Real-Time Rendering of Light Shafts on GPU

Shuyi Chen, Sheng Li, and Guoping Wang

Lab. of HCI & Multimedia, School of Electronics Engineering and Computer Science,
Peking University, China, 100871
{lisheng, wgp}@graphics.pku.edu.cn

Abstract. In the past, it is difficult to simulate light shafts effect in real-time. One major reason is the high computational expense to perform the physically-accurate computation of atmosphere scattering. Another is due to the limitation of computer resource, especially lack of power and programmability in the graphic hardware. Recently, with the advent of more powerful graphic card in standard PC platform and the development of programmable stages in the graphic pipeline, a lot of computational expensive algorithms are made available in modern commercial games. In this paper, we propose a novel method of rendering light shafts with atmospheric scattering based on actual physical phenomena. The proposed method utilizes hardware frame buffer object and a mesh refinement pattern to achieve photorealistic effect at high frame rate.

1 Introduction

Realistic image synthesis is one of the most important research subjects in computer graphics. To create physically-accurate realistic image, the effect of the scattering and absorption of light due to atmospheric particles is one of the most important elements to be taken into consideration. This effect mainly includes sunlight, skylight, aerial perspective and light beams caused by headlights of automobiles, street lamps, studio spotlights, and light passing through stained glass windows. During the past, those effects were seldom rendered correctly in real time. In computer games, simple texture blending or hardware fog was generally used to simulate light shafts. However, texture blending cannot handle the scenario when the viewer is totally inside the shaft volume. And hardware fog is totally wrong in terms of the physical model. Even though many physical models have been proposed to render light shafts, yet, few of them can be done in real time.

Recently, the processing speed of graphics card has been becoming faster and faster. In addition, the vertex processing unit (called vertex shader) and pixel processing unit (called pixel shader) have now become fully programmable. With the advent of programmability inside the graphics pipeline, a lot of expensive operations, which are used be done on CPU sequentially for each vertex, can now be carried out in parallel on GPU. Therefore, study of GPU-accelerated rendering is one of the most important research areas for real-time rendering.

G. Bebis et al. (Eds.): ISVC 2006, LNCS 4291, pp. 161 – 169, 2006.

In this paper, we proposed a rendering method of light shafts with atmosphere scattering by making use of GPU. The proposed method utilizes the programmability and parallel architecture of the GPU to display light shafts in real-time. Also, this method can handle shadow in atmosphere.

The paper is organized as follows. Previous work on rendering light shafts is discussed in Section 2 and an overview of the atmospheric scattering model and shading model we utilize is described in Section 3. In section 4, the core of our GPU-accelerated rendering method is proposed. Results and several examples are presented in Section 5. Finally, we draw the conclusion and discuss our future work.

2 Previous Work

A simple method to simulate the scattering and absorption due to atmospheric particles is to attenuate colors of objects according to the distance from the viewpoint. This method is computationally inexpensive and is implemented as one of the standard graphics APIs in OpenGL. However, this method is based on a heuristic function and is totally wrong in terms of actual physical phenomena. Hence, a more accurate model is required to simulate the atmospheric scattering effect.

To simulate the actual phenomena of atmospheric scattering, several models for atmospheric scattering have been proposed [1][2][3]. Based on Nishita's model, Dobashi proposed hardware-accelerated methods to render light shafts [4][5]. His method utilizes hardware texture blending and hardware Gouraud shading function to accelerate the rendering process and achieve interactive frame rate. However, due to the lack of programmability in the graphics pipeline, his method requires multiple passes and exploits little parallelism. Besides, in his method, the objects in the scene are required to render multiple times each frame to create the atmospheric shadow.

Our work uses the same shading model with Dobashi's, and is an extension and optimization on current programmable GPU. Also our proposed method only require two passes each frame and complex models in the scene are only rendered once to create the atmospheric shadow.

3 Overview of the Shading Model

Our method utilized a model proposed by Nishita [1]. We first describe the physical model briefly [6]. For the sake of simplicity, we only consider the case when there is only one light source. And our method can handle multiple light sources in a straightforward way.

