Real-Time Fog using Post-processing in OpenGL

Anonymous*
A Research

ABSTRACT

Fog is often used to add realism to a computer generated scene, but support for fog in current graphics APIs such as OpenGL is limited. The standard fog models are very simplistic, allowing for a uniform density fog with slight variations in attenuation functions. While recent extensions to the OpenGL standard provide height dependent, or layered fog, it does not correctly account for line-of-sight effects as the viewpoint moves through the fog.

In this paper, we present a new, fast but simple method for generating heterogeneous fog as a post processing step. Using standard OpenGL Shading Language features, it requires only the depth buffer from the normal rendering process as input, evaluating fog integrals along line-of-sight to produce realistic heterogeneous fog effects in real-time.

Keywords: Algorithm, Post-processing, Real-Time fog, Shader.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors

1 INTRODUCTION

As far back as version 1.1, OpenGL has provided support for fog. However, fog has always been implemented as a relatively cheap, and simplistic, depth based effect. The basic idea behind fog is that the distance between the viewpoint and a fragment is computed. This distance is then used to blend the fragment color with a fog color. While this produces reasonable results, there are a few problems. The OpenGL specification permits implementations to approximate the fragment distance using only the fragment depth, or Z value. Most API implementations use this approximation, which leads to some undesirable effects as shown in Figure 1, where more trees are visible along the horizon in the right image. Both images are generated from the same camera location. However, in the right image, the camera has been tilted downwards. While the camera-object distance has not changed for these trees, the rotation of the camera has resulted in reduced Z values. This causes a reduction in the computed fog density, allowing trees that were previously invisible, to become visible.

Figure 2 shows how this artifact can occur using OpenGL fog model. The grey band shows the visible region from no fog to complete fog. In the left diagram, Tree 1 is in the fully fogged region. With the camera rotated in the right diagram, Tree 1 falls into the partially fogged region, and thus become visible.

Recent additions to the OpenGL standard include the ability to specify the effective $'Z'$ value, or depth of a vertex for the purposes of fog computations. Often referred to as the fog extension, this permits significantly more control over the fog generation and allows effects such as height dependent fog, but at the expense of providing more per-vertex data - already a potential bottleneck, and more CPU cycles.

*e-mail: a@aol.com

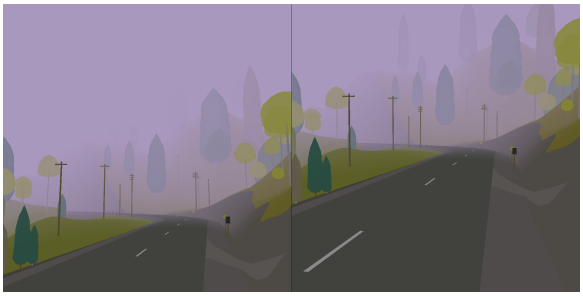

Figure 1: Problems with standard fog. The right image shows more trees on the hillside when the camera is tilted downwards.

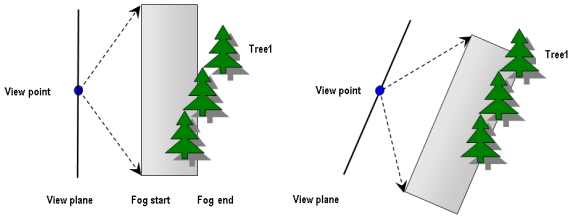

Figure 2: Fog artifacts

Another problem with the fog model in OpenGL is that only rendered objects are fogged, leaving the background unaffected, as shown by Figure 3.

The obvious solution to this problem is to set the background color to the fog color, but this only works for homogeneous fog, not the fog extension, which usually requires a graded background. Thus, the standard fog model employed by OpenGL is somewhat limited in its application.

Considerable work has been done to improve the situation using either physically-based or appearance-based approaches [1] [2] [4] [5] [6] [8] [9] [10] [11] [12] [16] [18] [19] [21] [22] [23] [25], with the ultimate goal of simulating fully heterogeneous fog in three dimensions. However, most approaches either cannot achieve the performance characteristics required for real time applications, or require sophisticated algorithms that must be integrated into the main rendering pipeline.

