Generating and Rendering Procedural Clouds in Real Time on Programmable 3D Graphics Hardware

Conference Paper · January 2006

DOI: 10.1109/INMIC.2005.334442 · Source: IEEE Xplore

3 authors, including:

Mahmud Hasan
RMIT University
5 PUBLICATIONS **6** CITATIONS

SEE PROFILE

Generating and Rendering Procedural Clouds in Real Time on Programmable 3D Graphics Hardware

M Mahmud Hasan
ReliSource Technologies Ltd.
mmahmudhasan@yahoo.com

M Sazzad Karim
TigerIT (Bangladesh) Ltd.
skremon@yahoo.com

Emdad Ahmed
Lecturer, NSU;Graduate TA,
Wayne State University, USA
emdadahmed@hotmail.com

Abstract

This paper discusses a process of generating and rendering procedural clouds for 3D environments using programmable 3D Graphics Hardware. Cloud texture generation is performed using Perlin noise [1] and turbulence functions. Our implementation is done in OpenGL supported GPUs with programmable vertex & fragment processing pipeline that supports OpenGL Shading Language (GLSL) [2]. We have performed a performance benchmark against other existing implementations [3] and found very convincing results, as our approach yields greater FPS than those reported earlier in the literature, as well as our solution is platform independent and portable. The technique can be used in real-time graphics applications, games, film special effects and visual simulations etc.

Keywords

Perlin Noise, Procedural cloud, GPU, Render to Texture, PBuffer, GLSL, Real time Rendering, FPS.

1. INTRODUCTION

Simulation of *natural clouds* in 3D environment for graphics applications is an interesting topic for graphics researchers and application developers. Many techniques have successfully been implemented to generate realistic images of clouds in the sky but simulating them in *real time* applications has been a great challenge. Generating quite natural and almost *photorealistic* clouds in real time using consumer level hardware requires some tradeoff between performance and accuracy. Traditional approaches for dynamic cloud modeling for graphics applications, games, film special effects and visual simulations suggest many different techniques. Non real time *volumetric* simulation [10] is a common approach for film special effects that generates very photorealistic, natural cloud model using industry level workstations and graphics hardware. Some of the latest game engines took the advantages of great polygon processing power in consumer level hardware to generate *billboard clouds* [8] for flight simulation or cinematic sequences. Though this technique allows great control in cloud formation and lighting effects, the dynamic nature of natural clouds is noticeably affected by this approach. Procedural techniques for cloud modeling (primarily based on noise, generated using Pseudo-Random Numbers [9]) is another great research topic in Computer Graphics. This technique focuses on the fractal properties of natural clouds and uses randomly seeded noise of different frequency to generate that fractal property. Still, this approach of cloud modeling suffers from performance lack in real time applications for expensive computational functions. Procedural approaches requiring noise generation and texture smoothing in CPU cannot deliver decent performance in real time 3D applications. Here we present our own procedural approach to render quite natural real time clouds using some latest hardware features like vertex and pixel shaders, high performance pixel pipelines to balance the computational load among CPU and graphics hardware. The goal of our approach is to perform all possible computations in the graphics hardware, thus minimizing the work load of the CPU.

2. RELATED WORK

Striving to render real-time clouds started many years back with older *GPUs (Graphics Processing Unit)* that did not have programmable vertex or pixel pipelines. Kim Pallister [3] presents a technique that used features such as: *alpha blending*, *bilinear filtering* and *render to texture surfaces* using *DirectX*. John C. Hart's paper [4] explains the usage of programmable GPUs and *OpenGL* to generate *perlin noise* using pixel shaders that can be used for cloud generation. But it used high performance industry standard SGI Workstations rather then low cost consumer level graphics hardware. *Geforce3* by *nVidia* was the first fully programmable GPU available for general consumers. But in Geforce3 programmability of *vertex* and *fragment pipelines* were not very intuitive to use and they were exposed through vendor specific *register combiners*.

