Realize Spectral Ray Tracing

Author: Zihao Fang

ID: 120090326

Overview

The current implementation of our ray tracer cannot model dispersion and chromatic aberrations because its light model is not wavelength-dependent. Currently, indices of refraction are constant rather than different for each wavelength. Therefore, in this final project, we will implement spectral ray tracing based on assignment 6.

The current works in GitHub and CS184 course materials provide effective references for implementing the project. We have implemented wavelength-dependent BRDF to represent the effect of dispersion and color temperature.

This project is really challenging and time consuming, especially tasks' requirement seems come from another project and we haven't implemented some features of it in our assignments.

In next part, I'll briefly elaborate on work I've tried and made.

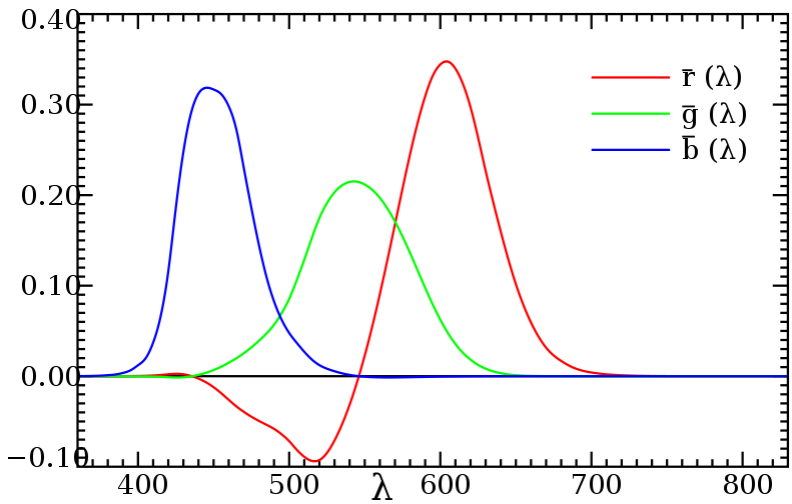
Implementation

1. Dispersion of the glass ball

1.1 Design and implementation

To support rendering different wavelengths of light, we refactored the struct of the `Ray` and function `Ray Camera::generate_ray(double x, double y, int color)` in `camera.cpp`. An argument `int color` is added to distinguish different type of lights. Different wavelengths are generated during the process of emitting rays according to different colors.

During ray generating procedure, a random number generator will generate a random seed. Three normal distribution of three kinds of lights will determine the wavelength of the light according to its color.



```
// wavelength generators
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);
std::normal_distribution<double> R(620, 30);
std::normal_distribution<double> G(530, 30);
std::normal_distribution<double> B(465, 15);
std::vector wavelengthList = {R, G, B};
// assign color and wavelength to the ray
ray.color = color;
ray.wavelength = wavelengthList[color](generator);
```

Due to different indices of fraction of lights with different wavelengths between different materials, we need to modify the function `BSDF::refract(const Vecotor3D wo, Vector3D *wi, double ior, double wavelength)` to receive an argument of wavelength and refactor the procedure of refraction.

```
auto eta = 1.000277 / ior;      // standard atmospheric pressure at 0 degree Celsius
auto fac = 589.29 / wavelength; // air refractive index at 589.29 nm
eta = eta / fac;
if (t < 0)  // existing case
    eta = 1.0 / eta;
```

