

# Real-time rendering of 3D clouds

Pantelis Elinas, University of British Columbia, and Wolfgang Stuerzlinger, York University

## Abstract

*The visual realism of a computer graphics simulation of outdoor scenery is greatly enhanced by realistic display of clouds. In this paper, we present an algorithm based on Gardner's work that can render perspectively correct 3D clouds in real-time on current graphics hardware. The use of 3D ellipsoids as primitives permits even close-ups without the viewer noticing the approximation. We describe an implementation using OpenGL.*

## 1.0 Introduction

Many virtual reality (VR) simulations display outdoor environments. Adding natural phenomena to the virtual environment increases the realism of such simulations. One important part of nature that fills much of the field of view in outdoor views is the sky. The sky itself can be approximated simply as a sphere sector or a plane with a color gradient. Many simulations then use texture mapping to add clouds onto the sky. Unfortunately, the 'painted-on' look of this technique is not very convincing. In VR simulations that support flying the viewer can go near or even fly into clouds. All approaches that use simple 2D textures are clearly inappropriate here because 2D approximations [3,6] cause visual artifacts.

Several different approaches have been proposed for perspectively correct rendering of 3D clouds. There are three main approaches: One category of algorithms is based on physics principles. The cloud is rendered by ray tracing, which computes the light propagating through the cloud [1,8,10]. The resulting images are by far the most realistic but they cannot be computed at interactive rates on today's hardware. The second alternative is to use volume rendering. OpenGL simulates this by compositing multiple slices of a 3D texture. The downside is the high texture memory consumption for the texture data, especially if multiple clouds are used. The third class of approaches, used by Gardner and others, is based on rendering 3D textured mapped primitives [2,4,5,6].

In the following sections we present a real-time implementation of 3D Gardner style clouds, as first introduced in [5]. Because our target application (a flight simulator) requires the ability to get close to clouds and that clouds can be viewed from all sides all 2D approaches were ruled out because of insufficient visual quality.

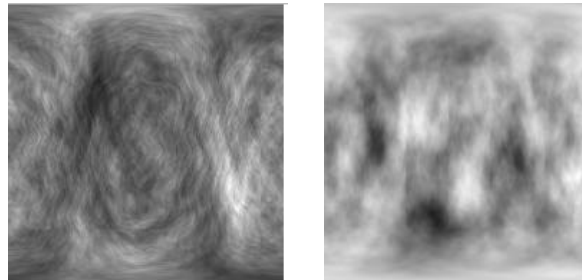
## 2.0 Overview of Gardner's method

Clouds exhibit an irregular structure both at a macroscopic as well as a microscopic level. Furthermore,

they are partly transparent, which is most apparent around the edges. The approach presented in this paper is based on Gardner's approach that uses multiple 3D ellipsoids to simulate the structure of a cloud. Each cloud ellipsoid has an irregular texture and is partially transparent. By compositing multiple cloud ellipsoids together the 3D appearance of an entire cloud is simulated. In this section, we review Gardner's original method. Section 3 will discuss our variation and implementation.

## 2.1 Rendering Cloud Ellipsoids

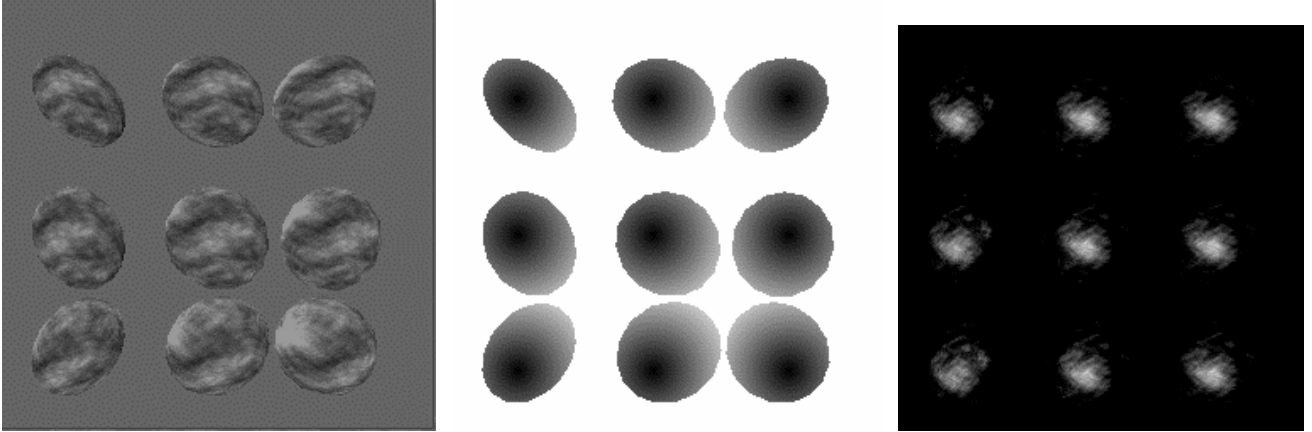
The ellipsoids are textured to simulate the irregular nature of clouds. Both the color and the transparency are varied according to a noise texture. In Gardner's approach the transparency texture is created using spectral synthesis that adds multiple 3D sine waves with increasing frequencies and decreasing amplitude with random offsets. In section 3.2 we will discuss our alternative using Perlin noise. The color of the noise texture is used for rendering the shaded ellipsoid. Unfortunately, it is not possible to use the transparency texture directly because it does not guarantee that the outside of the ellipsoid is more transparent than the inside.