Fig.1 shows the concept of atmospheric scattering. Here, a point light source is assumed. For parallel light source, the case will be simpler. In general, the intensity along a ray from the object reaching to the viewpoint is expressed by the Eq.1:

$$I_{eye}(\lambda) = I_{obj}(\lambda) * \beta_{\lambda}(T) + \int_{0}^{T} F_{\lambda}(\alpha) H(t) I_{p}(\lambda, t) \beta_{\lambda}(t) dt \tag{1}$$

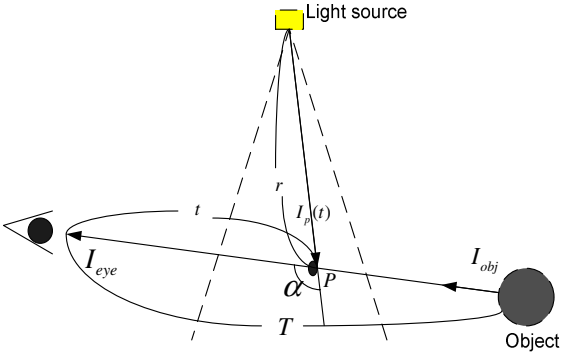

Fig. 1. Shading model for atmospheric scattering

where $I_{eye}(\lambda)$ is the intensity reaching the viewpoint, $I_{obj}(\lambda)$ is the intensity of an object, $\beta_\lambda(t)$ the attenuation ratio due to atmospheric particles between the viewpoint and a point P on the viewing ray, t the distance between the viewpoint and point P, T the distance between the viewpoint and the object, and $I_p(\lambda,t)$ the intensity of light from the light source reaching point P. $H(t)$ is a visibility function that returns the value 1 if the light source is visible from point P, or 0 otherwise. $F_\lambda(\alpha)$ is a phase function of the atmospheric particles and α is the phase angle (see Fig. 1). Because we only consider atmospheric scattering near to the ground, $\beta_\lambda(t)$ can be given by the Eq.2:

$$\beta_\lambda(t) = e^{-\beta_{sc}^\lambda t} \tag{2}$$

where β_{sc}^λ is the scattering coefficient for light with wavelength λ. And for point light source, $I_p(\lambda,t)$ is given by the Eq.3:

$$I_p(\lambda,t) = I_\lambda(\theta,\varphi)e^{-\beta_{sc}^\lambda r} / r^2 \tag{3}$$

where $I_\lambda(\theta,\varphi)$ is the intensity of light emanating from the light source toward the direction of point P and (θ,φ) indicates the direction. If the light source is a parallel light source, $I_p(\lambda,t) = I_0(\lambda)$, where $I_0(\lambda)$ is a constant. The phase function can be given by a weight sum of the Mie scattering and Rayleigh scattering, and the weight is selected according to atmosphere condition [7][8][9].

In Equation 1, the first term account for out-scattering, which is called aerial perspective, while the second term I_s account for in-scattering. The calculation of the first term is somewhat obvious, and can be easily implemented in shader. The calculation of the second term can be written as:

$$I_s = \sum F_\lambda(\alpha)H(t)I_\lambda(\theta,\varphi)\xi(\lambda,t) \tag{4}$$

$$\Delta I_s = F_\lambda(\alpha)H(t)I_\lambda(\theta,\varphi)\xi(\lambda,t) \tag{5}$$

$$\xi(\lambda,t) = e^{-\beta_{sc}^\lambda t}e^{-\beta_{sc}^\lambda r}\Delta t / r^2 \tag{6}$$

A fast method to calculate I_s is proposed in the following section.

4 GPU-Accelerated Rendering of Light Shafts

In order to compute the scattering effect of light shaft volume, virtual planes should be placed in front of the viewpoint firstly for sampling. Each virtual plane is parallel to the screen and is represented by a $n_u \times n_v$ lattices mesh (see Fig.2). For a viewing ray v, I_s is computed numerically by taking samples at intersections between the ray and the virtual planes. For point light source, $I_\lambda(\theta,\varphi)$ can be precompiled and used as a texture in rendering. Also we pre-calculate $F_\lambda(\alpha)$ and the result is then loaded into the graphic memory as a texture. Δt is a constant on the viewing ray since the sampling planes are placed at the same intervals.

To take into account shadow, $H(t)$ must be calculated for each lattice point. In our method, the camera is first placed at the position of the light source, then the scene is rendered and the color buffer is written with each pixel's depth value instead of each pixel's color. This color buffer, which contains the depth information of the objects in light space, is then used as a floating point depth texture to calculate $H(t)$ in later stage. The calculation of depth value can be done in shader. First, we calculate the position of each vertex in light space in vertex shader, and the result is assigned to a varying variable. Then, the fragment shader read the varying variable from output of the vertex shader, now this variable contains the position of the fragment in light space, and we write the z coordinate of this fragment into the color buffer. Also in this stage, since only the geometry information is required, texture and lighting are disabled to accelerate rendering.

