Clouds and Stars: Efficient Real-Time Procedural Sky Rendering Using 3D Hardware

Timothy Roden, Ian Parberry
Department of Computer Science & Engineering
University of North Texas, P.O. Box 311366, Denton, Texas 76203
http://www.cs.unt.edu

roden@cs.unt.edu, ian@cs.unt.edu

ABSTRACT

Real-time virtual reality applications, including games, increasingly use outdoor environments. A common task in an earth-type environment is to render a sky that is realistic both in terms of imagery and physics. Programmable graphics hardware offers the opportunity to procedurally generate and render a highly realistic sky at a minimal cost. We propose an integrated set of efficient algorithms that run in graphics hardware for interactive sky rendering that is fully parameterized for real-time control. Features of our method include multi-layered dynamic clouds and stars that individually flicker at varying intensity and rate.

Keywords

Real-Time, Procedural, Clouds, Sky, Stars, Game, Virtual, Environment

1. INTRODUCTION

The creation and rendering of clouds is the major challenge in rendering realistic skies. Two approaches to sky rendering have traditionally been used in games and similar applications. In the static approach, one or more photographs are textured onto a 3D model of the sky. For example, a cloud texture can be overlaid on top of a blue sky texture. The cloud texture can then be rendered offset over time to produce an animation. While the resulting sky can be very convincing given a high quality photograph, the lack of dynamics is a significant drawback. The second approach involves creating clouds procedurally. Procedural cloud generation can be divided into two categories, *volumetric* and *planar*, based on how the clouds are represented.

Volumetric techniques attempt to model the 3D space occupied by a cloud [1]. Planar methods model clouds as flat images that can be rendered as textures on a 3D model of the sky such as a box or dome [2,3,4]. Harris [5] used a hybrid approach in which a volumetric representation is mapped onto a series of flat planes called *impostors* [6]. The choice of which representation and rendering scheme to use depends on how the sky will be used. For a flight simulator in which an airplane is allowed to fly through clouds, a volumetric technique is appropriate. If the viewer is constrained to ground level the planar method could be used. In this paper we will present algorithms for planar sky rendering using a sky dome (Figure 1). The viewer is therefore constrained to ground level or near ground level.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACE2005, June 15-17, 2005, Valencia, Spain.
Copyright 2005 ACM 1-59593-110-4/05/0006…$5.00.

2. SKY REPRESENTATION

We use three separate textured 3D models to render the sky: a sky dome, a cloud dome and a flat square for the sun. As Figure 1 shows, we place a sky dome over the virtual world. The dome is always centered on the user. The sky dome is rendered using three textures blended together depending on the time of day. A blue gradient texture is used for the day sky while two textures representing stars are used for night.

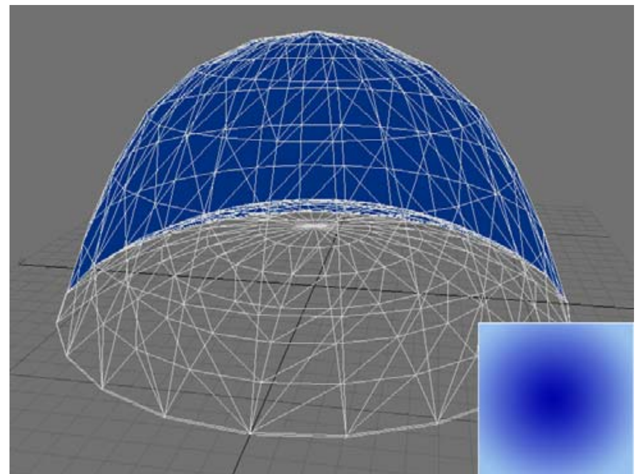

Figure 1. Cloud dome (gray) enclosed within sky dome (blue). The sky dome is textured by blending a blue gradient (inset) with the star textures.

Figure 2. Cover = 0.69, Height = 1.0, Density = 0, Sunset = 0.72, Speed = 0.1, Time = 0.35

A cloud dome is place inside the sky dome. Depending on cloud cover the cloud dome will be rendered partially transparent allowing the sky dome to be visible. A simple flat square is used for the sun with an appropriate texture. The sun is drawn so that it always faces the screen.