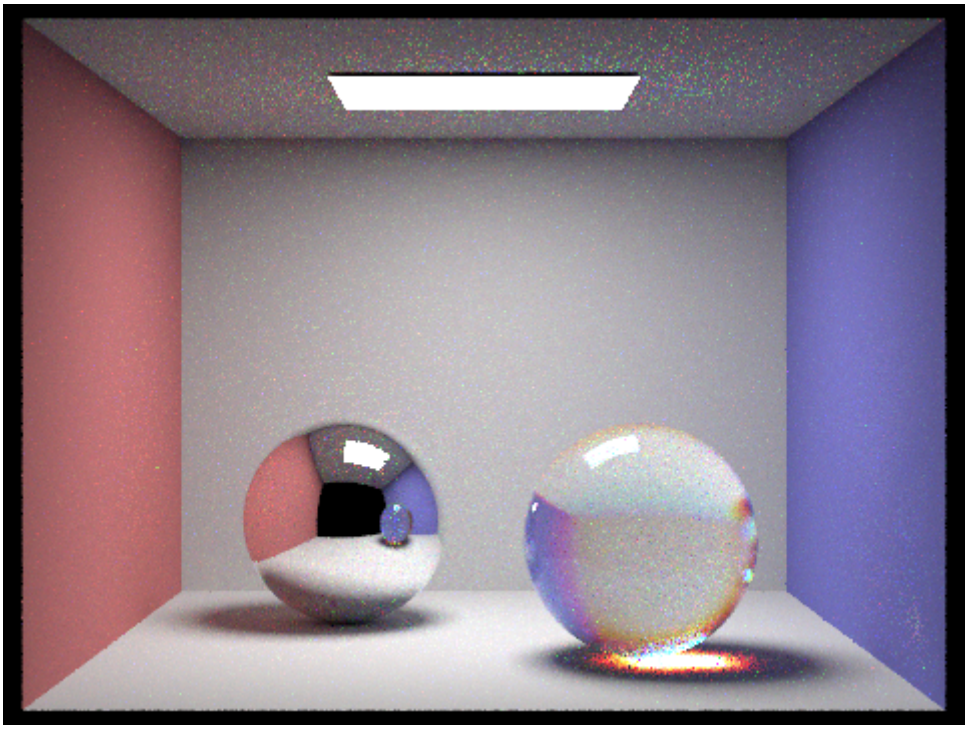
In this case, the index of refraction will change dynamically depending on the wavelength.

Furthermore, we need to modify the `PathTracer::raytrace_pixel(size_t x, size_t y)` to divide one ray into RGB channels. Then, combine these channels back and compute a mean value.

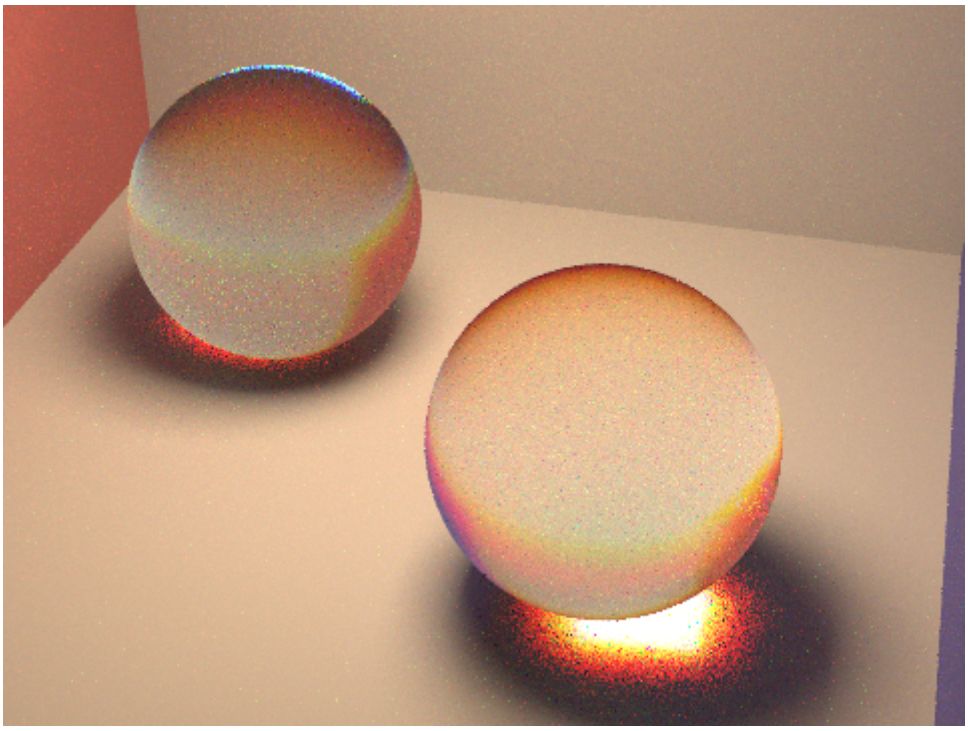
```
// take a few samples for each color channel
for (int i = 0; i < SAMPLE_PER_COLOR * 3; i++) {
    auto r = camera->generate_ray(sample.x / sampleBuffer.w, sample.y / sampleBuffer.h, i % 3);
    r.depth = max_ray_depth;
    rayList.push_back(r);
}
// combine three channels and divide by the number of channels
auto newRadiance = Vector3D();
for (const auto &r: rayList) {
    auto tmp = est_radiance_global_illumination(r);
    newRadiance[r.color] += tmp[r.color];
}
newRadiance /= (rayList.size() / 3);
```

