Real time rendering of heterogenous fog based on the graphics hardware acceleration

Dorota Zdrojewska *

Institute of Computer Graphics and Multimedia Systems
Technical University of Szczecin
Szczecin / Poland

Abstract

This paper discusses the subject of synthesis of the fog phenomenon in real time using computer graphics. The method of heterogenous fog simulation based on the Perlin noise and turbulence is presented. Implementation of the algorithm is done using hardware acceleration in the form of a GPU with programmable vertex and pixel processing pipeline.

Keywords: Fog synthesis, Perlin noise, vertex shader, pixel shader, computer graphics

1 Introduction

Visualization of atmospheric phenomena is an important use of computer graphics, and of these fog is perhaps most frequently imaged. Its presence in games, simulators and virtual reality environments significantly enhances realism and adds to attractiveness of generated scenes. The simulation in real time is usually limited to depicting a homogenous fog [2][12], with constant density distribution, often as a volumetric layered effect [1] [7] [4]. In the natural world the phenomenon is however more complex – it is a participating medium made of many non-homogenous layers moving with the wind and undergoing turbulent motion.

Existing methods of modelling heterogenous participating media can be classified in two groups. First of them use physical models of turbulent motion, based on FFT [6] or Navier-Stokes equations [3] [16]. These methods produce very realistic images, but require long time of computation. The second group approximates physical properties of participating medium, enabling its visualization in real time. Simulation consists on applying for example periodical or fractal functions [17] [9].

To speed up the simulation graphics hardware could be used. Modern GPUs (*Graphics Processor Unit*) offer more and more effective ways of accelerating graphics-related calculations. One of recent innovations is an architecture with programmable 3D pipeline that allows supplementation of standard graphics pipeline with vertex and pixel processing routines – so-called vertex and pixel shaders, which run in hardware. With their use it becomes possible to execute much more sophisticated algorithms in real time, including algorithms simulating phenomenon of heterogenous fog.

This paper discusses the problem of fog simulation and visualization, with special emphasis on real time rendering. Section 2 introduces the basic algorithms of homogenous fog simulating, including linear and exponential formulas. A method of modelling and animating a heterogenous fog based on Perlin turbulence is also described. Section 3 presents the general fog synthesis algorithm of which implementation, employing vertex and pixel shaders, is presented in section 4. The results are demonstrated and discussed in section 5. Conclusions and possible subjects of future work are contained in the last section 6.

2 Theoretical background

In the real world fog is a participating medium, consisting of small water droplets not larger than 0.05 in diameter. According to [17], radiance of light entering such medium is influenced by four factors:

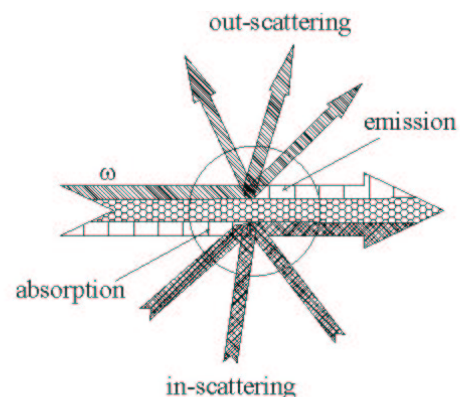

Figure 1: Absorption, emission, out-scattering and in-scattering in participating medium

*dzdrojewska@wi.ps.pl

- absorption
- emission
- out-scattering (diffusion)
- in-scattering

Absorption and scattering reduce radiance – part of light travelling along a ray ω is absorbed and refracted by fog particles. In-scattering is an opposite process – it occurs when light, after being refracted, starts moving in direction ω. In this model, particles of fog can also emit light. Figure 1 illustrates these processes.

