**Rendering HW2 Report**

**Realistic Camera Model Implementation**

Realistic Camera Constructor:

First, I read the lens specfile into my own data structure and used the thickness and radius to calculate the center of each lens for later use. Second, I used the xResolution and yResolution in parameter film to calculate the diagonal of Raster and used it and filmdiag to calculate the scale size of Raster to Film. Therefore, I can use it to get the transform RasterToCamera by scaling it from raster size to film size and translating it to correct coordinate. In the end, I calculated the weightCoeff(whole formula except the costheta^4) because the only part that will change in GenerateRay is the costheta.

Generate Ray:

There are three steps in generating rays: uniform sample from sample on film to first lens, refraction simulation of lens system, weight calculation.

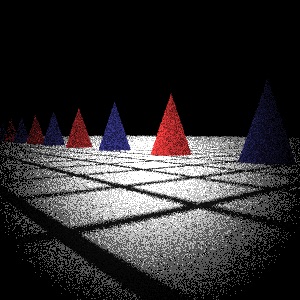
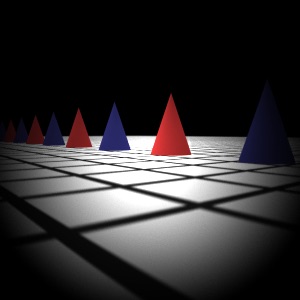
In uniform sample, I used UniformSampleDisk function in pbrt to get the coordinate on x-y plane from lensU and lensV in camera sample. Then, I multiplied x and y with the first lens’ aperture to make sure the ray can pass through first lens. At the end, I calculated the hit point on first lens and used it to create the ray from sample point on film to hit point for refraction simulation.

In refraction simulation, there are three steps: check the ray can pass through or not, using Snell’s law to get the new ray, update the ray for next lens’ refraction simulation. To check the ray can pass through or not, I used the radius and center point of lens to create a sphere and used it to check the intersection with ray. Then, I checked the hit point on sphere was outside the lens’ aperture or not. If there wasn’t any intersection or the hit point was outside the aperture, then return zero weight. After I got the hit point of lens, I used Hackbert’s method to get the new ray and updated the ray for next refraction simulation.

In weight calculation, I used the origin of ray and the sample point on film to calculate the cosine theta and got the weight with the weightCoeff calculating in Realistic Camera Constructor.

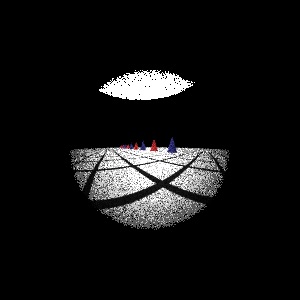
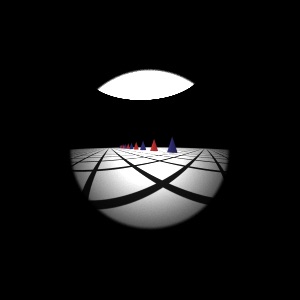
**Result**

Dgauss:

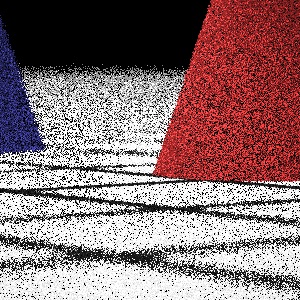
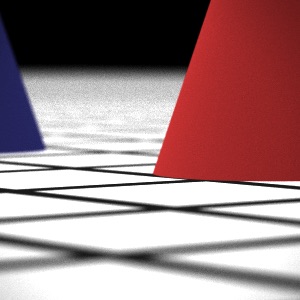
4 sample 512 sample

Fisheye:

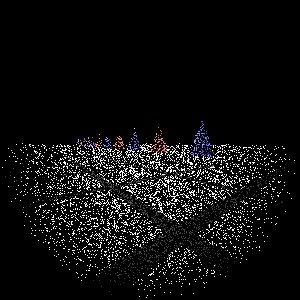
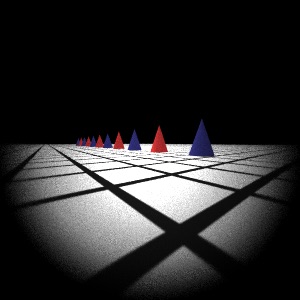
4 sample 512 sample