1.2 Sample results

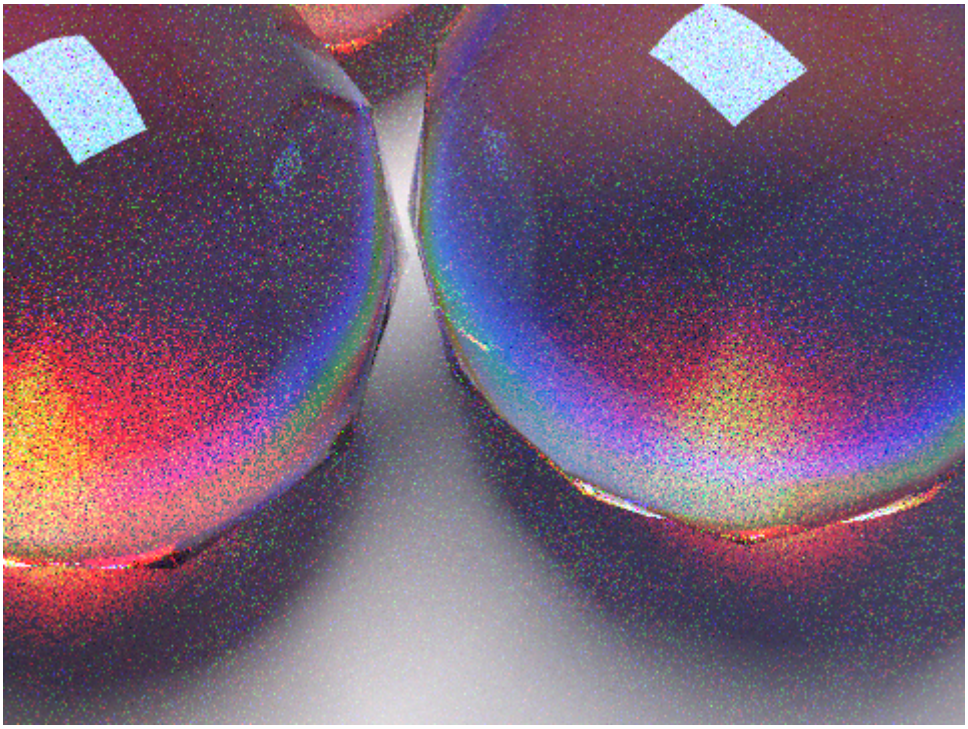
A real photo of a glass ball, and `CBspheres` rendered by us.



We can see that the light rays refracted in the glass sphere and formed a spot at the bottom. And the rendered result is quite similar to the photo in real.



We modified the spheres to gain a `.dae` with two glass spheres, and here is the result.