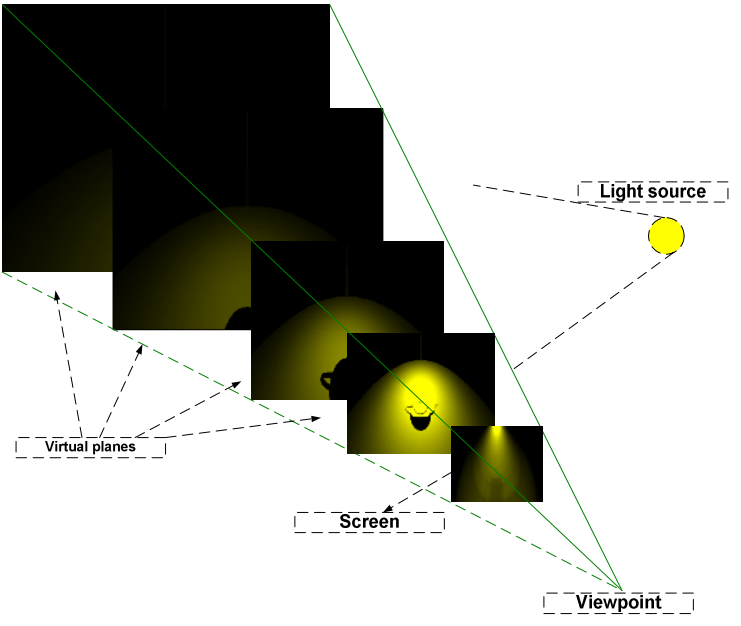

Fig. 2. Virtual planes and light computation on each lattices

In addition, every virtual plane is divided evenly into a $n_u \times n_v$ lattices mesh and $\xi(\lambda,t)$ is calculated for each lattice point. In our method, the calculation of the light intensity of each lattice point can be achieved in parallel by using a generic mesh refinement pattern [10]. This pattern is used to create additional inner vertices for each virtual plane by using the coordinates of its four corners. It is defined as a set of 2-tuple (u_t, v_t).

$$u_t = ((V_1 - V_0) \bullet (P - V_0)) * (P - V_0) / (|V_1 - V_0|^2 * |P - V_0|) \tag{7}$$

$$v_t = ((V_3 - V_0) \bullet (P - V_0)) * (P - V_0) / (|V_3 - V_0|^2 * |P - V_0|) \tag{8}$$

where $P(x_t,y_t,z_t)$ is a lattice point on a virtual plane, V_0, V_1, V_2, V_3 are the four corners of the virtual plane respectively (see Fig.3). Since each virtual plane is evenly divided into a $n_u \times n_v$ lattices mesh, the pattern is the same for all the virtual planes. Therefore, this pattern is uploaded into the graphics memory once during initialization, and is transferred from graphics memory to the vertex processing unit each time we draw a virtual plane. To calculate the world position and light intensity of each lattice point on the virtual plane, we send the world coordinates of V_0, V_1, V_2 and V_3 into the graphics pipeline as uniform variables and draw the pattern (each 2-tuple (u_t, v_t) in the pattern is regarded as a 2D vertex coordinate). In vertex shader, we read V_0, V_1, V_2 and V_3 and calculate the world coordinate of each lattice point using the Eq.9.

$$P = v_t * (V_3 - V_0) + u_t * (V_1 - V\ ) + V_0 \tag{9}$$

$\xi(\lambda,t)$ is then calculated using the world coordinate. Finally, $\xi(\lambda,t)$ of other points on the virtual plane can be interpolated by using their adjacent lattice points, and the interpolation is performed using Gouraud shading.

In the fragment shader, we compute the depth value (in light space) of the fragment and compare it with stored value in the depth texture (the lookup of the stored value can be achieved using projective texture mapping algorithm), if the fragment's depth in light space is greater than the stored value, it is then in shadow, $H(t)$=0, otherwise, $H(t)$=1. At last, $F_\lambda(\alpha)$ and $I_\lambda(\theta,\varphi)$ are read back from textures respectively and used to calculate ΔI_s together with $H(t)$ and $\xi(\lambda,t)$.

