Report for Project #2

DEADLINE: 11/17/2016, 14:00
INSTRUCTOR: Yung-Yu Chuang

王冠鈞 b03902027

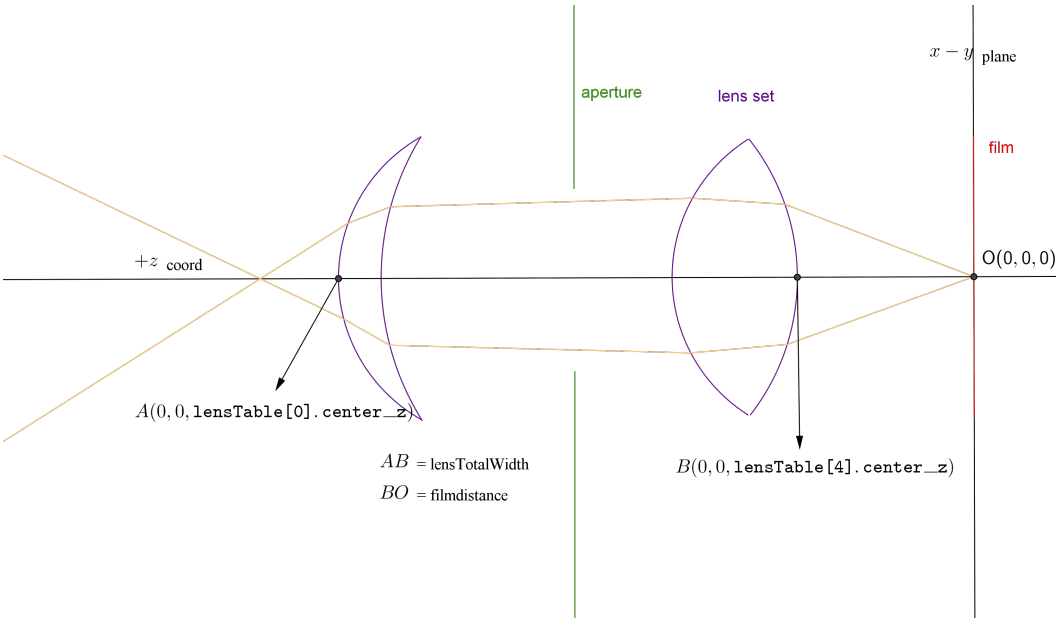
In this project, we are required to implement a new type of camera class, `RealisticCamera`, which is aimed to simulate the real-world cameras with complex lens. Unlike the simplicity of the existing cameras implemented in `pbrt-v2`, the new class requires a file telling the program about the data of lens, and the camera is expected to simulate the physical behaviours of the rays going through the lens, which requires some laws of physics studied from geometric optics. In the following sections, I'll have a thorough descriptions about my implementations, results, discoveries and pitfalls.

About `RealisticCamera` Class Implementation

The entire implementation requires: (1) completing the necessary class members and the constructor and (2) writing a function to generate sample rays. In the following, I'll describe how I finished each of the functions.

(1) *The class and the constructor.* Basically, the class just has to duplicate / save the parameters passed to the constructor, as well as read the `specfile` and create a lens table. In my implementation, the members in the private part of the class include several floating points, most of which are directly copied from the constructor function, except `lensTotalWidth`, which indicates the sum of all thickness of lens from rear to front as well as the inter-lens space, which will be used later.

For the lens table, I used a vector to save all the entries, and for the members of an entry is integrated into a class, `RealisticCameraLens`, along with two useful properties, `bool isAperture` (only true when n_d is 0), `float center_z` (the z coordinate of the intersection of the lens and the z-axis, will be explained later).



Before I continue, I'll first describe my camera space I constructed, and the example 5-entry (4 lens spheres and 1 aperture) lens set is shown as a figure[1] above. The transformed raster sample point will be on the $x-y$ plane, and will be generated in a square where the center is $(0,0,0)$. Also, the squared area in the $x-y$ plane is exactly the film. All the rays start from here. A the figure shown, all the lens surfaces are a part of some spheres, and I let the center point of the surface be on the z axis, with the coordinate pre-computed and stored in `center_z`. Thus the rays will start at the plane and have initial directions of positive z.