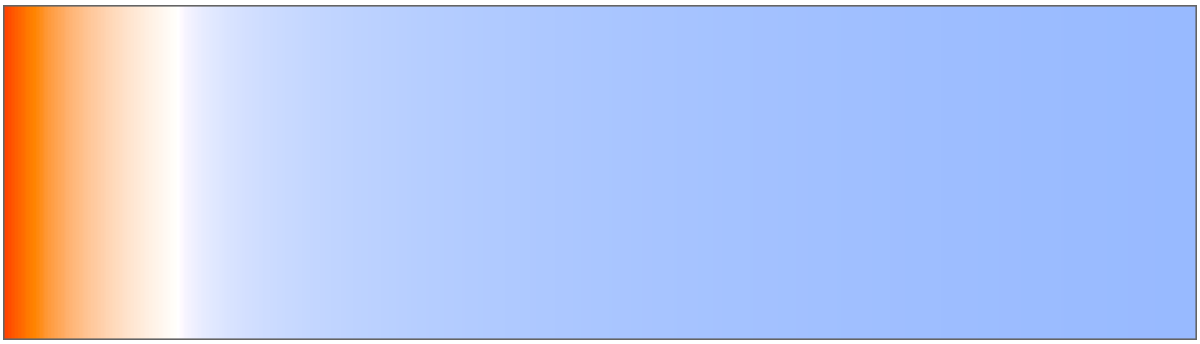
We zoomed in then rendered the `CBgems` and gained a beautiful scene with dispersion.

2. Color temperature

2.1 Design and implementation

We can describe the color of a visible light source by comparing it to how an ideal black body would look like if it had the same color. The color temperature is the temperature that the black body would need to have to match the color of the light source. We usually measure color temperature in kelvins. The color temperature scale only tells us the color of the light source, not how hot it really is. Sometimes, the light source is much cooler than its color temperature suggests.

A sample output ranging from 1000K to 40000K looks like this:



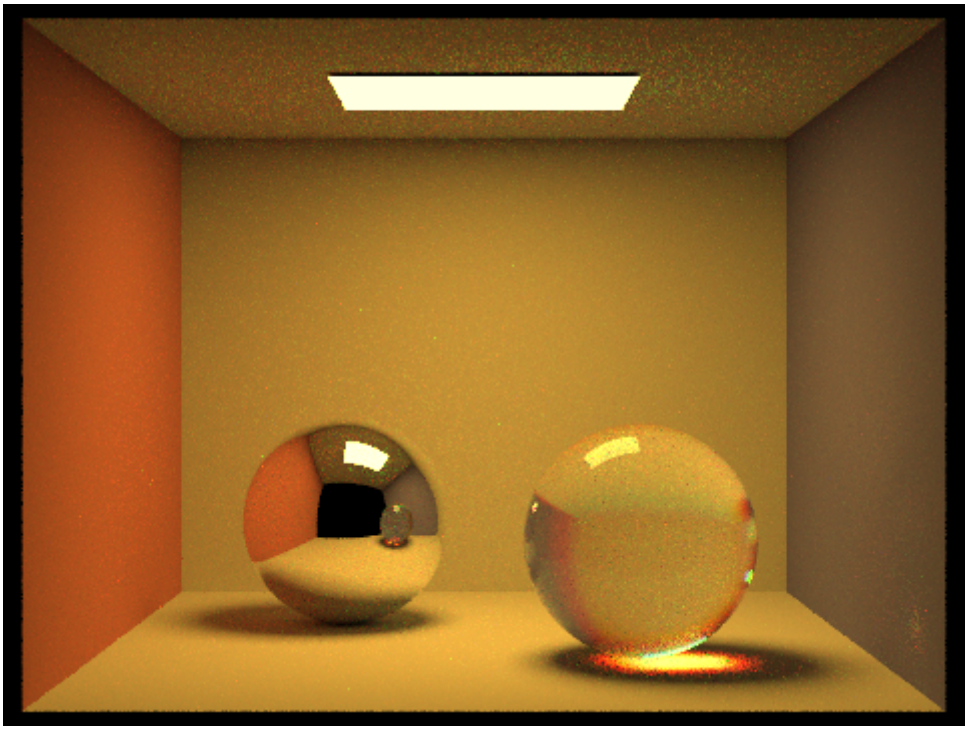
We implemented a function to perform the transformation `PathTracer::color_temperature(int temperature, int color)`. Our algorithm is referred to the idea of Tanner Helland.