The influence of these factors on radiance of ray travelling from point in the scene towards the observer is expressed by the transport equation, of which derivation was shown in e.g. [17] [14] [15]. Its simplified form is frequently used in computer graphics as so-called mixing equation:

$$C_{Final} = (f * C_{Current} + ((1 - f) * C_{Fog}) \quad (1)$$

The first element of equation (1) represents the loss of light, while the other its in-scattering and emission. These processes are mirrored by mixing pixels' colors $C_{Current}$ with fog color C_{Fog} based on its intensity f. The goal is to determine the final pixel color C_{Final}.

The final pixel's color depends on the value of coefficient $f \in< 0, 1 >$, which defines the intensity of fog at the point of space represented by the pixel. The smaller the value of f, the higher is the influence of fog on the color of the point in question, and hence – the higher drop in visibility. $f = 0$ means total "fogging" of the point, thus its final color is the color of fog. $f = 1$ means no "fogging", color of the pixel won't be changed at all.

Various effects can be obtained depending on the method used to calculate coefficient f – both homogenous fog of constant density and more complex, heterogenous phenomenon as well.

2.1 Synthesis of homogenous fog

To generate homogenous fog, coefficient f is usually computed using one of three basic methods: linear, exponential and squared exponential [5]. In linear method, f is computed using linear interpolation, depending on the distance of point from the observer and on its location in relation to the fogged area:

$$f = \frac{Fog_{End} - d}{Fog_{End} - Fog_{Start}} \quad (2)$$

where:

- Fog_{Start} – beginning of the area influenced by fog,
- Fog_{End} – the distance designating the border of visibility, beyond which nothing can be seen
- d – distance between considered point and the observer.

In exponential method and its squared variety, coefficient f depends on distance between observed point and the observer, and on the density of the fog.

$$f = e^{-(d*g)} \quad (3)$$

$$f = e^{-(d*g)^2} \quad (4)$$

where:

- d – distance of considered point from the observer,
- g – fog density coefficient.

The methods described above, thanks to their simplicity and resulting small computational cost, are used in real time graphics systems. They were implemented in Direct3D [2] and OpenGL [12] libraries. However, the effect of homogenous fog obtained with these methods does not in fact fully reproduce realistic appearance of this phenomenon, as the variable density of fog is not accounted for.

2.2 Generating heterogenous fog with Perlin noise

The natural phenomena usually do not change in regular ways, but are characterized by large degree of randomness. Such feature is also present in fog, which is a volume object of variable density, taking irregular shapes due to wind and air turbulence. Because of that, random noise function seems like a good candidate to help simulate it. An argument to the function are two- or three-dimensional coordinates of a point in space, and the result is a pseudo-random value of fog's density at these coordinates.

Noise generated by such function has a high frequency and hence displays rapid changes between contrasting values, which is not typical for fog density distribution. Therefore, it should be rather modelled with smooth noise created by interpolation of random value samples sequence. The method of generating such noise was proposed by Ken Perlin in [8].

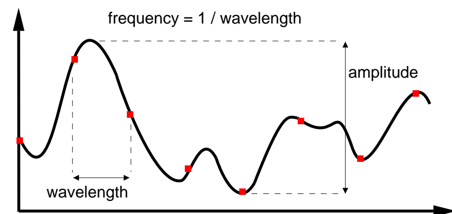

Figure 2: Amplitude and frequency of the noise function

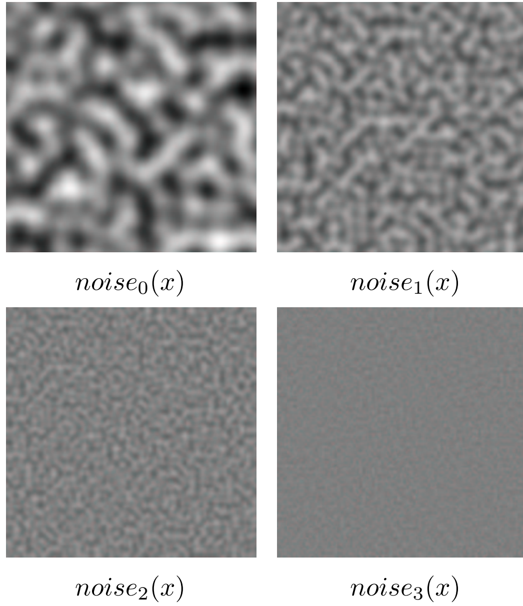

Figure 3: Textures of Perlin noise octaves [13]

The noise function is characterized by two properties: amplitude and frequency. As seen in the figure 2, amplitude is the difference between maximum and minimum noise value. Frequency is reciprocal of the distance between the noise samples.