**Figure 1:** Left image: A texture created with spectral synthesis. Right image: A texture created with Perlin noise. Both textures are generated to wrap around a sphere.

## 2.2 Calculating the Ellipsoid Transparency

This section discusses the calculation of the transparency for a single ellipsoid. The ellipsoid should be non-transparent at the center with transparency increasing as we move to the ellipsoid's edge. This closely approximates the transparency of a cloud in nature. Gardner used the following equation:



**Figure 2:** Left: The term  $I_i - T_i$  for 9 perspective correct cloud ellipsoids. Middle: The  $T_2 * (1 - g(x, z))$  term. Right: Alpha channel for the 9 ellipsoids; values shown correspond to OpenGL alpha values (opaque at intensity 1 to full transparency at 0).

$$\mathbf{a} = 1 - \frac{(I_i - T_1 - (T_2 - T_1) \times (1 - g(x, z)))}{D} \quad (1)$$

$$0 \leq \mathbf{a} \leq 1$$

where (as described in [5]),

- $I_i$  is the texture value at that pixel
- $T_1$  is the threshold at the center of the ellipse
- $T_2$  is the threshold at the boundary of the ellipse
- $z$  is the vertical image coordinate
- $x$  is the horizontal image coordinate
- $g(x, z)$  is a 2D function over the shape of the projected ellipsoid. It is normalized to 1 at the center of the ellipse and is 0 at the boundary.
- $D$  is the range of texture function values across which translucence varies from 0 to 1.

Finally, multiple cloud ellipsoids are composited with the computed transparency to create a more complex cloud.

### 3.0 Fast OpenGL Implementation

In this section we describe how the OpenGL 1.2 API [11] can be used to render each of the ellipsoids, to calculate the transparency of each resulting ellipse and finally to combine everything to render a cloud. Our implementation is focused around OpenGL as realized in SGI computer hardware; in other words we have tried to exploit the fast path of the rendering pipeline for SGI

machines. Our approach makes use of advanced OpenGL features such as the color matrix and subtracting blending modes. Most current hardware by other manufacturers supports these new features or will support them in the near future. We begin by describing changes to equation (1) to make it more suitable to hardware rendering.

### 3.1 Modified Rendering Equation

In our implementation we use a slightly modified version of the original equation used by Gardner; our modifications are motivated by limitations of the frame buffer such as its inability to store negative values and by the fact that the frame buffer clamps all values to [0,1]. Although the OpenGL accumulation buffer can store negative values it cannot be used suitably in this application. We use the following version of equation (1) in our implementation:

$$\frac{((I_i - T_1) - T_2 \times (1 - g(x, z)))}{D} \quad (2)$$

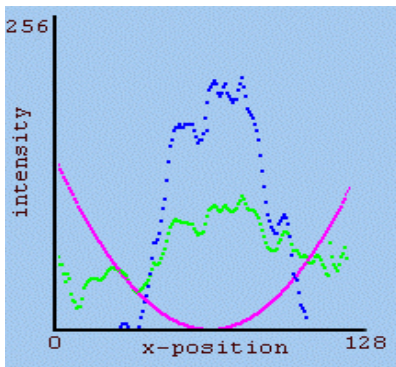
Figure [3] shows plots of this function and its components across the mid-section of an ellipsoid. In section 3.4, we describe how equation (2) is evaluated step by step within the OpenGL pipeline.

### 3.2 Perlin Textures

Because spectral synthesis is harder to control and in order to allow more flexibility in shading the clouds we use Perlin noise textures for the color. Perlin's noise functions are based on synthesizing the result with linearly interpolated output of a random number generator. Figure 1 shows a spectral synthesis and a Perlin noise texture. While it is possible to synthesize such textures on demand with graphics hardware (see [9]), we pre-compute all needed textures at startup.