Our current implementation uses Microsoft DirectX 9. Each of the three models is rendered using a Direct3D version 2.x pixel shader. Six parameters enable real-time control of the shaders: *cover*, *height*, *density*, *sunset*, *speed*, and *time*. *Cover* is the amount of cloud cover. *Height* is a measure of the clouds over the viewer. *Density* controls the density of the clouds. *Sunset* controls a red gradient that can be applied to the clouds. *Speed* governs the rate of movement of the clouds and is equivalent to wind speed. *Time* is a measure of the time of day. All parameters have a range of 0 to 1.0. To facilitate development in our test application we have created a set of on-screen sliders bars to control the six parameters, as shown in Figure 2.

3. CLOUDS

Real-time procedural cloud generation became practical with the introduction of programmable graphics hardware. Several authors have suggested using Perlin Noise [7] to generate procedural cloud textures. Pallister [3] proposed the idea of combining four noise textures to create clouds. The first texture, called the first octave, is used to generate the overall shape of the clouds. The second octave is scaled down and blended in to produce detail. Similarly, the third and fourth octaves are scaled to produce finer detail in the resulting texture. To achieve animation, Pallister suggested the noise textures be continually modified by interpolating with a second set of dynamically generated noise textures. Dube [2] used a similar approach with eight octaves and added a ray tracing algorithm in the shader to compute realistic per pixel lighting. A disadvantage of this is the shader takes considerable time to render, making it prohibitive at high screen resolutions. A problem inherent in both methods is the clouds can exhibit a high degree of regularity.

A virtual environment that requires a realistic sky is likely to have many other demands for high fidelity content such as terrain. With this in mind we have four goals for our procedural cloud shader. First, we want the shader to be as efficient as possible both in terms of execution time and resources. Second, we want to use as few textures as possible. Third, we want the shader to be fully parameterized to allow precise real-time control. Fourth, we want the clouds to appear unique over time and not exhibit a visible pattern.

We use four textures in our cloud shader as shown in Figure 3. A 512x512 grayscale photograph of clouds is blended with a 512x512 pre-computed noise texture to generate the overall shape of the clouds. The result is modulated by a detail texture. A gradient texture is used to optionally add a red tint to the final color for a sunset effect. In contrast to a purely procedural approach to generating the textures we found that an actual photograph combined with the two noise textures produced more interesting results as compared with using three noise textures. In practical application, any such photograph would need to be chosen carefully to achieve the desired aesthetic effect. For best results we recommend a photograph with a relatively thin cloud layer. The primary objective of using the photograph is to break up the regularity found in the noise textures in order to produce a more natural looking result.

Since each of the four textures used to generate the clouds is a monochrome image, all four can be stored together in a single four-channel RGBA texture. For example, the photograph can be stored in the red channel, the procedural texture in the green channel, the detail texture in the blue channel, and the gradient in the alpha channel. For simplicity, in the next section we present the shader code with each texture loaded separately. This is also the approach we have chosen for testing and development. In the final production version of the shader, however, we expect to pack the textures in order to save texture memory and speed the execution of the shader.

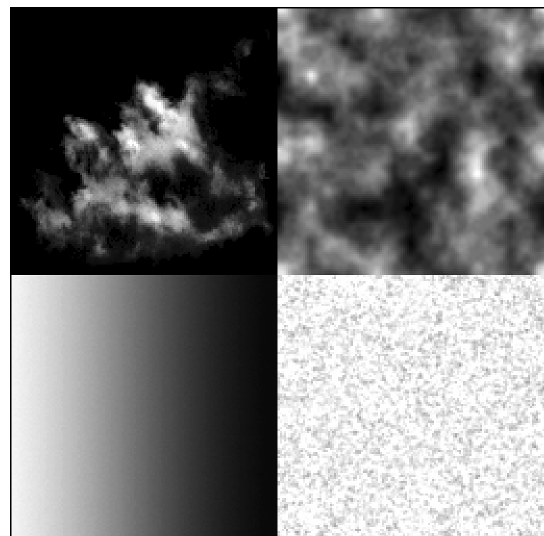

Figure 3. Cloud textures: photo of clouds (upper left), pre-computed noise (upper right), detail (lower right), sunset gradient (lower left).