In this paper we present a new method of generating real-time heterogeneous fog using post-processing technique. Our algorithm uses analytical functions to evaluate the fog integral along the line of sight, accurately computing the fog density on a per-pixel basis to produce heterogeneous fog effects. It requires only the depth buffer generated during the normal rendering process, making it very easy to integrate into existing rendering applications. Additionally, it is also extremely fast, being implemented entirely on the GPU.

In the next section, we review the existing approaches for real time fog rendering. Section 3 describes the technical details including fog approximation in appearance based fog simulation and our new algorithm with hardware acceleration technique. Section 4 presents the implementation as well as the integration of our al-

Figure 3: Background is not fogged

gorithm to the OpenGL rendering pipeline. Section 5 shows the results, section 6 compares our method with others and section 7 discusses our conclusions and ideas for future work.

2 Previous Work

Existing work for realistic fog rendering can be broadly categorized as physically-based and appearance-based.

In the physically-based approach, fog is considered as a standard participating medium, and is generated through global illumination algorithms. To reduce expensive computational cost, simplifications are made through single scattering and multiple scattering models. In single scattering, the participating medium is assumed to be optically thin, allowing the source radiance simplification to ignore multiple scattering within the medium. Analytic method [23] has been used to solve the integral problem presented in the participating medium with significant simplification. The deterministic method [11] [16] aims at numerical solutions of the same problem, for example, Zhu et al [25] applies 'depth peeling'[7] technique to calculate volumetric lighting, while the stochastic method [22] is found to be used in clouds instead of fog by applying random sampling to solve the equation for the radiance reaching the eye for the single scattering. In multiple scattering, both deterministic and stochastic methods used here contain two stages: an illumination pass for computing the source radiance, and a visualization pass for the final image. Some of the deterministic methods [21] extend the classical radiosity method that accounts for isotropic emitting and scattering medium (zonal method), other improvements can deal with anisotropy using spherical harmonics [2] [10], discrete ordinates, or implicit representation. Stochastic methods [6] [9] [12] [18] solve the transport equation by means of random sampling, using random paths along iteration points. In summary, while these methods can produce realistic scenes, they come with a high computational cost, even when hardware acceleration is implemented, they are not real-time. Cerezo et al [5] provides a comprehensive review of the participating medium rendering techniques.

In appearance based approaches, the idea is to produce visually pleasant results without expensive global illumination determination. Perlin [19] documented snapshots of thoughts such as using Gabor functions for simulating atmospheric effects. Biri et al [3] [13] suggested modeling of complex fog through a set of functions allowing analytical integration. Zdrojewska [26] introduced randomness in attenuation through perlin noise and produces heterogeneous looking fog without line of sight integral. Height dependent, or layered fog, varies the fog density based on the height [8] [14]([19] dose not contain implementation details). Mech [15] proposed to represent the gas boundary by a polygonal surface, using graphics hardware to determine the attenuation factor for each pixel inside or behind the gas. Nvidia developers[17] adopted similar idea proposed in [15], developed a RGB-encoding technique combining with 'depth peeling'[7] to render fog polygon objects as thick volumes on GPU. The advantage of these approaches is that they allow simple but fast rendering of fog. However, Biri's method is not yet fast enough on current hardware, and Zdrojewska's method does not take fog volume into account, causing incorrect results when considering motion parallax, making it more suited to simulate low cloud in natural environments without traveling through the fog (i.e. as opposed to man-made environments). Although Biri's algorithm used analytical functions to approximate the extinction coefficient, the performance of the implementation was constrained by the way in which the depth information was used for integration, and the pixel texture technique used in [8] did not produce real-time results. Fog polygon volume based approach [15] [17] suite more for small volumes such as light shaft, smoke, etc. An additional constraint imposed by many of these techniques is that the production of the fog effect must be integrated into the main rendering pipeline - something that is not practical for many existing rendering applications. Our new appearance-based approach avoids this constraint by using the depth buffer to reconstruct scene depth information, while performing line-of-sight integration on the GPU using analytical functions and true Euclidean distance.