The class also has a private member, $TransformRasterToCamera$, which transforms a point from raster space to camera space, which includes a flip, a scale, and translations on $x-y$ plane.

Finally, the constructor just have to do all the works aforementioned (i.e. copy the parameters, read lens data and precompute its z coordinate, calculate transform matrix), and all the required works at the beginning are done.

(2) *The function* `RealisticCamera::GenerateRay` *and other affiliated private functions.* In my implementation, the function does the following things: (a) Get sample, generate initial ray, (b) trace through lens and (c) calculate weight and return.

(a) The function first transform the sample point into camera space, and choose a random point on the plane $z = filmdistance$. Since the random point generated by the system is bounded in $[0,1]$, it should be multiplied with the radius of the rear lens sphere's radius of aperture. With the two points, The function can create the initial ray.

(b) Then it goes through the iterations of lens tracing. it can be discussed into two cases. First, if the ray meets the center aperture, it just check whether it is stopped by the aperture or not. Otherwise, it must meet a lens sphere. In this case, it first checks whether it intersects with the sphere (the intersection algorithm is modified from the `Sphere::Intersection()` function), then it checks whether it intersects with the lens part of the sphere. After that, it will compute the refraction ray[2] and, if not total reflected, update its ray state.

(c) Finally, when it is confirmed that it will go through all the lens, calculate its weight, using the weight function provided by the paper given.[3]

Tests, Results and Images

In this project, the running environment I used is the CSIE Workstation server. I used linux9 for execution. linux9 has a CPU of 2.53 GHz clock rate along with 16 cores, and 99 GB of memory. For the execution test, I used 12 of the cores (`-ncores 12`).

The following table is the time for rendering each image, in terms of seconds:

	4 samples / pixel			512 samples / pixel		
time	pbrt	real	user	pbrt	real	user
`dgauss`	1.1	1.471	2.570	59.5	59.940	4:27.500
`fisheye`	0.8	1.193	1.630	40.4	40.827	3:12.077
`telephoto`	1.7	2.068	2.043	65.2	1:05.590	4:07.467
`wide`	1.0	1.527	1.260	28.7	29.110	2:10.120

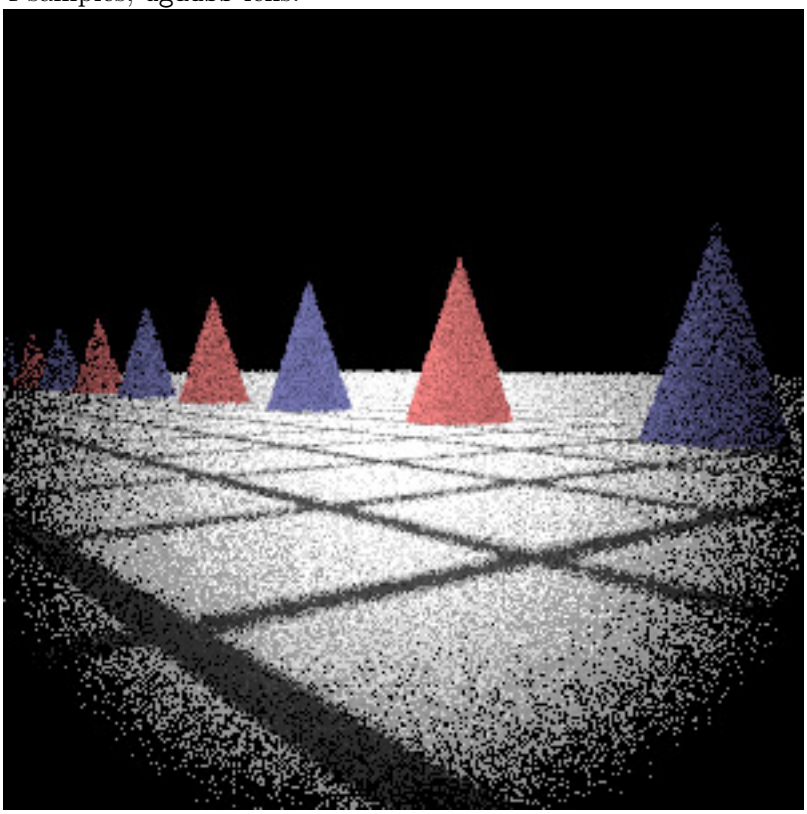
[1] Made with GeoGebra.