Figure 4. Cover = 0.57, Height = 0.52, Density = 0.84, Sunset = 0.43, Speed = 0, Time = 0.72

3.1 The Cloud Shader

The cloud dome shader is invoked once per frame for each pixel drawn into the frame buffer. The shader is written using Microsoft High Level Shader Language which has a C-like syntax.

```
[ 1] float cover, sunset_amount, density, amount_sunlight;
[ 2] float4 Main (
[ 3]    float2 tc0       : TEXCOORD0, // first cloud layer
[ 4]    float2 tc1       : TEXCOORD1, // second cloud layer
[ 5]    float2 tc2       : TEXCOORD2, // sunset gradient
[ 6]    float2 tc3       : TEXCOORD3, // detail texture
[ 7]    float4 lighting : COLOR ) : COLOR
[ 8] {
[ 9]    float4 c, cloud1, cloud2, sunset, detail, sun;
[10]    cloud1  = tex2D (InputImage0, tc0);
[11]    cloud2  = tex2D (InputImage1, tc1);
[12]    sunset  = tex2D (InputImage2, tc2);
[13]    detail   = tex2D (InputImage3, tc3);
[14]    c = cloud1 + cloud2 * (1 - cloud1);
[15]    c.a -= (1 - cover);
[16]    cover = clamp (cover, 0.05, 1);
[17]    c.a *= (1 / cover);
[18]    c.rgb = c.rgb + detail.rgb * (1 - c.rgb);
[19]    c.a *= lerp (detail.a, 1, density);
[20]    if (c.a > 0) {
[21]        c.rgba += (density * c.a * 2);
[22]        if (c.r > 1)
[23]            c.rgb = 1 - (c.r-1) * 0.5;
[24]   }
[25]   c = clamp (c, 0, 1);
[26]   sunset.a *= sunset_amount;
[27]   c.rgb = c.rgb * (1-sunset.a) + sunset.rgb * sunset.a;
[28]   c.rgb *= amount_sunlight;
[29]   return c;       // output final pixel color
[30] }
```

Line 1 lists four parameters passed to the shader by the application. The amount of sunlight is a value based on time. The other three parameters have already been described. Lines 10-13 load a pixel from each of the four textures. Line 14 blends the photograph and noise texture. Lines 15-17 reduce the amount of cloud cover and brighten the remaining clouds. Line 18 factors in the detail texture. Line 19 has the effect of making dense clouds smoother and less dense clouds somewhat bumpy. Lines 20-24 adjust the relative intensity of the clouds based on density with denser clouds becoming darker. Line 26-27 blends a sunset color according to the sunset gradient texture.

It is significant that only four of the six procedural parameters are used in the shader. Since the shader is invoked on a per pixel basis, not all of the six parameters are needed. In fact, for performance reasons, any per frame calculations should not be done in the shader but rather in the application prior to rendering and invoking the shader. The height parameter is used in the application to scale the texture coordinates. By stretching the texture across the cloud dome, the clouds appear to drop lower in the sky as shown in Figure 4. In comparison, the clouds in Figure 2 appear higher. The speed parameter is also used to adjust the texture coordinates in the application. By adding a value based on speed to the texture coordinates, the clouds appear to move over time.

3.2 Improvements

The approach proposed by Pallister of continually interpolating noise textures, while perhaps expensive in terms of execution time, produces clouds that change shape over time. We have found a simple method to achieve shape changes over time by allowing the two primary cloud textures to move in slightly different directions and at slightly different speeds. Since the primary shape of the clouds depends on the combination of the two textures the result is a gradual shape change.

Still another improvement to our basic shader is to add another layer of clouds. It is quite often the case that real skies exhibit clouds with quite different properties. For example we can add a thin upper layer of clouds by sampling our procedural cloud texture with its texture coordinate stretched in one dimension and adding the result to the final rendering as shown in Figure 5.

Figure 5. Multi-layer clouds rendered by adding another sample from the second cloud texture.

4. STARS

For the night sky we want a vast array of stars with each star flickering individually in such a manner that there is no visible pattern in any part of the night sky. A pixel shader is a natural place to put this functionality since the shader is invoked on a per pixel basis. The pixel shader for the sky dome blends between three textures. The blue gradient texture shown in Figure 1 corresponds to daylight and, depending on the time parameter, is blended with two textures to simulate the night sky – a *star texture*, shown in Figure 6, and a *monochrome texture* designed to simulate larger clusters of stars such as a galaxy, shown in Figure 7.

Figure 6 shows the four channels of a single 256x256 RGBA texture that encodes the data needed for the stars. This texture is tiled over the sky dome to produce the desired density of stars. The red channel, which is mostly black, represents the average intensity of each star. If the pixel is black then there is no star at that position. The shader uses a sine wave to change the intensity of stars over time. The green channel encodes the rate of flicker and essentially scales the sine wave in the horizontal dimension. The blue channel encodes the variance of each star and scales the sine wave in the vertical dimension. Finally, the alpha channel encodes a starting intensity change and can be viewed as an initial offset on the horizontal axis of the sine wave.