3 Technical Details

To compute fog color, we need to integrate along the ray between the camera position and the fragment position given the fog density function. There are two main aspects to generating fog using a post processing approach. The first is to reconstruct the 3D position of each fragment in the frame buffer, while the second is to compute the fog based on the fragment and eye positions.

3.1 3D Fragment Position Reconstruction

In order to evaluate the fog integral along line of sight accurately, the 3D position of each fragment in world coordinate needs to be reconstructed. The depth buffer stores transformed depth value for each fragment and can be used to reconstruct 3D fragment position. Generally, most post processing applications that need depth information use specialized shaders and alternate render targets to 'lay down' depth information in a form which can readily be used in the post processing stages. The disadvantage to this approach is that it requires significantly more integration into the rendering application. It can also impact performance (requiring an extra scene rendering pass) and bandwidth (requiring an extra render target). For our new approach, we use a standard post-processing technique of drawing a screen aligned polygon with customized vertex/fragment shaders, but use the regular depth buffer generated as part of the normal rendering process as an input texture instead. The problem with the depth buffer is that the depth information is non-linear, having been transformed by the modelview and projection matrices. The first step is to reconstruct scene depth. A naive approach to this is to use an inverted modelview/projection matrix, along with normalized x and y values, to compute the full 3D fragment position relative to the camera. Once this is known, along with the position and orientation of the camera, the real world position of the fragment can be obtained. However, this turns out to be quite expensive, so we use an alternative, faster approach. The first step is to compute the depth value z. From an examination of the projection matrix, we can deduce that

$$z = \frac{-P_{34}}{\frac{z'}{w'} + P_{33}} \tag{1}$$

where

$$P_{33} = \frac{f+n}{n-f} \tag{2}$$

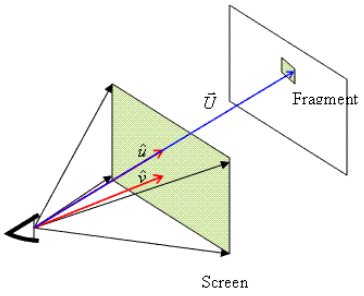

Figure 4: Reconstruct 3D fragment position, relative to the camera.

and

$$P_{34} = \frac{-2f \times n}{f - n} \tag{3}$$

where n is the distance to the near clipping plane, and f is the distance to the far clipping plane.

Given that we can obtain $\frac{z'}{w'}$ from the depth buffer, and both P_{33} and P_{34} can either be computed based on knowledge of the camera properties, or obtained directly from the projection matrix, we can easily convert a value from the depth buffer to an actual depth value. This also has the result of reducing it to a scalar operation, requiring one addition and one division per pixel.

The second step is to compute the real world position of the fragment based on this depth value, relative to the camera. To do this, we take advantage of the graphics hardware by setting up a unit vector in the direction of the fragment, using a vertex shader in the post processing. The hardware automatically interpolates this for us to use in the fragment shader, where it is used to generate the relative location of the fragment by computing the fragment Euclidean distance $|\vec{U}|$.

$$|\vec{U}| = \frac{z}{\hat{u} \cdot \hat{v}} \tag{4}$$

where $\hat{u}$ is the unit vector in the direction of the camera to the fragment, $\hat{v}$ is the unit vector in the direction of the camera, and z is defined in (1). Figure 4 shows 3D fragment position reconstruction. The 3D position of the fragment relative to the camera is then obtained by

$$\vec{U} = |\vec{U}| \cdot \hat{u}$$

The final step is to add the position of the camera to $\vec{U}$, producing the real world 3D position of the fragment $\vec{P}$, where $\vec{P} = (x_{fragment}, y_{fragment}, z_{fragment})$.