Telephoto:

4 sample 512 sample

Wide:

4 sample 512 sample

**Other**

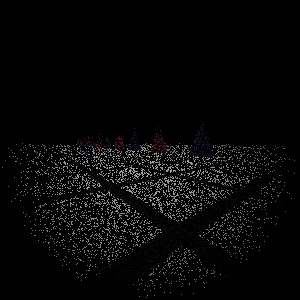
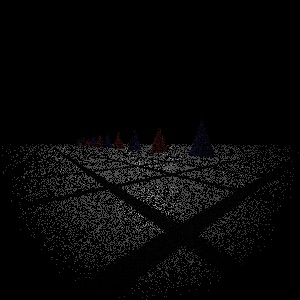
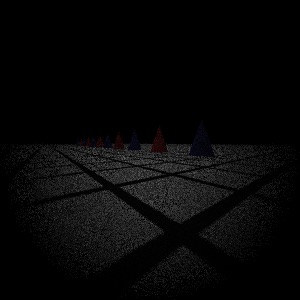
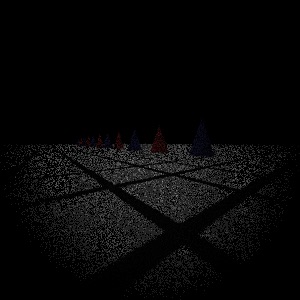
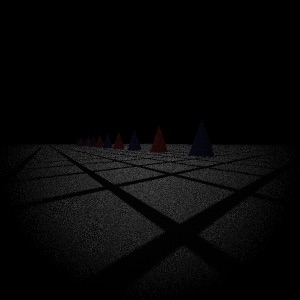
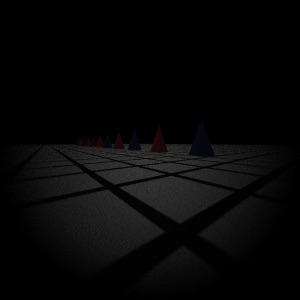
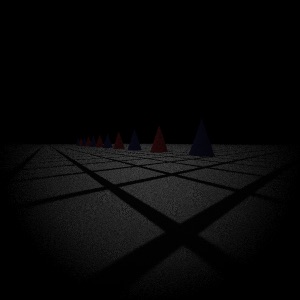
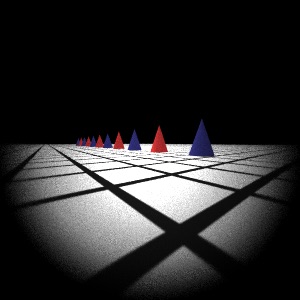
Table 1. Render time(sec)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **gauss** | **fisheye** | **telephoto** | **wide** |
| **4 sample** | 3.096 | 2.289 | 2.695 | 1.748 |
| **512 sample** | 318.593 | 218.931 | 250.545 | 132.987 |

According to Table 1 and all results in the Result section, I found that rendering 4-sample pictures is very fast but has very bad quality; rendering 512-sample pictures has good quality but consumes too much time. Thus, I want to find a balance between quality and time. Below are the pictures and time consuming of generating wide from 4-sample to 512sample.

Table 2. Render time (sec)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **4**  **sample** | **8**  **sample** | **16 sample** | **32 sample** | **64 sample** | **128 sample** | **256 sample** | **512 sample** |
| **wide** | 1.748 | 2.608 | 4.489 | 8.309 | 16.279 | 35.471 | 64.630 | 132.987 |

From Table 2, I found that the time consuming is linear. This means pbrt’s parallelization is good so that you spend twice time if you double the sample rate. From all results above, I found that pictures are clear enough if the sample rate is greater than 128. But the pictures are too dim until the sample rate is 512.

Besides, I have tried giving all ray 1.f weight like other built-in cameras in pbrt in realistic camera modeling. But the picture is too bright.

**Environment**

Mac OS X EI Capitan 10.11.1, 2.7GHz Intel Core i5 (4 cores), 8G Ram, Intel Iris Graphics 6100, build pbrt project by Xcode