler. Dynamically generated imposters. In *Modeling Virtual Worlds - Distributed Graphics*, MVD Workshop, pages 129–136, 1995.