3.2 Fog Computation

3.2.1 Homogeneous Fog

Appearance-based fog generally assumes the scattering of a constant ambient illumination in a homogeneous non-emitting medium. It simplifies the transport equation in participating medium model as follows:

$$L(x) = e^{\kappa_a |x_0 - x|} L(x_0) + (1 - e^{\kappa_a |x_0 - x|}) L_e \tag{5}$$

where L_e represents the constant amount of light, $L(x_0)$ is the radiance of light at space point x_0, and κ_a being a constant as the absorption coefficient to represent homogeneous medium.

The first term characterizes the loss of light of the object surface due to the absorption of the medium. The second term describes the contribution of the scattering within the medium. The OpenGL fog model uses this approximation to integrate a uniform attenuation due to medium absorption between the object and the viewer.

$$C = f \cdot C_{in} + (1 - f) \cdot C_{fog}$$

where $f = e^{-(density \cdot z)}$, z is the eye-coordinate distance between the viewpoint and the fragment, C_{in} is the fragment color, C_{fog} is the fog color. C_{in} maps to $L(x_0)$, C_{fog} maps to L_e.

3.2.2 Layered Fog

Layered fog extends equation (5) to non-uniform attenuation of the medium - $f(u)$, based on height variation.

$$L(x) = e^{-\int_{camera}^{fragment} f(u)du} L(x_0) + (1 - e^{-\int_{camera}^{fragment} f(u)du}) L_e$$

Once we have the fragment position in real world coordinates, we can compute the expected blending of the fragment color with the fog color, based on the amount of fog between the camera and the fragment. To do this, we need to evaluate the fog density integral along the camera-to-fragment vector. In the case of a homogeneous fog, this computation is trivial, while in the case of fully heterogeneous fog, it can be computationally very expensive. Layered fog, where the fog density f, is dependent only on the height y, is a special case of heterogeneous fog. Suppose the total fog between the camera, and the fragment position, is given by the integral F along the camera-fragment vector.

$$F = \int_{camera}^{fragment} f(u)du$$

where $f(u)$ is the fog density at a 3d space position. Since the fog density is only dependent upon y, this can be simplified:

$$F = \frac{1}{\sin(\theta_{fragment})} \int_{y_{camera}}^{y_{fragment}} f(y)dy \tag{6}$$

where $\theta_{fragment}$ is the angle of inclination of the camera to fragment vector, calculated on per pixel basis, y_{camera} is the y coordinate of the eye point in world space, and $y_{fragment}$ is the fragment y position in world coordinate. Figure 5 shows the geometric relationships among Euclidean distance $|\vec{U}|$, θ and the height $|y_{fragment} - y_{camera}|$. We can easily apply $|\vec{U}|$ to express equation (6) as follows:

$$F = \frac{|\vec{U}|}{|y_{fragment} - y_{camera}|} \int_{y_{camera}}^{y_{fragment}} f(y)dy \tag{7}$$

Obviously, equation (7) coverts absorption of a ray along line of

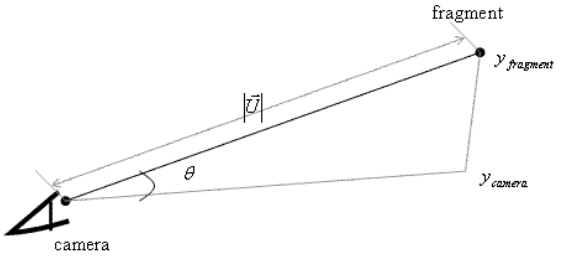

Figure 5: Height Dependent Fog Computation Per Fragment

sight to its vertical component by a scaling factor. Combining equation(4), we define layered fog as below:

$$F = \frac{z}{\hat{u} \cdot \hat{v}} \cdot \frac{1}{|y_{fragment} - y_{camera}|} \int_{y_{camera}}^{y_{fragment}} f(y)dy \tag{8}$$

The method of evaluating this integral depends on the fog function. Functions that can be integrated analytically are generally trivial to compute in a fragment shader, but it can be difficult to achieve realistic looking since most analytical functions produce periodic patterns unless a large number of terms are used, and this can be slow to evaluate. Hard coded functions can be used to produce customized fog patterns, well suited to layered fog. Non-analytical functions present the biggest problem, since approximation methods require looping that seriously degrades performance. Our solution to this was to pre-compute the fog integral across its expected range, storing the result in a texture that could be used as a lookup table in the fragment shader.