### 3.3 Rendering the Ellipsoids

To generate the correct final result, each of the cloud ellipsoids must be handled independently i.e. its transparency and shading must be calculated independent of other ellipsoids in the cloud. Furthermore, because we want to be able to get near to the cloud perspectively correct rendering of the ellipsoids is essential. We use the backbuffer and render each ellipsoid into it's own sub-region of it. The buffer is partitioned in  $N \times M$  square regions. The size of the region is selected to maximize the trade-off between image quality and rendering speed.



**Figure 3:** Green: Plot of  $I_r - T_l$ . Pink: Plot of  $T_2 * (1 - g(x, z))$ . Blue: Plot of equation (2) with  $D = 0.25$ .

In order to render each of the ellipsoids with the correct perspective, we do the following:

1. Calculate the ellipsoid's bounding box in eye space.
2. Calculate the projection of the bounding box on the near plane from the points calculated in the previous step.
3. The computed rectangle on the near plane specifies then the (skewed) viewing frustum and the viewport.
4. Render the ellipsoid as desired.

The result of the above steps can be seen in the left part of Figure 2, where 9 ellipsoids are shown rendered so as to evaluate equation (2). The shape and the texture on the ellipsoids vary non-linearly because of the perspective distortion depending on the relative position on the screen. For the figure each of the ellipsoids is rendered within a square viewport of 80x80 pixels.

### 3.4 Calculating the Transparency

Calculating the transparency of the cloud involves the evaluation of equation (2) for each of the ellipsoids. At this point we reduce the transparency calculation from three

dimensions to two. The calculations that follow are over the ellipse that results after rendering the ellipsoid. Equation (2) consists of additions, subtractions and multiplications which means that it is suitable for implementation using OpenGL blending

Notice that the term  $I_r - T_l$  has the potential of returning a negative result. However, this happens only if the lowest intensity in the texture is smaller than  $T_l$ . We evaluate the term  $I_r - T_l$  in the alpha channel of the backbuffer. This is implemented efficiently using the following steps:

1. Disable writing to all but the alpha channel
2. Render a viewport filling flat polygon with  $T_l$  in the alpha channel
3. Evaluate  $I_r - T_l$  by rendering the textured ellipsoid (interpreted as alpha channel data) with subtractive blending.

The second part  $T_2 * (1 - g)$  is computed in the color channels of the backbuffer. The first issue in this equation is calculating the  $g$  function. We illuminate the ellipsoid using a spot light implemented as a projective texture in order to approximate a function that falls off gradually towards the edges of the ellipse. Figure 4 shows the used spotlight texture used which exhibits radial fall off. We already have all the information for setting up the projective texture from the previous step of calculating  $I_r - T_l$ . Figure 5 shows one of the ellipsoids illuminated by the spotlight texture of figure 4. The term  $T_2 * (1 - g)$  is then evaluated as follows:

1. Enable writing to the color channels and disable writing to the alpha channel.
2. Render a viewport filling flat polygon with color 1.0.
3. Load the texture matrix with the perspective transformation for the current ellipsoid and enable automatic texture generation.
4. Render the non-textured ellipsoid with smooth shading and subtractive blending.
5. Set-up multiplicative blending with `glBlendFunc(GL_DST_COLOR, GL_ZERO)`.
6. Render a viewport filling flat polygon with color  $T_2$ .

At this point the alpha channel holds  $I_r - T_l$  and the color channels hold  $T_2 * (1 - g)$ . The next step is to combine the two terms i.e. subtract the latter from the former. Because this operation is the same for each ellipsoid this can be done for all ellipsoids in one step simultaneously as follows:

1. Set-up subtractive blending and enable writing to the alpha channel.

2. Set the color matrix so that the red channel is copied into the alpha channel (all zeros except the lower left entry, which is set to one)
3. Copy all pixels from the frame buffer back onto itself. Because the color matrix is applied before the subtractive blend, the correct result will be computed

Now we have evaluated equation (2) with the exception of the  $I/D$  term in the alpha channel of the back buffer. The  $I/D$  term can be included in the latter step by simply setting the value in the color matrix to  $I/D$  instead of 1. Hence, the scaling adds no additional performance penalty to the calculation.

### 3.5 Calculating the Color

We have now calculated the transparency of each of the ellipsoids that make the cloud. We also need to calculate the color of the ellipsoids. To achieve this, we render each of the ellipsoids into the RGB channel of the back buffer properly lit by the Sun as well as textured using the Perlin noise textures for modulating the shading. This operation is performed using the `GL_MODULATE` texturing mode. The transparency and color of each of the ellipsoids are now stored in the back buffer. What remains is to properly combine the data in the back buffer to render the complete cloud.