The procedure for our rendering method using OpenGL is summarized as follows:

1) Preprocess $F_\lambda(\alpha)$ and $I_\lambda(\theta,\varphi)$ as textures and upload them into the graphics memory. Precompute the pattern of the lattice mesh and upload it into the graphics memory through a static Vertex Buffer Object.
2) For each frame, attach a floating point depth texture to a frame buffer object, and place the camera at the position of the light source, then the scene is rendered into the frame buffer object and color buffer is updated with the depth value of the objects. Since we only care about the depth value, texture and lighting can be disabled to accelerate rendering.
3) For each virtual plane, send the coordinates of its four corners to vertex shader as uniform variables, and draw the pattern of the lattice mesh. In the shader, the process of each lattice point is as the following:
   a) Calculate the eye coordinates of the lattice points using the four corners of the virtual plane and the refinement pattern.

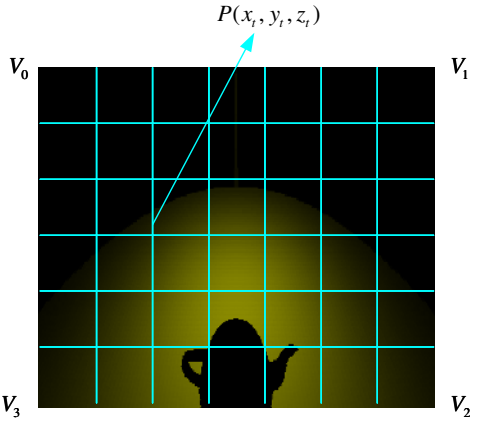

Fig. 3. Virtual plane refinement pattern

b) Calculate the light coordinate of the lattice point. By using the projective texture mapping technique, we can lookup the corresponding light intensity I_λ and depth value **z** (in light space) for each lattice point in texture $I_\lambda(\theta,\varphi)$ and the depth texture respectively.

c) Calculate the depth of the lattice point in light coordinate, and compare it with **z**. If **z** is smaller, this lattice point is in shadow. Otherwise, we lookup $F_\lambda(\alpha)$ from texture and compute ΔI_s.

4) All the virtual planes are rendered using additive blending function.

5 Experiments

Our experiments are performed on a PC workstation (Pentium 4 3.2GHz HT, 1 Gigabyte RAM) with ATI X800 GTO. All the images are rendered at the resolution of 800x640. Some results are shown in table 1. We find out that the efficiency of our method is inversely proportional to the number of virtual planes. Also, as the refinement pattern becomes finer, the computational time increases. This should be due to

Table 1. Statistics data of light shaft rendering

Figure No.	Virtual planes	Lattices points	Depth texture	Light-map	Frame rate (fps)
4(a)	50	32x32	256x256	64x64	30
4(b)	50	64x64	256x256	64x64	26
4(c)	50	64x64	512x512	64x64	24
4(d)	50	64x64	512x512	256x256	22
4(e)	100	32x32	256x256	64x64	16
4(f)	100	96x96	512x512	128x128	10
5(a-d)	100	96x96	512x512	128x128	9