3. CONTRIBUTION OF THIS PAPER

This paper explains the technique we used to implement dynamic cloud for the real time applications using consumer level graphics hardware. The approach of this work was to implement a computationally efficient technique, the dynamic cloud phenomenon, for consumer level application (Computer games, Special effects or visual simulations) using some latest hardware features facing the portability issue. Improvement of performance and acceptability of the technique to almost all hardware vendors was our prime goal. Though our technique neither claims for best performance in all hardware platforms (GPU hardware) nor boast for ultimate result for photorealism, we tried to get some respectable output for toady's games and real time application in respect to dynamicity, visual quality, performance and portability over traditional practice. We also believe that our technique is the first step to dynamic, photorealistic cloud modeling for the consumer level graphics applications that can show some positive path for future implementations.

4. HARDWARE & SOFTWARE REQUIRMENT

All of the recent GPUs are designed to support *OpenGL 2.0* that features a high level shading language *GLSL*. Our goal was to implement our technique for such GPUs that will become a standard for consumer level graphics hardware. We used *beta drivers* supporting OpenGL 2.0 and we have also used the beta SDK provided by *3DLabs* for demonstrations. We could also use CG for this purpose, but GLSL is definitely going to be the most commonly used and most robust shading language to be used with OpenGL based software. We have tested our application in an *ATI Radeon 9500Pro*, which was the first card to support OpenGL 2.0 and GLSL. CPU requirement was not that high, we used a 1.3 GHz *Celeron* powered machine with 256 Megabyte of RAM for testing, as most of our computations are done in the GPU. Specifically, the following OpenGL extensions are required [2]:

```
GL_ARB_fragment_shader
GL_ARB_vertex_shader
GL_ARB_shader_object
GL_ARB_shading_language_100
```

5. TECHNIQUES USED

5.1 Noise Generation

Using some Perlin Noise [1] techniques we usually generate multiple *octaves* of noise textures. However, generating noise in real time is quite computationally expensive. So, we generated a relatively large (512x512) noise array as pre-calculation phase and saved the array as texture. We used that texture as a *noise lookup table* to build noises of smaller resolution (e.g 64x64; 128x128). Addition of noise octaves is done using graphics hardware so we only need to *pre-generate* a simple noise lookup texture (table) using *seeded* random number generator that generates a color value of [0,1] for each pixel.

5.2 Rendering to Texture Surfaces using *PBuffer*

Rendering directly to texture surfaces is exposed in OpenGL through using *WGL_ARB_pbuffer, WGL_ARB_render_texture* extensions in the windows platform [5][6] (other extensions are available for *X-Windows* and *MAC*). Pbuffer is a *pixel buffer* that works like an off-screen *frame-buffer* to render something and also can be used as a render-target for textures. That is, we can directly render into a texture instead of rendering to the frame-buffer and then copying pixels from it to make a texture object.

5.3 Creating Texture from a Portion of Frame-Buffer Data

We also need the Pbuffer to render the noise lookup table and cut some portion off the buffer to create a random noise. For this we used *glCopyTexSubImage*(...)* function to copy pixel data from the frame-buffer directly into a texture surface [7].

5.4 Texture Filtering

We need to apply appropriate *filtering techniques* for *smoothing* the textures and stretching *(up-sampling)* them to a larger one. OpenGL provides *bilinear magnification filtering* functions that are good enough for this purpose [7].

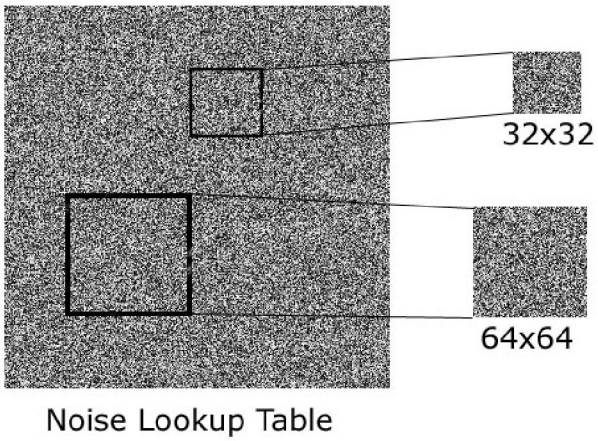

Figure 1: Noise Generation from Lookup Table

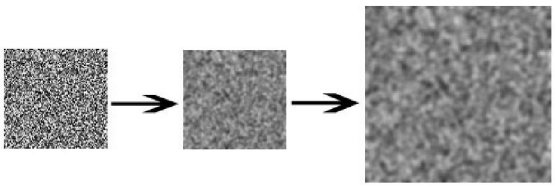

Figure 2: Smoothing a 64x64 noise and then up-sampling to 512x512 using bilinear filtering