3.2.3 Heterogeneous Fog

By using a fog density function that is independent in each dimension, we can easily extend our method to produce 3 dimensional heterogeneous fog effects. In this case, the fog integral F can be modeled as follows:

$$F(x,y,z) = F(x)F(y)F(z)$$

As with layered fog, functions that can be integrated analytically can be evaluated directly in the fragment shader, or the integrals can be pre-computed and stored in textures.

3.2.4 Putting it Together

Once the fog integral has been evaluated, the fragment can be blended with the fog color. We choose an exponential blending function

$$f = e^{-F}$$

although any suitable function can be used.

3.2.5 Color Gradient Fog

We can extend this algorithm to evaluate fog integrals on individual color components using different fog density functions. This can be used to produce color variations in fog, for example, simulating smog when certain colors are scattered or absorbed more than others. Using layered fog as an example, the fog computation is modeled by the following:

$$F_{r||g||b} = \frac{z}{\hat{u}\cdot\hat{v}} \cdot \frac{1}{|y_{fragment} - y_{camera}|} \int_{y_{camera}}^{y_{fragment}} f_{r||g||b}(y)dy \qquad (9)$$

where $f_{r||g||b}(y)$ is fog density function for r, g or b color component. The fragment is then blended using the individually computed component values as follows;

$$(S_r + 1 - D_r, S_g + 1 - D_g, S_b + 1 - D_b)$$

where (S_r, S_g, S_b, S_a) and (D_r, D_g, D_b, D_a) are the source and destination blending factors respectively.

4 Implementation

Our implementation is based upon a custom hierarchical scene management and OpenGL based rendering engine developed by the authors, although any OpenGL based rendering engine should be suitable. Written in C++, our application provides 'hooks' into various parts of the rendering pipeline, allowing us to easily insert our post processing algorithm.

The first step is to render the scene into the frame buffer, which is done by the application as before. Once the scene rendering is complete, the scene post processing is triggered just prior to the frame buffer swap. At this point, the depth buffer is captured using the OpenGL copy-to-texture functionality. Next, the depth conversion parameters P_{33}(equation(2)) and P_{34}(equation(3)) are computed based on the camera properties. The post processing is then initiated by drawing a screen-aligned polygon, textured with the depth texture, and using custom vertex and fragment shaders written in the OpenGL Shading Language. The vertex shader is used to set up the eye-to-fragment unit vector, defined in real world coordinates. This is interpolated by the hardware pipeline, and delivered to the fragment shader for use in computing the fragment position. The depth conversion parameters needed for converting the depth buffer values are passed to the fragment shader as uniform variables. Figure 6 shows the steps in our post processing implementation, and how it integrates into the rendering application.

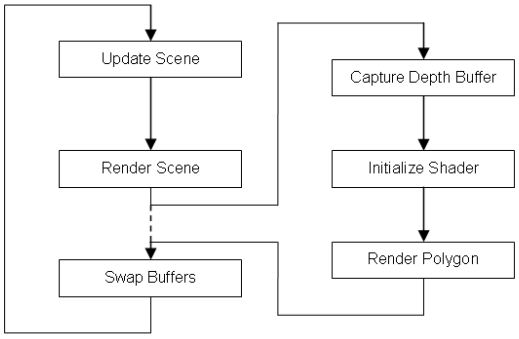

Figure 6: Integration of Post-processing

To blend the fog color with the fragment color, we take advantage of the standard OpenGL blending functionality and simply compute an alpha value in the fragment shader. The output from the fragment shader is a fragment color consisting of the RGB fog color, and the computed alpha (fog) value. The rendering pipeline takes care of the rest of the work by blending the fog color with the existing fragment in the frame-buffer.