The function looks like this

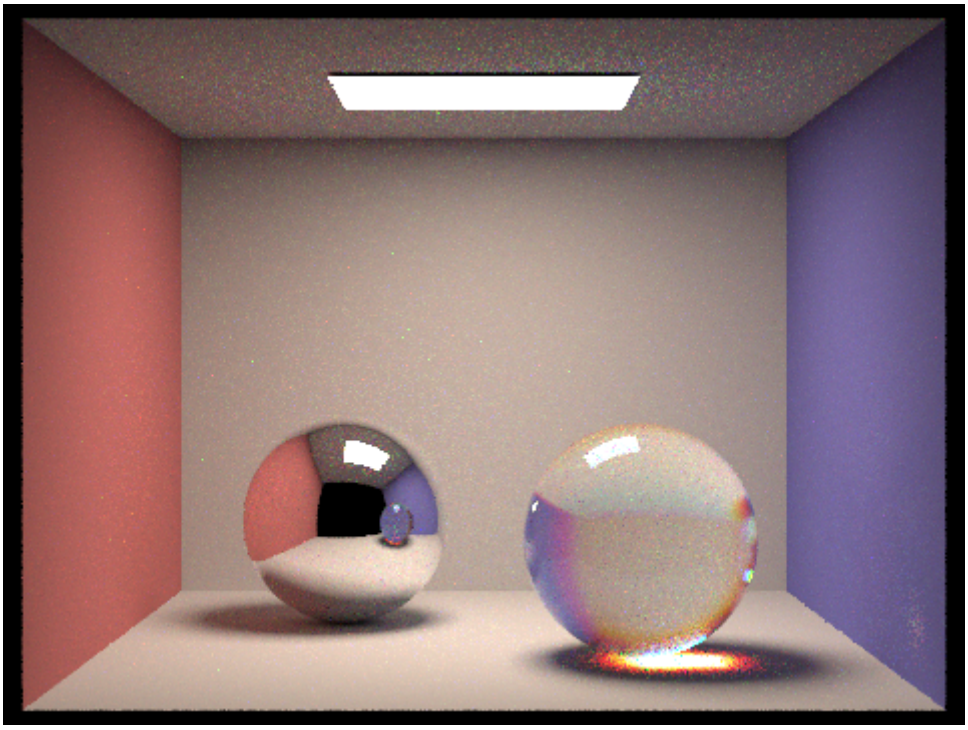
```
double PathTracer::color_temperature(int temperature, int color) {
    double radiance;
    temperature /= 100;
    switch (color)
    {
        case 0: // red
            if (temperature <= 66) {
                radiance = 255;
            } else {
                radiance = temperature - 60;
                radiance = 329.698727446 * (pow(radiance, -0.1332047592));
                if (radiance < 0) radiance = 0;
                if (radiance > 255) radiance = 255;
            }
            break;

        case 1: // green
            if (temperature <= 66) {
                radiance = temperature;
                radiance = 99.4708025861 * log(radiance) - 161.1195681661;
                if (radiance < 0) radiance = 0;
                if (radiance > 255) radiance = 255;
            } else {
                radiance = temperature - 60;
                radiance = 288.1221695283 * pow(radiance, -0.0755148492);
                if (radiance < 0) radiance = 0;
                if (radiance > 255) radiance = 255;
            }
            break;

        case 2: // blue
            if (temperature >= 66) {
                radiance = 255;
            } else {
                if (temperature <= 19) {
                    radiance = 0;
                } else {
                    radiance = temperature - 10;
                    radiance = 138.5177312231 * log(radiance) - 305.0447927307;
                    if (radiance < 0) radiance = 0;
```

```
                    if (radiance > 255) radiance = 255;
                }
            }
    }
    return radiance;
}
```