5.5 Vertex & Fragment Program

The *texture coordinates* for multiple textures were assigned through *Vertex Programs* for *mutitexturing*. Most part of our technique of combining the noise octaves is done through *Fragment Programs* (or *pixel shaders*) programmed via GLSL, such as applying *interpolation*, *color value subtraction* and *exponentiation*.

6. IMPLEMENTATION

6.1 Generate Noise Octaves

We generated 4 noise octaves for our implementation of resolutions: 32X32, 64X64, 128X128, and 256X256. The noises are generated by cutting a texture portion randomly from the larger (512X512) noise lookup table as shown in Figure 1. However, for performance cost it is impractical to update them each frame, so we kept two textures per octave and updated them after some interval. We updated the octave with highest frequency (256X256) *most recently* and octave with lowest frequency (32X32) *least recently*. We made sure that, in a given frame not more than one octave gets updated. We used different update period values that will not be same at a given frame (e.g. 1.0, 2.01, 4.05, 8.11 seconds respectively for each octave) (those values were chosen based on trial-and-error so that performance do not degrade sharply in some interval/ regular pattern) to balance the GPU work load.

6.2 Smoothing and Up-sampling Noise Octaves

When we render the initial noise texture for each octave into a texture surface (using Pbuffer) the OpenGL bilinear texture filtering smoothes the texture. After each 'noise update' we need to up-sample or stretch each noise octave texture to the resolution of the final output (512X512) (Figure 2). For this OpenGL bilinear texture magnification filtering gives acceptable result when we render the octaves into a 512X512 texture. With this process we eventually get 2 textures for each octave that are ready to be send to Fragment Programs for making the composite turbulent noise.

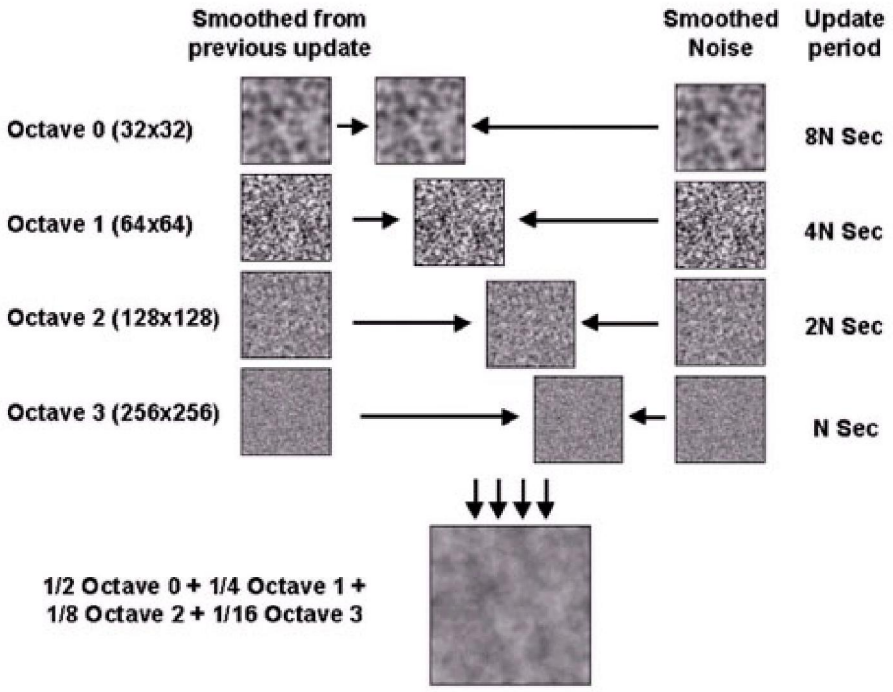

Figure 3: Combining 4 noise octaves to generate the turbulent noise (taken from [3])