The fragment shader itself is relatively simple. Pseudo code for generating layered fog is given for it below.

```
    read fragment depth from depth texture
    perform scene depth reconstruction
    compute eye-to-fragment vector
    compute fragment position in real world
    compute Euclidean distance between the
camera and the fragment
    evaluate fog integral
    compute alpha value
```

5 Results

The algorithms in this paper were implemented on a 2.8GHz Pentium IV platform, with 1GB RAM and an nVIDIA 6800 graphics card, running Linux.

A number of test scenes were chosen using a variety of resolutions and fog functions.

The first test scene Figure 7, uses a simple four point ramp function to produce a simple height dependent (layered) fog. The idea behind this fog function was to produce a thin layer of fog, just above the ground.

Figure 8 is an underwater scene that demonstrates another use of fog. This scene uses a fog density that increases linearly with depth to simulate the increasing attenuation of light in water.

Figure 9, the cave scene, uses a two point fog function, with a uniform density up to a height y_1, then falling linearly to zero at a point y_2.

Figure 7: Serene Street - A low level ground fog using a simple height based function

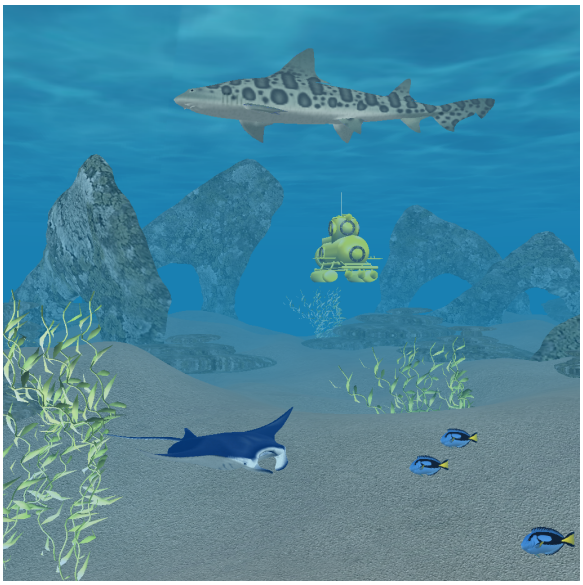

Figure 8: Underwater Adventure

Figure 10, another cave scene, with heterogeneous fog based on Perlin noise functions (pre-integrated) for X and Z, and an exponential function for the height.

Figure 11, the Village scene, uses heterogeneous fog based on 3 Perlin noise functions.

Figure 12 shows a selection of scenes that use a simple fog density function that increases linearly with decreasing height. The heterogeneous appearance is a result of depressions in the terrain, and significantly improves the perception of depth.

Tests indicated that the cost of generating layered fog effect ranges between 0.39 to 3.08 milliseconds for the above scenes, depending on the screen resolution and complexity of the fog function. The maximum screen resolution was 1024×1024.

6 Discussion

Since our technique is post-processing, and evaluates the fog integral along the line of sight on per pixel basis, it does not suffer from the popping effect discussed in figure 2, nor the background problem shown in figure 3. It eliminates limitations in methods such as Biri's [3], which produces noticeable artifacts for fog rendering when the texture size is small compared to the image size, or compromised performance if a larger texture is used. To enhance the heterogeneous appearance of the fog rendering, we apply Perlin noise functions instead of equipotential functions used for local refinement by Biri et al. Our method can also easily simulate the effects of wind to animate the fog by providing time dependent offsets when evaluating the fog integrals.