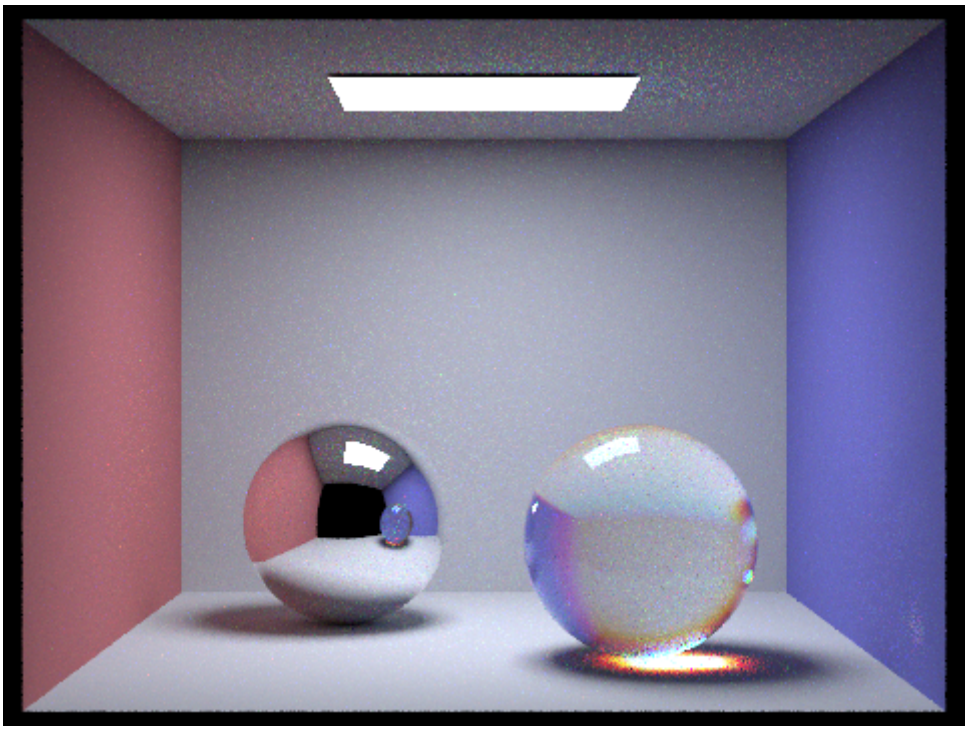
2.2 Sample results



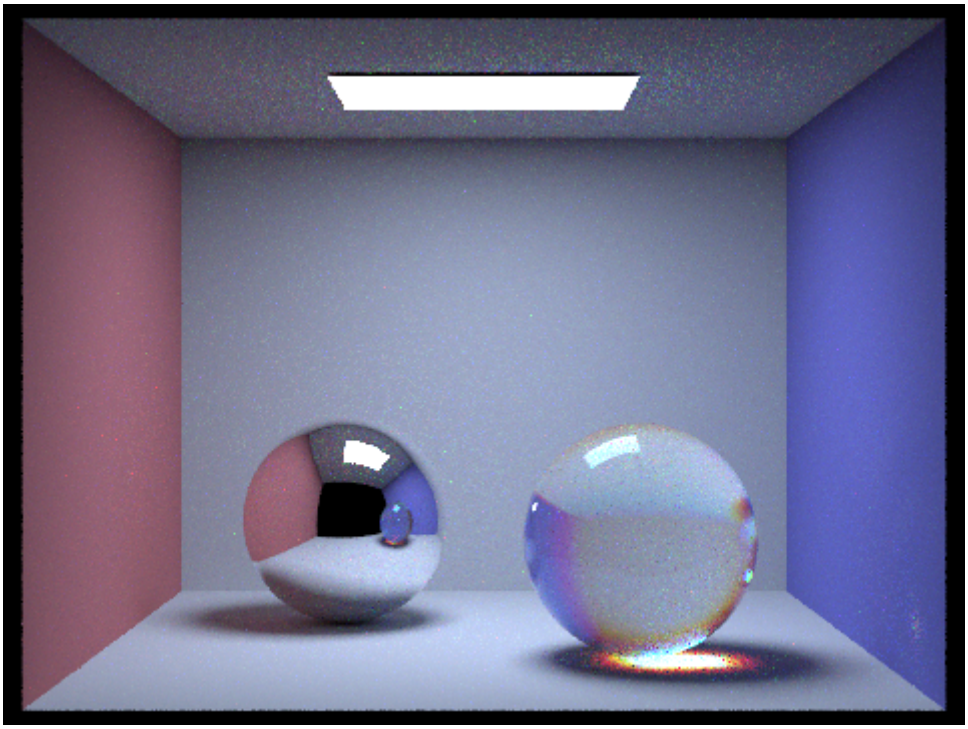
2000K



4000K



10000K



40000K

3. Wavelength dependent BSDF

3.1 Design and implementation

To have a wavelength dependent BSDF, we first implemented a helper function `BSDF::wavelength_dependent_BSDF(double wavelength, Vector3D reflectance)` in `advanced_bsdf.cpp`.

The idea of the helper function is listing as below:

$$\text{albedo}_\lambda = \frac{1}{N}(\text{albedo}_{\text{red}}\frac{1}{\lambda - \mu_{\text{red}}} + \text{albedo}_{\text{green}}\frac{1}{\lambda - \mu_{\text{green}}} + \text{albedo}_{\text{blue}}\frac{1}{\lambda - \mu_{\text{blue}}})$$

