Real-time Cloud Rendering

Yifang Cao
Yifang.Cao@stonybrook.edu

ABSTRACT
This project report is intended to address the details of implementing the method for realistic real-time rendering of clouds as described in the paper of Mark J. Harris and Anselmo Lastra [1]. The project is implemented with OpenGL/C++, which implements both shading and rendering algorithm and dynamic impostor generation technique. The report will illustrate both mathematical concepts of rendering techniques and detailed explanations of OpenGL/C++ implementation. Since this cloud rendering technique is intended to be used for games and flight simulators, this report will also explain how to implement first person view camera and the skybox to realize a flying environment.

Keywords
Computer graphics, cloud rendering, real-time, flight simulation, anisotropic, multiple scattering.

1. INTRODUCTION

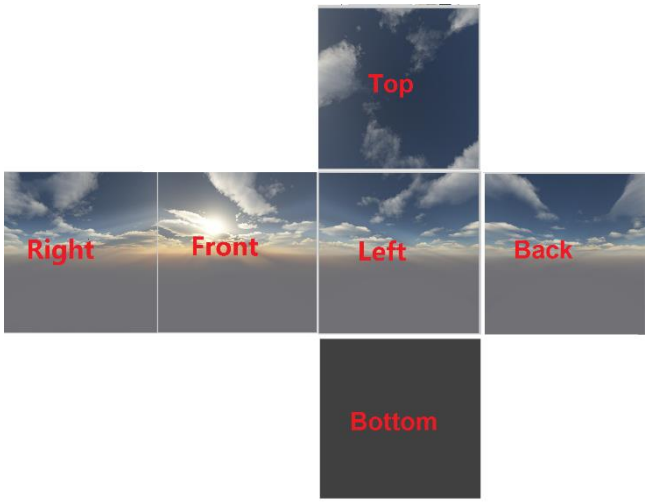
The paper of Harris aprovides a shading algorithm to render static clouds, which approximates multiple forward scattering in a preprocess ahead of time, and first order anisotropic scattering at run time. Impostors are also suggested to speed rendering frame rate, along with an easy solution to reduce artifacts caused by direct impostors rendering. The result of this project demonstrates a constant-shape clouds (particles are static relative to each other) illuminated from two light sources and the interactive flight simulation, in which an object fly within and around realistic clouds.

Section 2 introduces a simple environmental setup for a flight simulation, including drawing a skybox and providing first person view camera controls. Section 3 gives a derivation and description of the shading algorithm, and implementation details in OpenGL/C++. Section 4 discusses the concept of dynamically generated impostors.

2. Flight Simulation Setup

2.1 Skybox

A skybox employs a cube to create background of the scene. The cube texture mapping is used to add distant three dimension surrounding illusion. The texture used in this project consists of six separate images in TGA format. A TAG loader class is used in the project to load TAG images. Each image is mapped to a side of a cube.

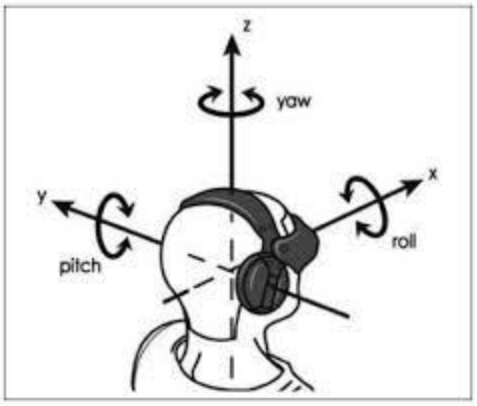


Each side of the cube is drawn separately with GL_QUADS, and slightly overlap with its neighboring sides to get rid of visible seams. The lighting is disabled so that the skybox's light intensity and color could be fixed. The size of the skybox is a bit smaller than the maximal clipping distance. The skybox responds to rotational transformation only. Any translation is ignored.

2.2 First Person Camera

The first person view camera controls are needed to translate and rotation objects in the scene to simulate a flying experiment. For simplicity, this project implements a First Person Shooting style

view camera, which only supports rotations along the yaw and the pitch.



To make rotation simple, the eye is positioned slightly off the origin and looks at the origin with an upright vector by calling gluLookAt. The rotation angles could be obtained through mouse dragging.