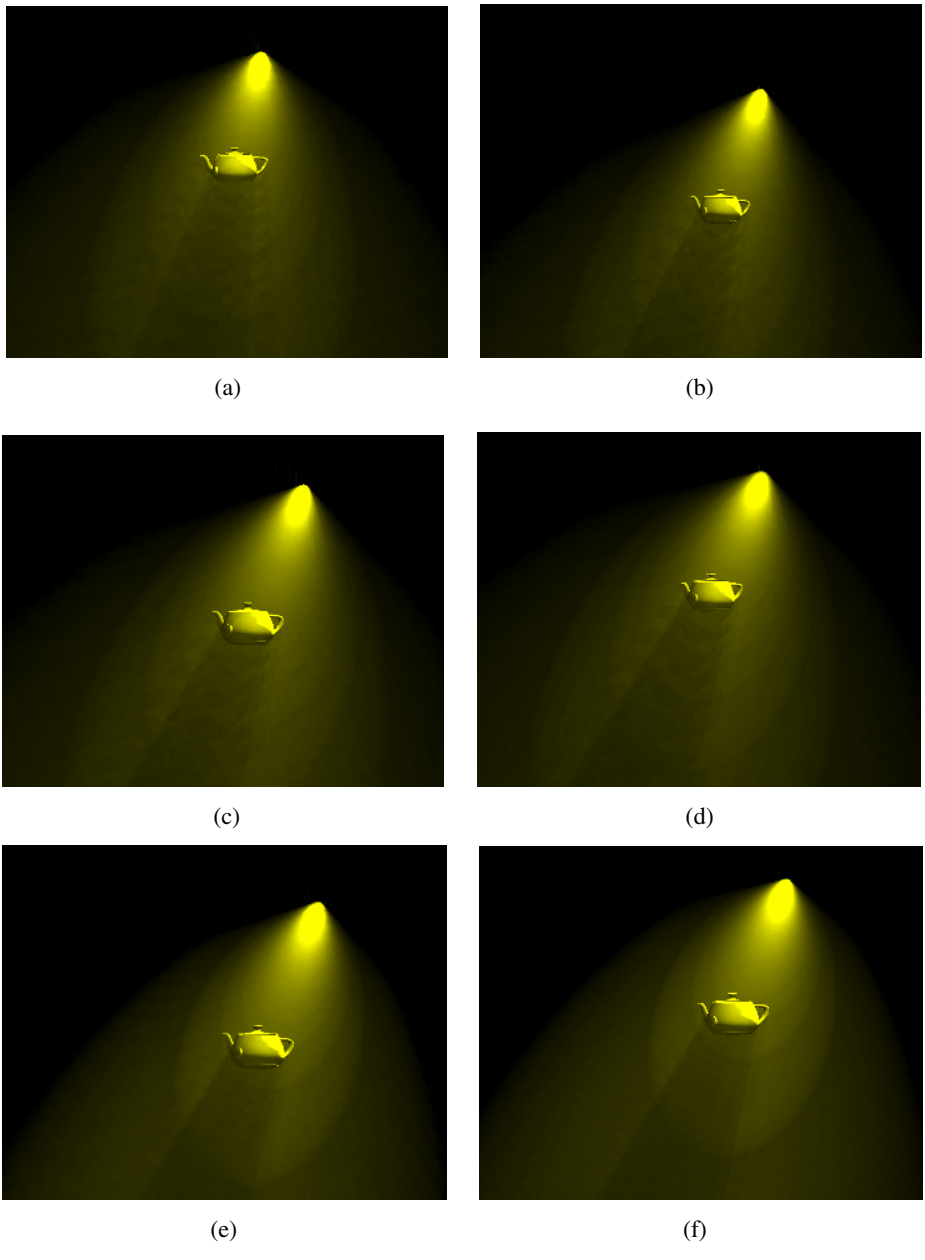

Fig. 4. Rendering of the light shaft using different rendering setting listed in Tab. 1

a bottleneck between the vertex fetch unit and the vertex processing unit, since there is usually 6-8 parallel vertex processing pipeline while the vertex fetch unit might fetch more vertices at a time. Besides, as we increase the resolution of the depth