The sky dome shader factors in a 128x128 galaxy texture that is stretched over the entire sky dome (as shown in Figure 7) to make a more realistic and interesting night sky. In addition, stars inside the galaxies receive an additional intensity increase in order the make the galaxy effect more dramatic.

5. CONCLUSIONS AND FUTURE WORK

With increases in the power and flexibility of graphics hardware it becomes easier to render realistic imagery at interactive rates, but it is still necessary for interactive techniques to be both fast and resource efficient. For procedural sky rendering, our tests have proven that our approach is both efficient and produces good visual quality. Currently, our test application runs, with no optimizations, at approximately 70 frames per second in 1600x1200 resolution on a Pentium 4 3GHz computer with an NVidia GeForce 6800 Ultra graphics card. By using texture packing it is possible to create a cloud shader with far greater realism at a minimal cost in texture memory.

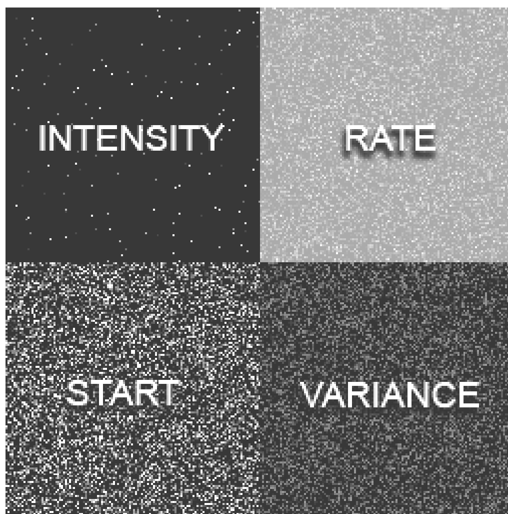

Figure 6. 128x128 section of stars texture with data encoded in separate red (upper left), green (upper right), blue (lower right), and alpha (lower left) channels.

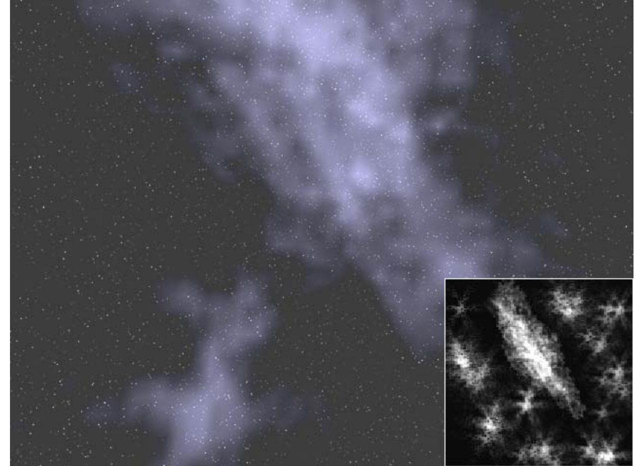

Figure 7. Night sky (contrast applied) with 128x128 galaxy texture (inset).

Another potential improvement to the proposed cloud shader is to store the cloud textures in high dynamic range textures [8]. Using the standard 8-bit per color textures often results in noticeable aliasing artifacts, particularly when the cloud cover is diminished.

6. REFERENCES

[1] Ebert, David, et al. Texturing and Modeling: A Procedural Approach, 3rd edition. Morgan Kaufmann Publishers, San Francisco, 2003.

[2] Dube, J. “Realistic Cloud Rendering on Modern GPUs,” in Game Programming Gems 5 (Kim Pallister, ed.), Charles River Media, 2005, pp. 499-505.

[3] Pallister, K. ‘Generating Procedural Clouds Using 3D Hardware,” in Game Programming Gems 2 (Mark Deloura, ed.), Charles River Media, 2001, pp. 463-473.

[4] St.-Laurent, S. Shaders for Game Programmers and Artists, Thomson Course Technology, Boston, 2004.

[5] Harris, M. and Lastra, A. “Real-Time Cloud Rendering,” Computer Graphics Forum (*Eurographics 2001 Proceedings*), 20(3):76-84, September 2001.

[6] Schaufler, G. “Dynamically Generated Impostors”, GI Workshop, *Modeling - Virtual Worlds – Distributed Graphics*, November 1995, pp. 129-135.

[7] Perlin, K. An Image Synthesizer, in *Proceedings ACM SIGGRAPH*, 1985, pp. 287-296.

[8] Debevec, P. “Image-Based Lighting”, in IEEE Computer Graphics and Applications, Jan/Feb 2002.