The translations depend on the current rotation angles. Thus, the translation values need to be carried out with proper trigonometric functions. The equation set for moving forwards is:

$$\text{eyeX} -= \sin(\text{yaw}) * \text{SPEED}$$
$$\text{eyeY} += \sin(\text{pitch}) * \text{SPEED}$$
$$\text{eyeZ} += \cos(\text{yaw}) * \text{SPEED}$$

The equation set for moving leftwards is:

$$\text{eyeX} -= \cos(\text{yaw}) * \text{SPEED}$$
$$\text{eyeZ} -= \sin(\text{yaw}) * \text{SPEED}$$

The equation sets for moving downwards and rightwards are just the inverse of the above. For translations, instead moving the eye position, we translate all the objects in the scene to the negative of the intended direction. Thus, the position of the eye never needs to be moved.

3. Cloud Rendering

3.1 Particle System

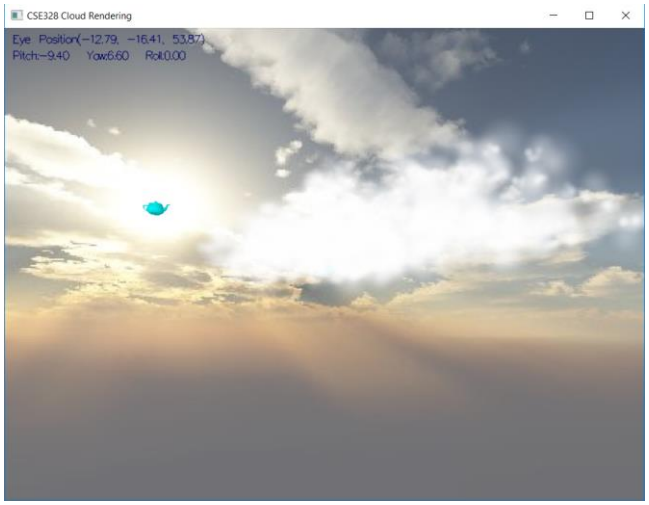
As the paper mentions that the cloud rendering algorithm can be Used with any model composed of discrete density sample in space. The project uses particles to represent puffs that make up a whole cloud. Each particle object stores its own location information, RGBA color data, and size.

Particles are rendered using screen-oriented billboards mapped with a splat texture. The splat texture has highest opacity at the center, and the opacity falls out towards its sides. The texture can be generated with the following equation:

$$\boldsymbol{v_p = v_0 * (2f^3 - 3f^2 + 1)}$$

v_p is the alpha value of any pixel p on a billboard. v_0 is the alpha value at the center. The value of v_0 can vary depend on the style of clouds. f is the Euclidian distance between the center divided by the radius of the billboard and the pixel p. The project uses 64*64-pixel billboards. Thus 64*64 values are pre-computed and stored to make the billboard texture. Then, each billboard can be drawn by plat texture mapped GL_QUADS with color equal to particle's RGBA value and the corresponding size and location of the particle.

To make all the billboards always face towards the eye location, the current modelview matrix needs to be loaded, erase the rotation values, and then load this modelview matrix back. When drawing billboards, the depth test needs to be disabled to make the edges invisibles. At this point, the cloud can be successfully rendered without light attenuation by the rendering method described in Section 3.3. The following image shows a cloud without light attenuation.



Cloud without Light Attenuation

3.2 Preprocess for Multiple Scattering

3.2.1 Understanding the Math and Physics

In reality, scattering illumination models simulate the emission and absorption of light by a medium as well as scattering through the medium. The multiple scattering algorithm described in paper is more physically accurate than the single scattering algorithm. With multiple scattering that samples N directions on the sphere, each additional order of scattering that is simulated multiplies the number of simulated path by N.

The cloud rendering technique is a two-pass algorithm. The first pass preprocesses cloud shading, and store the shading information for rendering in the real time. Specifically, the preprocess pass computes the amount of incident light at each position $\boldsymbol{P}$ from direction $\boldsymbol{l}$. The incident light $\boldsymbol{I}(\boldsymbol{P},\boldsymbol{l})$ is composed of all driect light from direction $\boldsymbol{l}$ that is not intervened by particles along the path, plus light scattered from other particles to position $\boldsymbol{P}$. The multiple scattering algorithm is following:

$$I(P,w) = I_0 e^{\int_0^{Dp}\tau(t)dt} + \int_0^{Dp} g(s,w)dt\, e^{-\int_s^{Dp}\tau(t)dt}ds$$