6.3 Interpolating the Octaves with Previous Updates

We keep two noise textures (we call them front and back) for each octave and we update them after a certain period. We calculate a value based on linear interpolation in range [0, 1] that determines how much we need to interpolate from one texture to another. We keep interpolating from the back texture to front and after we reach to the front texture (value of 1) we update the back texture and interpolate from front to back texture and vice versa. The interpolation value for each octave was passed to the shader in each frame. In the Fragment Program we computed one interpolated texture element (*texel)* value per octave using this *interpolant* as shown below:

```
vec4 Octave1 = (Interpolant1*texval1 +
(1.0- Interpolant1)*texval2);
vec4 Octave2 = (Interpolant2*texval3 +
(1.0- Interpolant2)*texval4);
vec4 Octave3 = (Interpolant3*texval5 +
(1.0- Interpolant3)*texval6);
vec4 Octave4 = (Interpolant4*texval7 +
(1.0- Interpolant4)*texval8);
```

6.4 Combining the Octaves into a Single Turbulent Noise

After interpolation we get one texel value per octave and we combine them to generate the *turbulent noise* output by taking a *weighted contribution* for each octave. Generally we take contributions that are multiples of two and each the octave's contribution is half of the one at its next lower frequency. We used the following expression:

```
Intensity= (Octave1*0.5 + Octave2*0.25
+ Octave3*0.125 + Octave4*0.0625).red;
```

At this point we get the turbulent noise we need to make it look like clouds. The process is shown in the Figure 3.

6.5 Composing the Final Cloud Texture

To compose the final cloud we use two variables: one for *density* to calculate the *cloud coverage* over the sky and another for *sharpness* to increase the *intensity of fluffiness* of the clouds [figure 4]. Normally there are some clear parts of the sky, for this we subtract a value for each fragment and *clamp* the value to 0:

```
intens = intens - density;
if(intens < 0.0)
   intens = 0.0;
```

Next, we increase the sharpness of the cloud portions using some exponential function:

```
intens = pow(sharpness, intens);
```

Then we invert the value as the exponential function reverses it:

```
intens = 1.0 - (intens*1.0);
```

Finally, we take a sky color and a cloud color and perform a *linear blending* between them with the final intensity value we just computed. The *mix(..)* [2] function linearly blends the two values base on the equation: x*(1-a) + y*a.

```
vec3 color = mix(SkyColor, CloudColor,
intens);
```

The density and sharpness factors can be changed to alter the look of the clouds. In Figure 4 we can see some results from our generated cloud image.

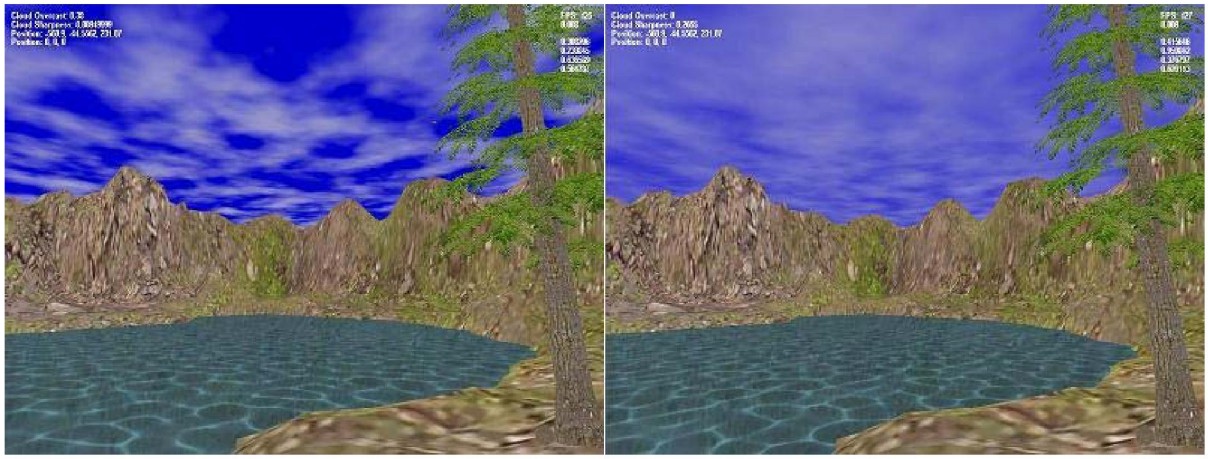

Figure 4: Dynamically generated cloud with different parameters. (Left): High density, low coverage cloud, (Right): Low Density, high coverage cloud.