The main idea behind simulating the fog phenomenon is to add several noise functions with various frequencies and amplitudes together, according to the Perlin's turbulence equation 5. In result, distribution of heterogenous fog's density is obtained.

$$turbulence(x) = \sum_{i=0}^{N-1} \frac{noise_i(x)}{2^i} \tag{5}$$

In the formula 5 each $noise_i(x)$ function is representing the component fog's density at the space point with x coordinates (2, 3 or more dimensional). All these functions are known as octaves. The reason for this is that each of them has double frequency of the previous one.

The number N of octaves can vary, but for rendering images N does not have to be large. It comes from the fact that if an octave has too high frequency, it cannot be displayed with respect to screen and the grey scale resolutions.

3 Algorithm

Modelling the heterogenous fog phenomenon takes place in three basic steps:

- generating the noise octaves textures
- computing the fog factor
- blending colors of the scene with the fog color

The first step of the algorithm is creating four noise textures. Their colors are in the grey scale and represent component values of the heterogenous fog's density. Two fundamental parameters have influence on final appearance of turbulent texture of fog: frequency and amplitude of the noise.

Frequency specifies the rate at which colors change in generated texture. Low frequency noise is changing gradually, and shifts of contrasting colors are following slowly. Increase of frequency is causing more rapid changes, and in result rising amount of perturbations in the image.

Amplitude is a parameter describing the range of texture colors. The larger amplitude the larger rate of color change and more varying image of fog. Decrease of amplitude restricts the range of colors, and in effect fog's density distribution is becoming smoother.

Figure 3 shows four example Perlin noise textures used to simulate turbulent fog phenomenon.

The first texture, with the lowest frequency, is describing general shape of fog. Following textures are representing the noise with double frequencies, in relation to preceding ones. They are increasing level of detail in the image by putting in it the turbulence effect.

After creating textures their colors are mixed together according to the Perlin's turbulence formula (5). In result heterogenous fog's density distribution texture is generated, as shown in the figure 4.

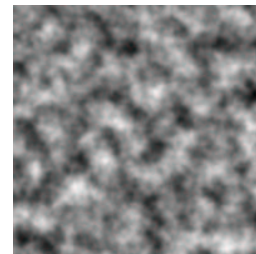

Figure 4: Perlin's turbulence texture [13]

Having the final fog texture generated, the fog factor can be calculated. It is done by involving the heterogenous density distribution $turbulence(x)$ in the exponential fog equation (3) or (4):

$$f(x) = e^{-(d*g*turbulence(x))} \tag{6}$$

$$f(x) = e^{-(d*g*turbulence(x))^2} \tag{7}$$

Using the exponential formula is aimed at enhancing the realism of the simulated effect. Since a distance coefficient is considered, the visibility is lower at the point being far away from the camera, just like in real world.

The last step of the algorithm is to blend scene colors with the fog color, depending on the fog's factor f, with use of the mixing formula (1).

4 Implementation

Realization of the heterogenous fog algorithm is based on hardware acceleration in form of vertex and pixel shaders ([11],[10]). The algorithm is implemented in Direct3D 9.0 with NVIDIA's Cg language.

The noise octaves are represented by the volumetric textures, defined in three dimensions. Various noise frequencies are obtained by different resolutions of textures. The largest frequency is represented by the largest resolution texture and vice versa. Textures are calculated in preprocessing with use of the rand() function, and then passed in the pixel shader's constant registers. Smoothing the noise is realized by setting the linear filtering of the texture. Before generating textures, the amplitude is set. The maximum range of colors (the grey scale) is between 0 and 255.