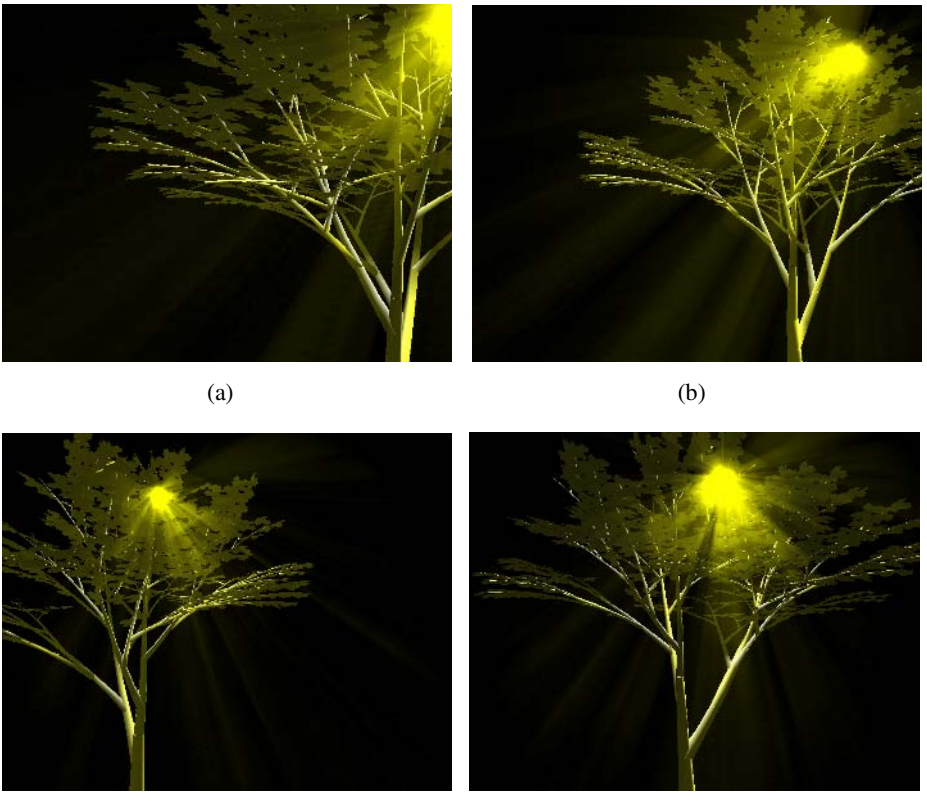

Fig. 5. Rendering of the light shaft when viewpoint located in the shaft volume under a complex scene

texture, the performance drops a little bit. This should be due to the bandwidth bottleneck between GPU and graphics memory. What is more, in Fig. 5(a-d), the scene complexity is around 43k triangles, and the performance drops only 10% when compared with the scene showed in Fig. 4.

6 Conclusion

In this paper, we have proposed a new method of rendering light shafts with atmospheric scattering in real-time. The proposed method makes use of the programmability and parallel architecture on modern GPU to achieve fast frame rate and photorealistic effect. The advantages of our method are as follows.

1) By utilizing the depth texture, our method requires two passes for each frame. The depth texture is only rendered at the beginning of each frame, and is read multiple times for each virtual plane. Hence, the objects in the scene casting

shadows are rendered twice in total at each frame. (One is for the depth texture, the other is for the scene displayed on the screen)

2) Our method utilizes a mesh refinement pattern, and this pattern is the same across all the virtual planes. It is uploaded as a static vertex buffer into graphics memory and reused multiple times when we draw a virtual plane. Therefore, the footprint between CPU and GPU is greatly reduced, making full use of the fast transfer rate between GPU and graphics memory (around 35GB/s).
3) In the past, $\xi(\lambda,t)$ of each lattice point is calculated sequentially on CPU. In our method, by making use of a mesh refinement pattern, $\xi(\lambda,t)$ of lattice points can be calculated in parallel inside the graphic pipeline.

In future work, we need to develop a method to render multiple scattering in real-time [9]. Also, the sampling error due to the numerical integral in Equation 4 needs to be solved in order to create more photorealistic images.

Acknowledgements

This research work is supported by National Key Project of Fundamental Research (973 Project No. 2004CB719403), NSFC (60573151, 60473100).

References

1. T. Nishita, Y. Miyawaki, E. Nakamae, A Shading Model for Atmospheric Scattering Considering Distribution of Light Sources, Computer Graphics, 21(4):303–310, 1987.
2. R. V. Klassen. Modeling the Effect of the Atmosphere on Light. ACM Transactions on Graphics, 6(3): 215-237, July 1987.
3. K. Kaneda, T. Okamoto, E. Nakame and T. Nishita. Photorealistic image synthesis for outdoor scenery under various atmospheric conditions. The Visual Computer 7, 5 and 6, 247-258, 1991.
4. Y. Dobashi, T. Yamamoto, T. Nishita. Interactive rendering method for displaying shafts of light, In Pacific Graphics, 2000.
5. Y. Dobashi, T. Yamamoto, T. Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. Proceedings of the conference on Graphics hardware, 2002.
6. A. Preetham, P. Shirley, B. Smits, A Practical Analytic Model for Daylight, Proc. SIGGRAPH'99, 91–199, 1999.
7. L. Rayleigh. On the scattering of light by small particles. Philosophical Magazine 41, 447-451, 1871.
8. K.N. Liou. An Introduction to Atmospheric Radiation. International Geophysics Series Volume 84. Academic press. 2002.
9. T. Nishita, Y. Dobashi, K. Kaneda, and H. Yamashita. Display method of the sky color taking into account multiple scattering. In Pacific Graphics, pp. 117-132, 1996.
10. T. Boubekeur, C. Schlick. Generic Mesh Refinement on GPU, in: Proceedings of ACM SIGGRAPH/Eurographics Graphics Hardware, 2005.