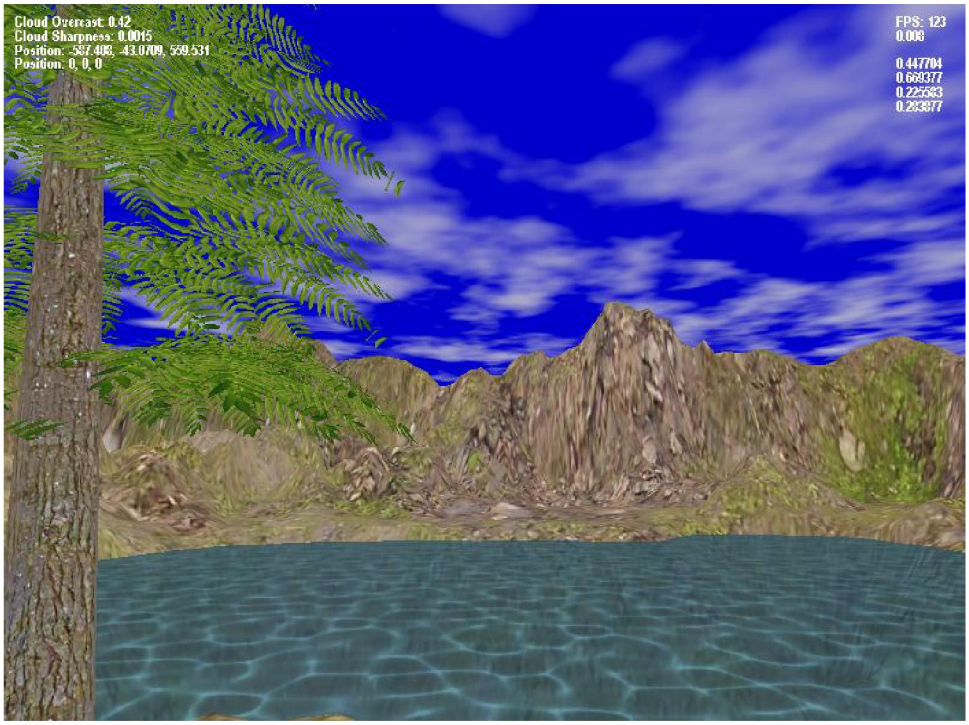

Figure 5: Final dynamic cloud mapped on sky plane from our demo project

6.6 Mapping the Cloud on the Sky Geometry

The shader gives the final cloud texture that we can use in any sky geometry of our choice, such as a plane or more conveniently a dome. For a plane it is just a huge quad and for the dome we normally use dome-shaped mesh for which texture coordinates are pre-computed or we can project the final texture output to the dome mesh using *texture projection*. We also scroll the texture coordinates in one direction to simulate the moving behavior of the clouds. We can see a snapshot from our demo in Figure 5 where the cloud image is mapped on a plane.

7. PERFORMANCE ANALYSIS & RESULTS

The performance we were able to achieve was very decent to be used with real-time graphics applications. We rendered the cloud created with 4 octaves and mapped it over a world with 20000 to 30000 polygons giving us 60 to 100+ frames per second. The demo provided by [3] showed instant drop of frame rate in regular basis, probably for multiple updating of octaves (using render to texture) in a given time frame. But in our case we tweaked the time range to avoid multiple octave update a single frame. This helped us to avoid instant drops in frame rate, thus keeping the frame rate consistent. We compared the test results with the demo provided by [3] in the same machine. Our demo was loaded with a 20,000 poly textured terrain. And the benchmarking result was very positive as indicated by Chart 1 in the following page. The average frame rate for our demo was about 86.5 FPS whereas the demo from [3] showed 60 FPS on average. The noticeable difference in highest and lowest frame rate of our demo suggests the existence of efficient optimization of pixel processing in the 3D Graphics Hardware pixel pipeline. Moreover, as we have rendered the program using OpenGL Vertex Arrays without any polygon optimization, there is a scope of improving the performance using Vertex Buffer Object (VBO) and spatial, hierarchical partitioning system (e.g. Octree or BSP) in our case. So we believe that the bench marking result was quite satisfactory.