[2] I used Heckbert's Method here.

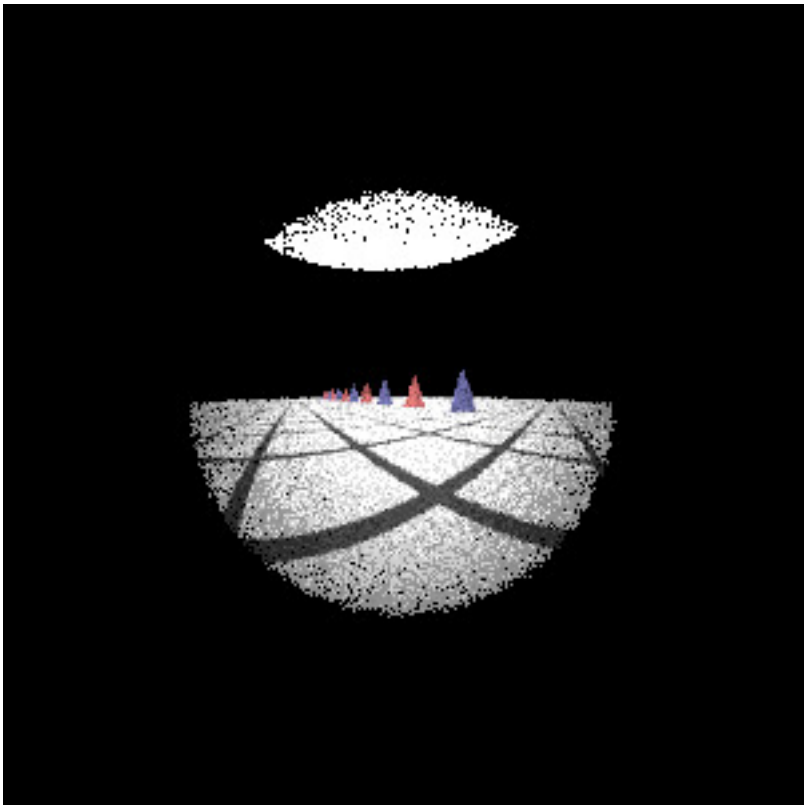
[3] $weight = A\frac{cos^4\theta}{Z^2}$, where A is the area of the rear lens, θ is the angle between the ray vector and the z axis, Z is `filmdistance`.

The following are the rendered images.

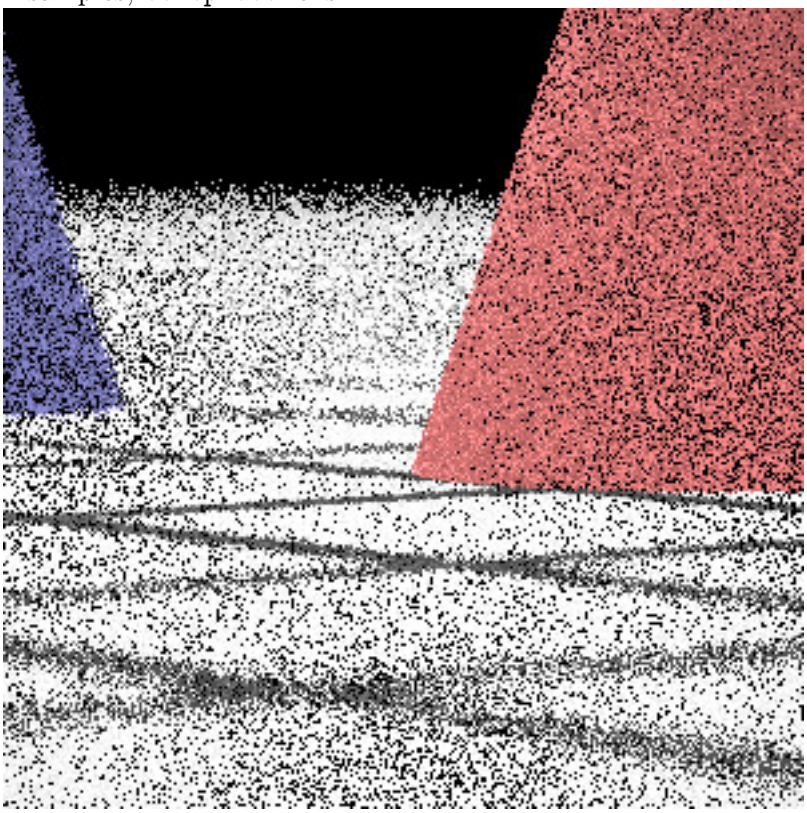
1. 4 samples, `dgauss` lens:



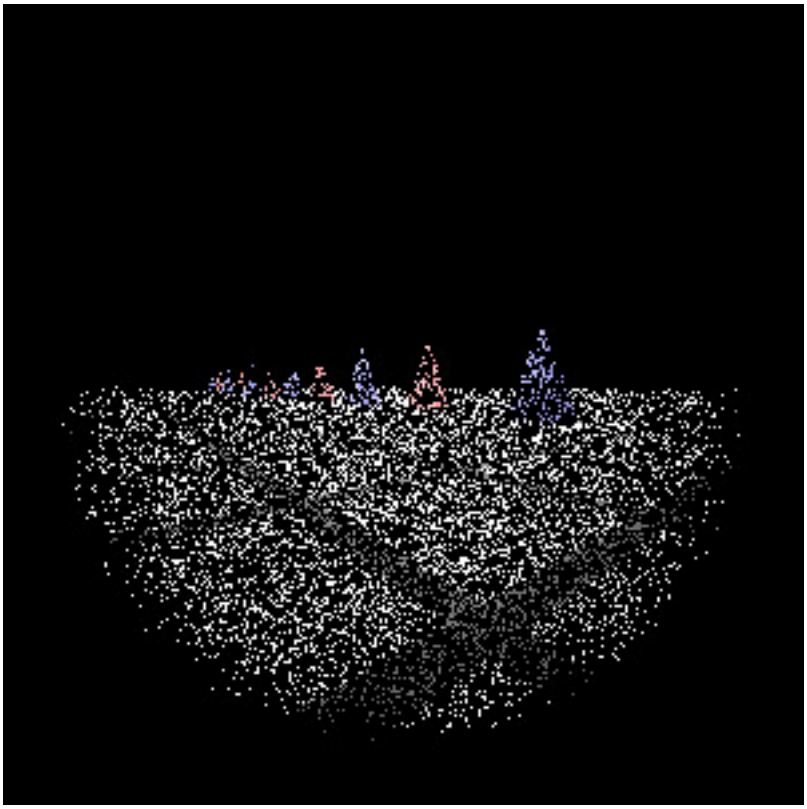
2. 4 samples, `fisheye` lens:



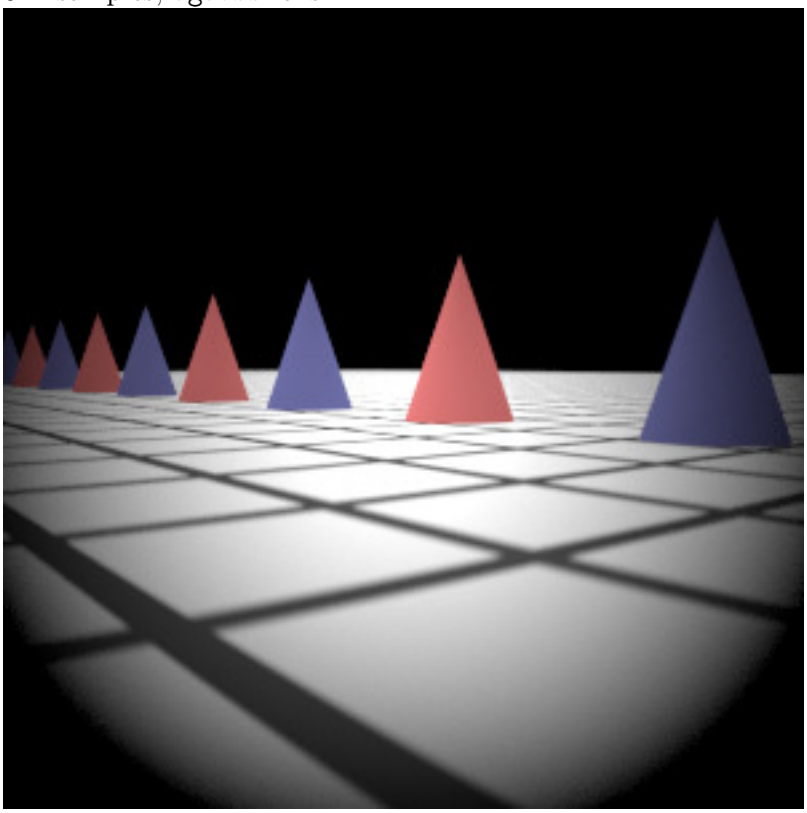
3. 4 samples, `telephoto` lens:



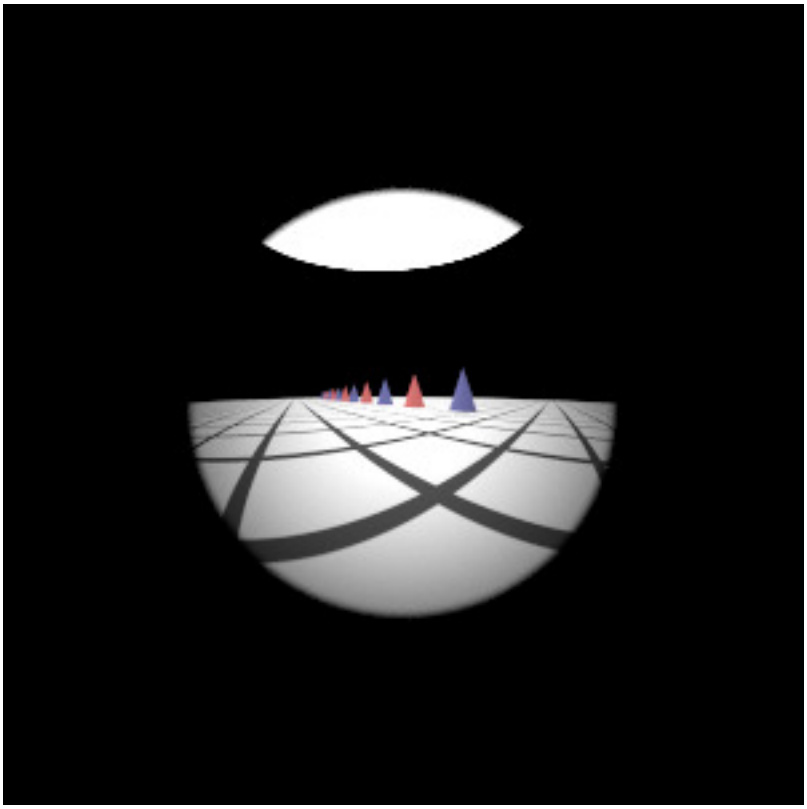
4. 4 samples, `wide` lens:



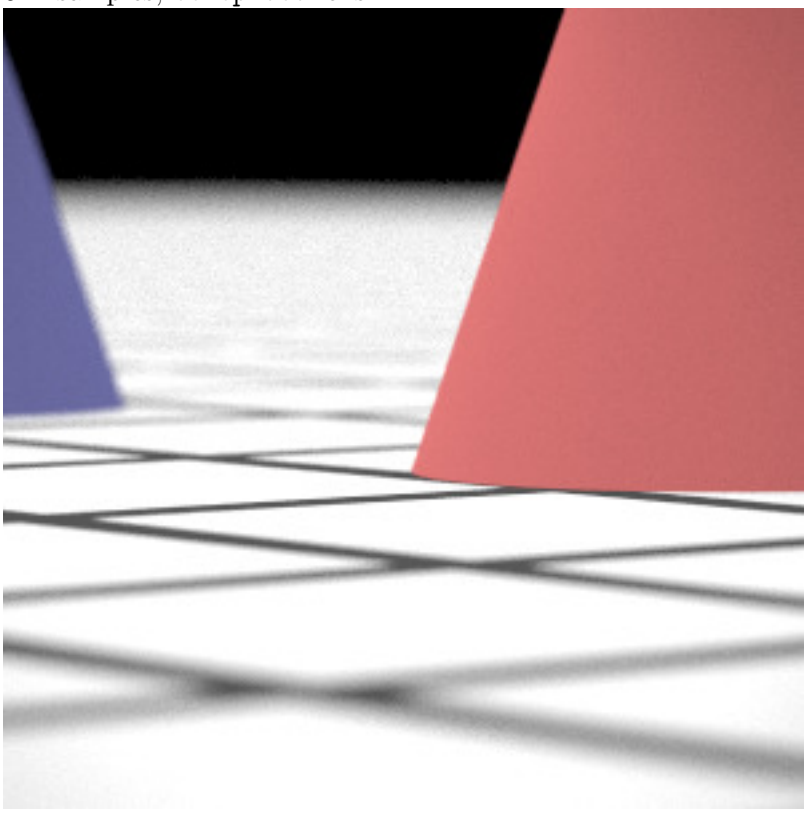
5. 512 samples, `dgauss` lens:



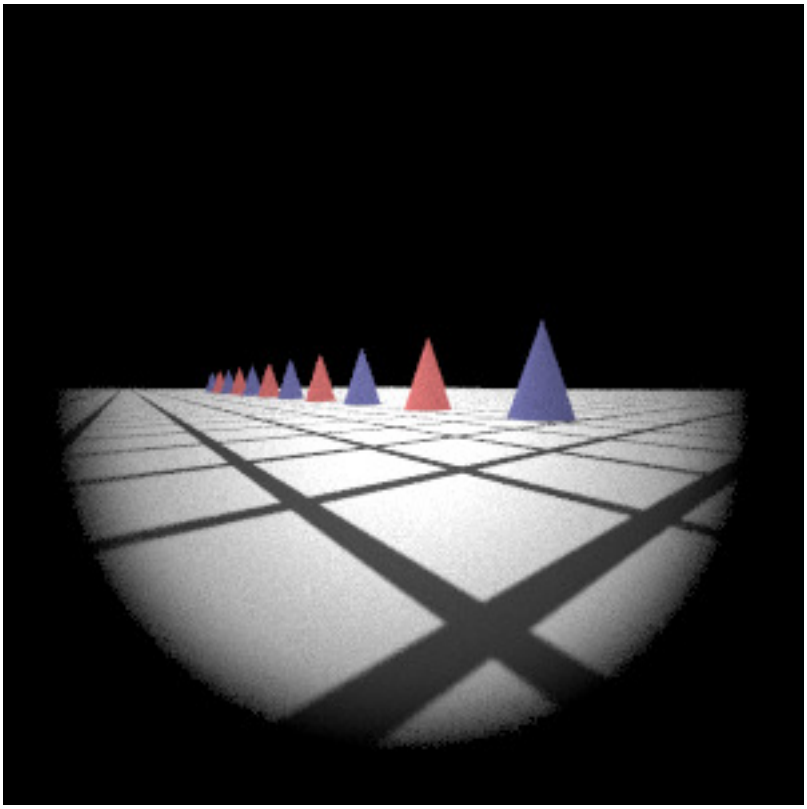
6. 512 samples, `fisheye` lens:



7. 512 samples, `telephoto` lens:



8. 512 samples, `wide` lens:



Discoveries and Pitfalls

It is important to fully understand the meanings of the variables, and the coordination systems (the transformation, in particular), or one will mess up everything. Also, there are some pitfalls on handling the refraction, especially the total reflection, as well as the direction of the normal when using Heckbert's Method.