Coordinates of noise textures are calculated for each vertex of the scene in the vertex shader program. Its main task is translation of these coordinates, what in result causes movement of the textures in various directions with different speeds. In this way animation of turbulent fog is realized. Calculating of the fog's factor is done for each pixel of the screen in the pixel shader program. It is also mixing base scene colors with the fog color.

4.1 Vertex Shader program

Vertex shader program is making the same computations for each vertex of the scene separately. Vertex's positions and base texture coordinates are passed from the vertex stream in input data registers (POSITION, TEXCOORD0) represented by the following structure:

```
struct VertexInput {
    float4 Position        : POSITION;
    float2 BaseTexCoords   : TEXCOORD0;
};
```

The results of computations are passed in the vertex shader's output registers, described by the VertexOutput structure. It includes vertex's position in the world-view-projection space, base texture coordinates, four noise texture coordinates and the distance between considered vertex and the camera.

```
struct VertexOutput {
    float4 Position        : POSITION;
    float2 BaseTexCoords   : TEXCOORD0;
    float3 NoiseTexCoords0 : TEXCOORD1;
    float3 NoiseTexCoords1 : TEXCOORD2;
    float3 NoiseTexCoords3 : TEXCOORD3;
    float3 NoiseTexCoords4 : TEXCOORD4;
    float4 Distance        : TEXCOORD5;
};
```

Constant registers are set with the data necessary for calculations. Important to fog simulation is the camera position (CameraPos) and settings of the noise textures' animation (Animation). The camera position is used to calculate the distance between the vertex and observer. It is necessary for computation of the fog's factor. The distance is passed in the output TEXCOORD5 register, and then, after rasterisation, gets in the pixel shader input.

Animation of the fog layers is done by transforming noise textures coordinates. All textures' coordinates are translated according to the coefficient, which is modified in the main program, and set in the constant "Animation" register.

Here is the pseudocode of discussed vertex shader program:

```
VertexOutput main
      ( VertexInput IN,
        // constant registers:
        // combined world-view-projection matrix:
        uniform float4x4 WorldViewProj,
        // world matrix:
        uniform float4x4 World,
        // camera position:
        uniform float4 CameraPos,
        // textures animation coefficients:
        uniform float4 Animation)
{
 // computing the vertex's world position:
 float4 WorldPos  = mul (World, IN.Position);
 // calculating the world distance
 // from vertex to camera:
 OUT.Distance = distance(CameraPos, WorldPos);

 // calculating the noise textures coordinates:
 float3 Coords;
 // transformation of the noise textures' coordinates:
 OUT. NoiseTexCoords0 = Coords + Animation.x;
 ... OUT. NoiseTexCoords3 = Coords + Animation.w;

 // passing the base texture coordinates on the output
 OUT.MainTexCoords = IN.MainTexCoords;
 // computing the vertex's
 // world-view-projection position:
 OUT.Position  = mul( WorldViewProj, IN.Position );
}
```

4.2 Pixel Shader program

Pixel shader program is making the same computations for each pixel of the rasterised scene separately. Input data registers are almost the same as vertex shader's output, with except of POSITION register, which is only used for rasterisation, and is not used in the pixel shader. The input registers are described by the following structure:

```
struct PixelInput {
    float2 BaseTexCoords   : TEXCOORD0;
    float3 NoiseTexCoords0 : TEXCOORD1;
    float3 NoiseTexCoords1 : TEXCOORD2;
    float3 NoiseTexCoords3 : TEXCOORD3;
    float3 NoiseTexCoords4 : TEXCOORD4;
    float3 Distance        : TEXCOORD5;
};
```

The noise octaves' textures are placed in the pixel shaders' constant registers, which also contain the base texture and fog settings. With use of them and input data, the fog factor is computed. The result of pixel shader's calculations is final "fogged" color of the considered pixel, which is passed in the output described by the following structure:

```
struct PixelOutput {
    float4 Color : COLOR;
};
```