### 3.6 Compositing the Cloud

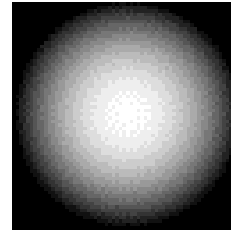
The last step in rendering the complete cloud is to properly combine the data in the back buffer to produce the complete cloud. This is accomplished by repeating the following steps for all ellipsoids:

1. Copy the data from the back buffer into a texture.
2. Setup alpha blending.
3. Setup an orthographic projection and viewport that fills the screen.
4. Calculate each ellipsoid's bounding box in eye space and calculate the 2D bounding rectangle on the near plane.
5. Render a textured rectangle in place of the 2D bounding box with the texture created in step 1.

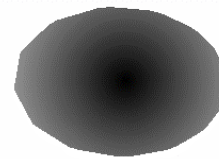
Note that it is necessary to render all ellipsoids back to front to get the correct result from the blending operation.

### 4.0 Performance and Visual Issues

In this section we discuss a few performance issues and present alternative ways to implement the above approach. Note that the optimal combination of techniques that provides the shortest rendering time depends strongly on the used platform. Unfortunately, it is necessary to



**Figure 4:** Texture with radial fall-off.



**Figure 5:** Ellipsoid illuminated using one minus the spotlight texture of Figure 4.

benchmark each combination of hardware, OpenGL implementation, and CPU characteristics to determine the optimum.

#### 4.1 Calculation of $T_2*(I-g)$

In section 3.4 we described how equation  $T_2*(I-g)$  is computed. Depending on the hardware platform there may be more efficient methods to get the same result. The first alternative is to rewrite the equations as  $T_2-T_2*g$ . We calculate  $T_2*g$  by rendering the ellipsoid with color  $T_2$  and using `GL_MODULATE` as texturing mode. We can achieve the subtraction in the same step if we clear the buffer to color  $T_2$  and use subtractive blending. An alternative to evaluate the term  $T_2*g$  is to use the OpenGL scale operation to scale the value for  $g$  by  $T_2$ .

#### 4.2 Resolution Issues

Our current implementation approximates each ellipsoid with  $10 \times 10$  rectangles. To increase the visual accuracy if the ellipsoid is near the viewer and conversely to increase performance if the ellipsoid is far away this resolution could be adapted to the current screen size of the ellipsoid.

The current implementation uses a constant size region for each ellipsoid. By adapting the size of the regions one can create more accurate images, as less image resampling and/or interpolation artifacts will appear. Moreover, performance will be maximized. The downside is that this requires a more sophisticated algorithm that determines how to tile the frame buffer with arbitrary rectangles.

Furthermore, this technique cannot be used directly with the compositing method discussed in section 3.6 as textures in OpenGL need to be have dimensions that are powers of 2.

A viable alternative is to compute the screen size of each ellipsoid and round it up to the nearest power of 2. This allows the use of a simpler frame buffer tiling algorithm that works only for images with powers of 2. A good heuristic is to sort the tiles by size and then allocate with a first fit strategy.

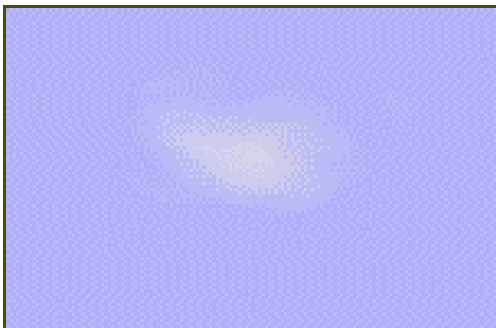
If there are many ellipsoids then the required buffer size may exceed the available space. This can be addressed by doing multiple passes where each of the passes creates a different subset of the ellipsoid images.