7 Conclusion and Future Work

In this paper we demonstrate a new, flexible and efficient method of generating advanced fog effects in real time. We apply post-processing techniques in the GPU to produce natural looking heterogeneous fog using Euclidean fragment distance reconstructed from the depth buffer. Since the implementation requires only the normal depth buffer as input, we avoid the need for customized shaders in the standard rendering pipeline, making it very easy to integrate into existing rendering applications. In summary, our contribution includes the following attributes:

- Accurate scene depth reconstruction from the depth buffer.
- Accurate 3D fragment position reconstruction in world space from scene depth.
- An easily integrated GPU based post-processing technique for real time frame rates.
- Realistic Heterogeneous fog and extension of color gradient fog.

Our method makes the assumption of independent fog density on three dimensions. In future work, we need to solve arbitrary fog density integral problem including correlations across dimensions, and extend this technique into a global illumination model.

References

[1] N. Adabala and S. Manohar. Modeling and rendering of gaseous phenomena using particle maps. *Journal of Visualization and Computer Animation*, 11(5):279–294, 2000.

[2] N. Bhate and A. Tokuta. Volume rendering of media with directional scattering. In *Third Eurographics Workshop on Rendering*, pages 227–245, May 1992.

[3] V. Biri, S. Michelin, and D. Arques. Real-time animation of realistic fog. In *Rendering Techniques 2002 (Proceedings of the Thirteenth Eurographics Workshop on Redering)*, June 2002.

[4] P. Blasi, B. L. Saec, and C. Schlick. An importance driven monte-carlo solution to the global illumination problem. In *Fifth Eurographics Workshop Render*, pages 173–183, June 1994.

[5] E. Cerezo, F. Perez, X. Pueyo, F. Seron, and F. X. Sillionn. A survey on participating media rendering technique. *The Visual Computer*, 21(5):303–328, 2005.

[6] E. Dumont. Semi-monte carlo light tracing applied to the study of road visibility in fog. In *Proceedings of the Third International Conference on Monte Carlo and Quasi Monte Carlo Methods in Scientific Computing*, 1998.

[7] C. Everitt. Interactive order-independent transparency. http://developer.nvidia.com/attach/6545, 2001.

[8] W. Heidich, R. Westermann, H. Seidel, and T. Ertl. Applications of pixel textures in visualication and realistic image synthesis. In *Proc. ACM Sym. On Interactive 3D Grahpics*, pages 127–134, April 1999.

[9] H. W. Jensen and P. H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Computer Graphics (ACM SIGGRAPH 98 Proceedings)*, pages 311–320, 1998.

Figure 9: Mystic Cave - Layered Fog using a simple two point ramp function

Figure 11: Barnyard Breeze - Heterogeneous fog using Perlin noise functions in 3 dimensions

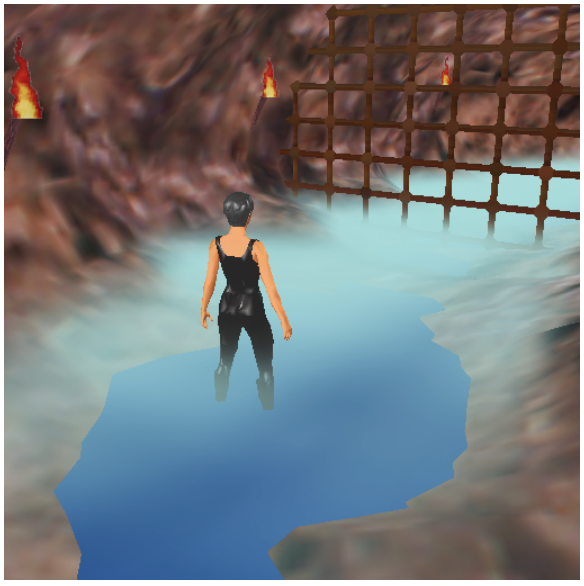

Figure 10: Mystic Cave - Heterogeneous Fog based on Perlin noise functions in 2 dimensions, and a height based exponential function