Pseudocode of the pixel shader program is presented below:

```
PixelOutput main
       ( PixelInput IN,
         // constant registers:
         // base texture:
         uniform sampler2D BaseTex,
         //four octaves of Perlin noise:
         uniform sampler3D NoiseTex0,
         uniform sampler3D NoiseTex1,
         uniform sampler3D NoiseTex2,
         uniform sampler3D NoiseTex3,
         // homogenous fog settings:
         uniform float4 Fog)
{
 // setting color of fog
 float4 FogColor = float4(0.8f,0.8f,0.8f,1.0f);
 // read the base color
 float4 BaseColor = tex2D(BaseTex, IN.BaseTexCoords);

 // read the noise values
 float k0 = tex3D(NoiseTex0, IN.NoiseTexCoords0);
 ... float k3 = tex3D(NoiseTex3, IN.NoiseTexCoords3);

 // computing turbulent fog's density distribution:
 float Turbulence;
 Turbulence = (k0 + k1/2.0f + k2/4.0f + k3/8.0f);
 // calculating fog's density according to turbulent
 // and homogenous density coefficients:
 float Density = Fog.Density * Turbulence;
 // calculating the fog factor
 float f = exp(-pow(Density * IN.Distance.x,2.0f));

 // blending the scene and fog colors:
 OUT.Colour = lerp(FogColor, BaseColor, f);
}
```

5 Results

Application testing was performed on a machine with the nVidia's GeForce FX 5200 chipset, which possesses full DirectX 9 support with vertex and pixel shaders' 2.0+ versions. The test scene consists on about 10000 triangles. For such number, the animation speed is about 20 frames per second.

Before demonstrating the results of application working, let's see the figure 5 presenting the example homogenous fog effect. It was simulated with the exponential equation (3). Generated image looks quite good but not natural enough. It comes from the fact, that fog's density is constant over the whole scene, so only the distance fading can be observed.

Figure 5: Homogenous exponential fog

The realism of simulated phenomenon may be improved by differentiation of the fog's density. It can be achieved by using the noise textures. The appearance of fog effect depends on the textures' settings. Changing their resolutions has influence on the condensation of mist's layers. Smoothness of the effect depends on the range of texture colors.

The results of application working are shown in the following figures. Figure 6 presents the heterogenous fog effect applied to the scene according to the (7) formula. The g coefficient is set up on 0.044, and 3D noise textures resolutions are: 16^3, 32^3, 64^3, 128^3. In figure 7 effect obtained with the (6) formula can be seen, with the same settings. Figures 8 and 9 are demonstrating influence of changing textures resolutions, on the fog's appearance. The noise frequency is decreased, what is expressed by the lower texture resolutions (8^3, 16^3, 32^3, 64^3). Figure 10 shows the result of modification of the g coefficient, enlarged to 0.06.

Figure 6: The fog effect applied with the formula (7) (g=0.044, texture resolutions: 16^3, 32^3, 64^3, 128^3)

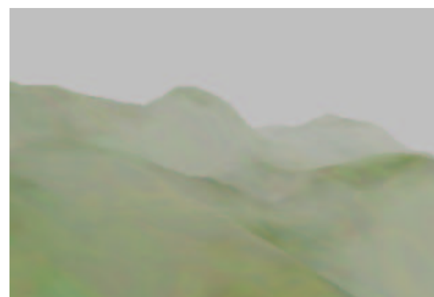

Figure 7: The fog effect applied according to the formula (6) (g=0.044, texture resolutions: 16^3, 32^3, 64^3, 128^3)

As seen in the presented pictures, different settings of fog's attributes produce various images of simulated phenomenon. The lower the texture resolutions, the larger and smoother fog's layers. Controlling the animation settings provokes different directions and speeds of the fog's movement. Thanks to this the wind blows can be simulated.