$\boldsymbol{I_0}$is the ligh intensity in direction w ouside the cloud. $\boldsymbol{\tau(t)}$ is the extinction coefficient of the cloud at depth t, $\boldsymbol{Dp}$ is the depth of $\boldsymbol{p}$ in the cloud along the light direction. The function $\boldsymbol{g(s,w)}$ is defined as

$$g(x,w) = \int_{4\pi} r(x,w,w')I(x,w')dw'$$

It represents the light from all direction $\boldsymbol{w'}$ scattered into direction $\boldsymbol{w}$ at the point x. $r(x,w,w')$ is the bi-directional scattering distribution function, and determines the percentage of light incident on x from direction $\boldsymbol{w'}$ that is scatterd in direction $\boldsymbol{w}$. It is defined as

$$r(x,w,w') = a(x)\tau(t)p(w,w')$$

, where $a(x)$ is the albedo of the medium at x, and $p(w,w')$ is the phase function. An accurate multiple scattering needs to calculate this quantity for a sampling of all light flow directions. However, as the paper suggest we only need to consider multiple scattering in the light direction l, and approximate over only a small solid angle γ around the forward direction, and assume r and I are constant because γ is small. Thus, the function is simplified as the following:

$$g(x,l) = r(x,l,-l)I(x,-l)\,\frac{\gamma}{4\pi}$$

Because we are using particles to represent the cloud, the light path can be split into discrete segments s_j, for j from 0 to N, where N is the number of particles along the light path from 0 to $\boldsymbol{Dp}$. Then, the light intensity $\boldsymbol{I_k}$ of any particle k can be expressed by the following recursive relationship:

$$I_{\boldsymbol{k}} = \begin{cases} g_{\boldsymbol{k-1}} + T_{\boldsymbol{k-1}} * I_{\boldsymbol{k-1}}, 2 \le k \le N \\ I_{\boldsymbol{0}}, k = 1 \end{cases}$$

It basically means light tracing is performed along the direction the light incident on any particle p_k is equal to the intensity of light scattered to $p_k$from p_{k-1} + its transparency. The paper suggests the following values used for implementation:

$$Extinction: \tau = 80$$
$$Albedo: a = 0.9$$
$$Solid\ Angle: \gamma = 0.09$$

3.2.2 OpenGL Implementation

The implementation of the preprocess pass can rely on frame buffer read back and alpha blending. The following OpenGL functions need to be setup before drawing:

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_ALPHA_TEST);
glAlphaFunc(GL_GREATER, 0);
glEnable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
```

```
lTexEnvi(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Then sort all particles in ascending order to the light position, and clear frame buffer with white color. The eye position should be temporally moved to a position outside the cloud along the light direction, where the current view through the eye right fits the size of the whole cloud. Set gluLookAt from sun direction to the center of the cloud, and perform glOrtho parallel projection to clip off everything else except the cloud itself.

For each particle P_k:

(1) Calculate number of pixels to sample within a solid angle $\gamma = 0.09$. This number is calculated by:

$$\text{Pixels} = Distance(light, P) * \gamma$$

(2) Use glReadPixels to sample all pixels in this small area centering at (P.x,P.y).

(3) Take the average of sampled pixel color values, I_k

(4) $I_k \mathrel{*}= light\ color$

(5) $P.color = a * \tau * I_k * \frac{\gamma}{4\pi}$

(6) $P.alpha = 1 - e^{-\tau}$

(7) Render P with color = $P.color * Phase$, and size = $P.size$.The value of the phase is 1.5 here. The phase will be explained in Section 3.4.

End_For

The following

3.3 Real-Time Rendering.

The color value of each particle has been stored in the preprocess pass. It is now ready for real time rendering. The implementation is quite similar with previous section. OpenGL functions need to be setup the same way as in Section 3.2.2.

Firstly, particles should be sorted in the descending order from the eye position to make sure the blending function works properly. Then, calculate l, which is direction from the light.

For each particle P_k:

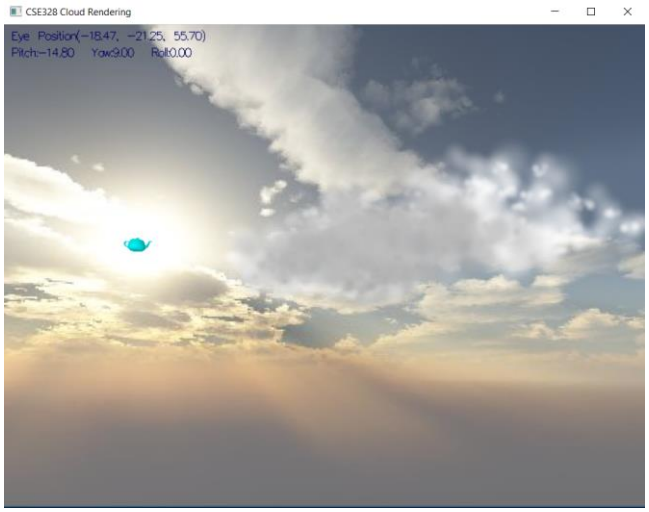
(1) Calculate the direction vector w, which is the direction from particle to the eye.