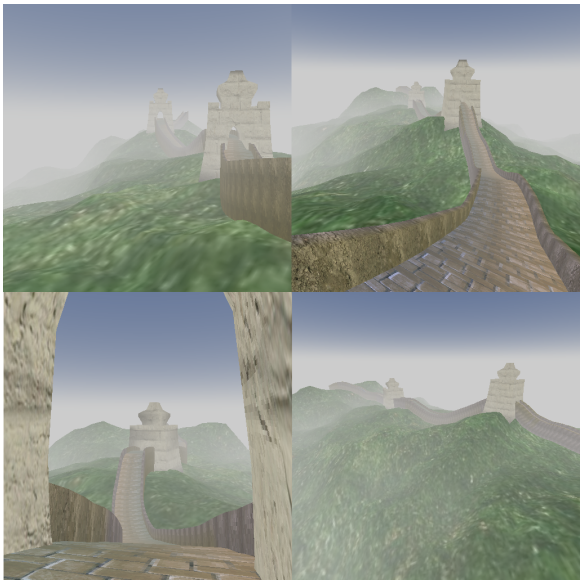

Figure 12: Great Wall Morning Mist - Layered fog exaggerates the terrain features

[10] J. T. Kajiya and B. P. V. Herzen. Ray tracing volume densities. In *Computer Graphics (ACM SIGGRAPH 84 Proceedings)*, volume 18, pages 165–174, July 1984.

[11] R. Klassen. Modeling the effect of the atmosphere on light. *ACM Transactions on Graphics*, 6(3):215–237, 1987.

[12] E. P. Lafortune and Y. D. Willems. Rendering participating media with bidirectional path tracing. In *Rendering Techniques96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 91–100. Springer-Verlag/Wien, 1996.

[13] P. Lecocq, S. Michelin, D. Arques, and A. Kemeny. Mathematical approximation for real-time rendering of participating media. In *Proceeding of Eighth Pacific Conference on Computer Graphics and Applications*, page 400, 2000.

[14] J. Legakis. Fast muilti-layer fog. In *Siggraph'98 Conference Abstracts and Applications*, page 266, 1998.

[15] R. Mech. Hardware-accelerated real-time rendering of gaseous phenomena. *Journal of Graphics Tools*, 6(3):1–16, 2001.

[16] T. Nishita, Y. Miyawaki, and E. Nakamae. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In *Computer Graphics (ACM SIGGRAPH'87 Proceedings)*, volume 21, pages 303–310, 1987.

[17] Nvidia. Fog polygon volumes-rendering objects as thick volumes. http://download.developer.nvidia.com/developer/SDK/individual_Samples/DEMOS/Direct3D9/src/ FogPolygonVolumes3/docs/FogPolygonVolumes3.pdf, 2004.

[18] S. N. Pattanaik and S. P. Mudur. Computation of global illumination in a participating medium by monte carlo simulation. *The Journal of Visualization and Computer Animation*, 4(3):133–152, July-September 1993.

[19] K. Perlin. Using gabor functions to make atmosphere in computer graphics. http://mrl.nyu.edu/ perlin/experiments/garbor/, Year unknown.

[20] R. J. Rost. *OpenGL Shading Language, 2nd Edition*. Addison Wesley, 2006.

[21] H. E. Rushmeier and K. E. Torrance. The zonal method for calculating light intensities in the presence of a participating medium. In *Computer Graphics(ACM SIGGRAPH'87 Proceedings)*, pages 293–302, July 1987.

[22] G. Sakas. dfast rendering of arbitrary distributed volume densities. In *Eurographics'90*, pages 519–530, July 1990.

[23] P. J. Willis. Visual simulation of atmospheric haze. *Computer Graphics Forum*, 6(1):35–42, 1987.

[24] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide, 5th Edition*. Addison Wesley, 2005.

[25] F. L. Y. Zhu, G.S. Owen and A. Aquillio. Gpu-based volumetric lighting simulation. In *Computer Graphics and Imaging*, pages 160–162, August 2004.

[26] D. Zdrojewska. Real time rendering of heterogenous fog based on the graphics hardware acceleration. http://www.cescg.org/CESCG-2004/web/Zdrojewska-Dorota/, 2004.