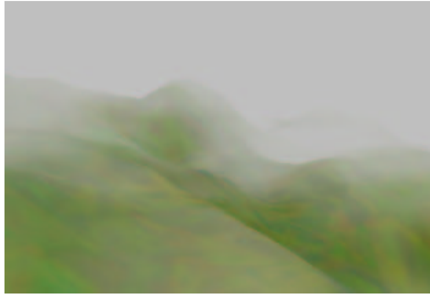

Figure 8: The fog effect obtained with the formula (7) (g=0.044, texture resolutions: 8^3, 16^3, 32^3, 64^3)

Figure 9: The fog effect applied with the formula (6) (g=0.044, texture resolutions: 8^3, 16^3, 32^3, 64^3)

Using of the Perlin noise and turbulence produces much more realistic looking fog effect than homogenous methods do, it also makes possible the mist movement animation. However the results show that employing a suitable method of homogenous fog computation also has a big influence on simulated phenomenon's appearance. Using the exponential square (4) formula causes better visibility near the observer, and covering the objects situated far away. The exponential (3) equation generates fog, which more softly decreases the range of visibility. By changing homogenous fog's density factor g it is possible to control the global field of view, which decreases when g rises, and vice versa.

6 Conclusions

Applying the Perlin's turbulence function allows obtaining realistic fog effects. In connection with homogenous fog factor computing, Perlin's turbulence method produces natural looking fog layers with visibility decreasing with the distance from the observer.

In the presented implementation, noise textures are generated in preprocessing, what causes that fog's layers to repeat during the animation. To solve this defect, noisy density distribution could be calculated in real time while the applications executes.

Figure 10: The fog effect applied with the formula (6) (g=0.06, texture resolutions: 8^3, 16^3, 32^3, 64^3)

This paper can be the lead into the future works on modelling and animation of natural phenomena, such as clouds or more complex haze. In the future elements of raytracing could be applied to obtain additional atmospheric effects as the light influence, shadows caused by fog layers, different hours of a day.

Constant evolution and improvement of modern GPUs will make possible executing more and more advanced and mathematically complex algorithms on the graphics hardware without large cost in computation time.

References

[1] Dan Baker, Charles Boyd. Volumetric Rendering in Realtime.

[2] Douglas Rogers. Implementing Fog in Direct3D. *NVIDIA Corporation*, 2000.

[3] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[4] Wolfgang Heidrich, Rudiger Westermann, Hans Peter Seidel, and Thomas Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *Symposium on Interactive 3D Graphics*, pages 127–134, 1999.

[5] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 1990.

[6] Jos Stam, Eugene Fiume. Turbulent wind fields for gaseous phenomena. *Computer Graphics*, 27(Annual Conference Series):369–376, 1993.

[7] Justin Legakis. Fast Multi-Layer Fog. *ACM SIGGRAPH Conference Abstracts and Applications, Technical Sketch*, 1998.

[8] Ken Perlin. An Image Synthesizer. *SIGGRAPH*, pages 287–296, 1985.

[9] Kim Pallister. Generating Procedural Clouds in Real Time on 3D Hardware. *Intel Corporation*, 2000.

[10] NVIDIA Corporation. NVIDIA nfiniteFX Engine: Programmable Pixel Shaders. *http://www.nvidia.com*.

[11] NVIDIA Corporation. NVIDIA nfiniteFX Engine: Programmable Vertex Shaders. *http://www.nvidia.com*.

[12] OpenGL Architecture Review Board. *OpenGL Programming Guide*. Addison-Wesley, Third edition, 2002.

[13] Paul Bourke. Perlin Noise and Turbulence. *http://astronomy.swin.edu.au/p̃bourke/texture/perlin/*, 2000.

[14] Philipp Slusallek. From Physics to Rendering. 1996.

[15] Simon Premoze. Light Transport in Participating Media. *Light and Color in the Outdoors, A SIGGRAPH Course*, 2003.

[16] Jos Stam. Stable fluids. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 121–128, Los Angeles, 1999. Addison Wesley Longman.

[17] V. Biri, S. Michelin and D. Arques. Real-Time Animation of Realistic Fog. *Thirteenth Eurographics Workshop on Rendering*, 2002.