(2) c = $P.color * Phase(w, l)$

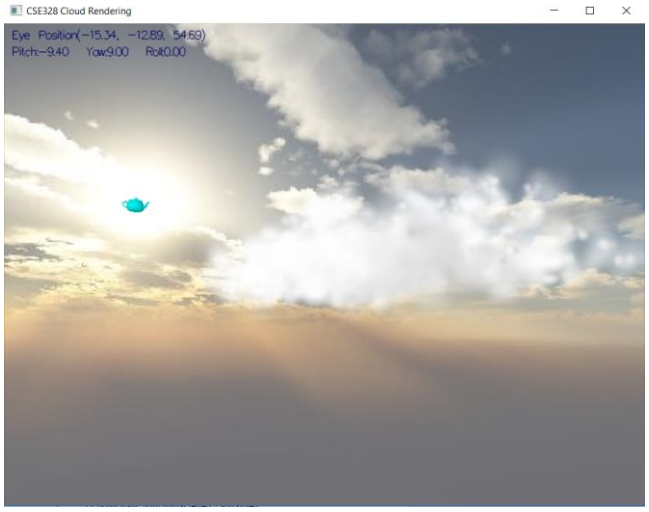
(3) Render P with color c, and size $P.size$

End_For

3.4 Phase Function and Anisotropic Scattering



Isotropic Scattering



Anisotropic Scattering

The phase function in the previous sections is a way to redistribute light scattering for a given incident light direction. This redistribution introduces anisotropic behavior to the cloud. The above two images demonstrate the same cloud with and without anisotropic scattering.

In this project, the phase function used is a simple Rayleigh scattering phase function

$$p(\theta) = 3/4(1 + \cos^2\theta)$$

The value (θ) is the angle between the incident an scattered directions. In section 3.2.2, the incident direction is l and the scattered direction is $-l$, for eye is along the light direction, thus the phase value for the preprocess is always 1.5.

$$p(\theta) = \frac{3}{4}(1 + \cos^2 0) = 1.5$$

4. References and Citations

4.1 Dynamically Generated Imposter

Because real-time particle rendering is a computationally expensive task, when the number of clouds become larger, it may reduce the running performance. In order to keep a high frame rate and reduce pixel overdraw, imposters can be deployed.

The procedure to generate the cloud imposters is like the preprocess pass. This procedure also move the view frustum to a position where the cloud's bounding volume is tightly fit the viewport. Instead sample a small area, this procedure reads the whole scene, and make the pixels read into a texture. glCopyTexSubImage2D can be used to efficiently grab the current framebuffer and converted into a texture. These textures can then be mapped onto billboards for rendering.

For clouds that are far away from the eye, their relative motion does not change their appearance too much, which allows us to re-use impostors in multiple continuous frames. To determine when to update the impostors, the paper introduces two error metrics:

(1) Translation Error: Angle induced by translation relative to the capture point.

(2) Zoom Error: the resolution of texture given by the equation

$$resolution\ of\ texture * \frac{Obj\ Size}{Obj\ Distance}$$

If one of the error exceeds the tolerance threshold, the imposter must be updated.

When the viewpoint is insight the cloud, it is impossible to bound the cloud size into the viewport. Instead, the texture should be grabbed from the same frustum used to display the whole scene, and render the impostor with screen size.

4.2 A Simple Solution to Reduce Artifacts.

One of the drawback of the impostors is that they are two-dimensional. When objects pass through it, artifacts can be observed. A simple solution to this problem is impostor splitting. For Each object in the cloud, the cloud split into two parts, one behind the object and one in front. Those split imposters are rendered from back to front. In this way, there will be no more intersections between objects and imposters.

5. Conclusion

The project provides an implementation of Harris and Lastra's real-time cloud rendering techniques, which includes multiple forward scattering, and dynamically generated impostors. In addition, the project provides first person view camera controls, and skybox background to simulate a flight simulator environment.

6. References

[1] Harris, Mark. J. and Lastra, Anselmo. (2001), Real-Time Cloud Rendering. Computer Graphics Forum, 20: 76–85. doi:10.1111/1467-8659.00500.