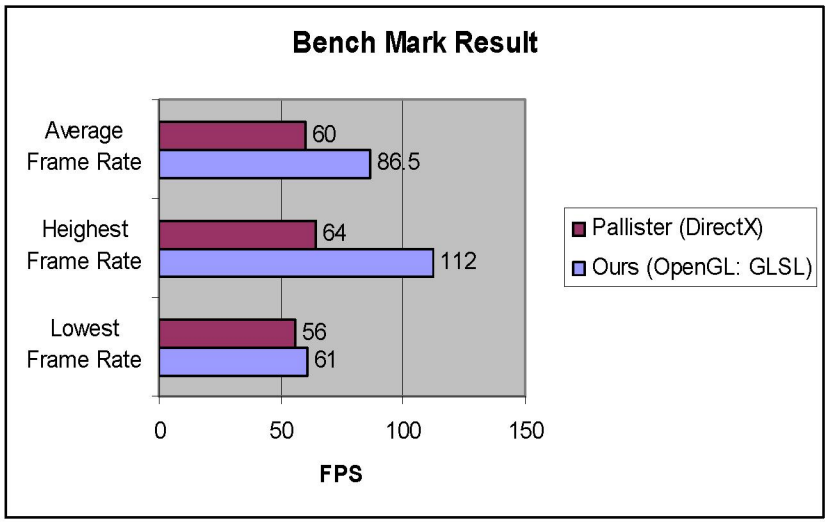

Chart 1: Bench mark result comparing our demo using OpenGL Shading Language and demo provided in [3] using DirectX.

8. LIMITATIONS

One limitation of implementation is that we are not able to produce various types of clouds depending on natural conditions and weather parameters. To accomplish that we will need to use more parameters to control the design of the clouds. The performance of glCopyTexSubImage*(…) function is slower on some GPUs when we copied a high resolution image (256X256 or larger).

9. CONCLUSION & FURTHER IMROVEMENTS

The scope for improvement is obviously unlimited as we strive to achieve totally *natural photorealism* in real-time. The look and construction of clouds can obviously be modified. However the most important addition to the features will be to add *atmospheric illumination* effects with the clouds to simulate the *color scattering* over the clouds in different light emission from the sun.

REFERENCES

[1] Ken Perlin, "Making Noise", http://www.noisemachine.com/talk1/, Based on a talk presented at GDCHardCore on Dec 9, 1999.

[2] John Kessenich, "Feature of The OpenGL Shading Language", http://developer.3dlabs.com/openGL2/index.htm, 03 May, 2005 White Paper.

[3] Kim Pallister, "Generating Procedural Clouds in Real Time on 3D Hardware", Game Programming Gems 2, 2001 Charles River Media.

[4] John C. Hart, "Perlin noise pixel shaders", Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pg 87-94, 2004.

[5] Chirtopher Oat, "Rendering to an off-screen buffer with WGL_ARB_pbuffer", ATI Technical papers, www.ati.com/developer/atipbuffer.pdf

[6] Christopher Wynn, "OpenGL Render-to-Texture", nVidia Technical Presentation, Developer Tools, developer.nvidia.com

[7] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, "OpenGL Programming Guide", Third Ed. Addison-Wesley 2000.

[8] Xavier Decoret, Fredo Durand, Francois X. Sillion, Julie Dorsey "Billboard Clouds for Extreme Model Simplification" ACM Transaction on Graphics, vol 22, issue 3, July 2003.

[9] Dean Macri, Kim Pallister "Procedural 3D Content Generation" Intel DevX, February 25, 2004

[10] Joshua Schpok, Joseph Simons, David S. Ebert, Charles Hansen "A real-time cloud modeling, rendering and animation system" Proceedings of the 2003 ACM SIGGRAPH/ Eurographics Symposium on Computer Animation.