### 4.3 Visibility of Other Objects

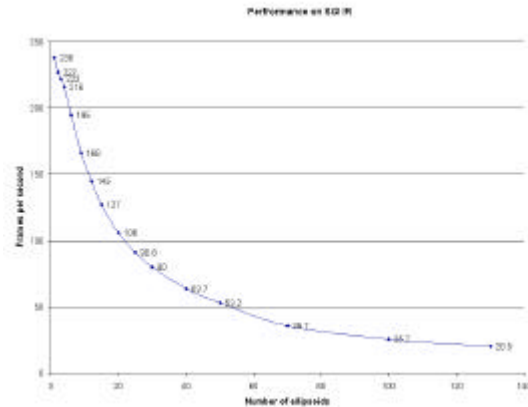
It is possible to extend the presented approach to visualize objects that fly ‘through’ the clouds. Because we use alpha blending to composite the image this extension must first render all opaque objects into the color and z-buffer. Then all the transparent objects (including the cloud ellipsoids) are rendered back to front. For each ellipsoid the z-buffer contents can be generated with a 2-pass approach. The first pass renders a flat shaded ellipsoid only into the z-buffer and sets the stencil buffer wherever the ellipsoid is in front. The second pass uses the stencil buffer to composite only the visible pixels of the ellipsoid. Note that this approximates the correct result only to the extent that the cloud is represented as a collection of ellipsoids. The exact interaction of each ellipsoid with the object cannot be modeled with the current approach.

### 5.0 Examples and Discussion

In this section we present a few samples of real-time rendered 3D clouds. Figure 6 shows a single ellipsoid cloud rendered at 238 fps on a SGI Onyx2 computer. Note that the geometry is rendered three times: first to generate  $I_r$ , a second time to generate the  $g$  function, and a final rendering to get the shaded color.



**Figure 6:** Rendering of cloud made of a single ellipsoid.



**Figure 7:** Performance statistics for an SGI Onyx2 for clouds of different complexity i.e. composed of different numbers of ellipsoids.

The image in figure 8 presents a more complicated cloud rendered using 27 ellipsoids at 90 fps on the same machine. Figure 9 presents another cloud rendered at 80 fps on the Onyx machine. Figure 10 shows two clouds made of a total of 6 ellipsoids. Here it is obvious that even a small number of ellipsoids is sufficient to produce realistic looking clouds. Figure 7 shows performance statistics for clouds of different sizes of clouds i.e. clouds made of different numbers of ellipsoids.

We have taken precautions to optimize the rendering of the geometry for our implementation platform. We are currently limited by texture memory transfer bandwidth. On other platforms performance may be limited by geometry processing, as the number of ellipsoids direct influences the rendering speed, too.

This technique works best for clouds at medium distance. For far away clouds it is not necessary to simulate the 3D structure and impostors are more appropriate. For clouds that are extremely close to the viewer the approximation introduced here may not be good enough and a volumetric representation should be used. We are also considering a port to one of the new multi-texturing graphics cards. Finally, we will look into adding the ability to render objects within the cloud and better calculation of the illumination of the cloud by the Sun.

### Acknowledgments

This research was supported in part by various NSERC programs. Furthermore, the authors would like to thank Mike McCool for valuable feedback on an early version of this paper.

## Web Information

Please consult the following WWW page for additional material:

<http://www.acm.org/jgt/papers/ElinasStuerzlinger00>

## REFERENCES

1. Blinn F. James, Light Reflection Functions for Simulation of Clouds and Dusty Surfaces, *Computer Graphics*, 16 (3), July 1982.
2. Ebert S. David *et al.*, *Texturing and Modeling: A Procedural Approach*, 2nd edition, Academic Press, 1998.
3. Dobashi Y., Kaneda K, Yamashita H, Okita T and Nishita T, A Simple, Efficient Method for Realistic Animation of Clouds, *Siggraph* 2000.
4. Gardner Y. Geoffrey, Simulation of Natural Scenes Using Textured Quadric Surfaces, *Computer Graphics*, 18 (3), July 1984.
5. Gardner Y. Geoffrey, Visual Simulation of Clouds, *Computer Graphics*, 19 (3), 1985.
6. Hugo Elias, Cloud Cover, [http://freespace.virgin.net/hugo.elias/models/m\\_clouds.htm](http://freespace.virgin.net/hugo.elias/models/m_clouds.htm)
7. Hugo Elias, Perlin Noise, [http://freespace.virgin.net/hugo.elias/models/m\\_perlin.htm](http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)
8. Kajiya T. James, Von Herzen P. Brian, Ray Tracing Volume Densities, *Computer Graphics*, 18 (3), July 1984.
9. Neyret F. Mine A., Perlin Textures in Real Time Using OpenGL, RR-3713, INRIA, 1999.
10. Nishita Tomoyuki *et al.*, Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light, *Siggraph* 1996, pp 379ff.
11. Mason Woo, Jackie Neider, Tom David, Dave Shriener, OpenGL Architecture Review Board, *OpenGL 1.2 Programming Guide*, 3rd edition, Addison-Wesley, 1999.



**Figure 8:** Cloud constructed of 27 ellipsoids.



**Figure 9:** Cloud made out of 30 ellipsoids



**Figure 10:** Cloud constructed of 6 ellipsoids.