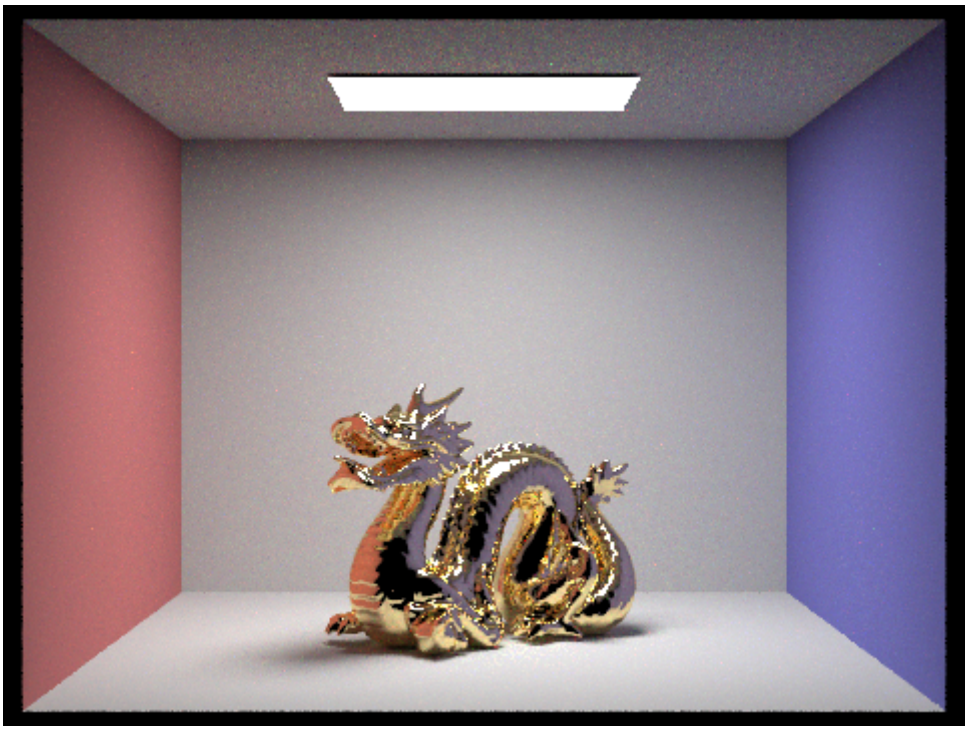
```
Vector3D BSDF::wavelength_dependent_BSDF(double wavelength, Vector3D reflectance) {
    double red_cof = 1.0 / (wavelength - 600);
    double green_cof = 1.0 / (wavelength - 550);
    double blue_cof = 1.0 / (wavelength - 450);
    double N = red_cof + green_cof + blue_cof;
    double albedo = (red_cof * reflectance[0] + green_cof * reflectance[1] + blue_cof * reflectance[2]) / N;
    return albedo;
}
```

Then invoke the helper function to compute result in `BSDF::f()`.

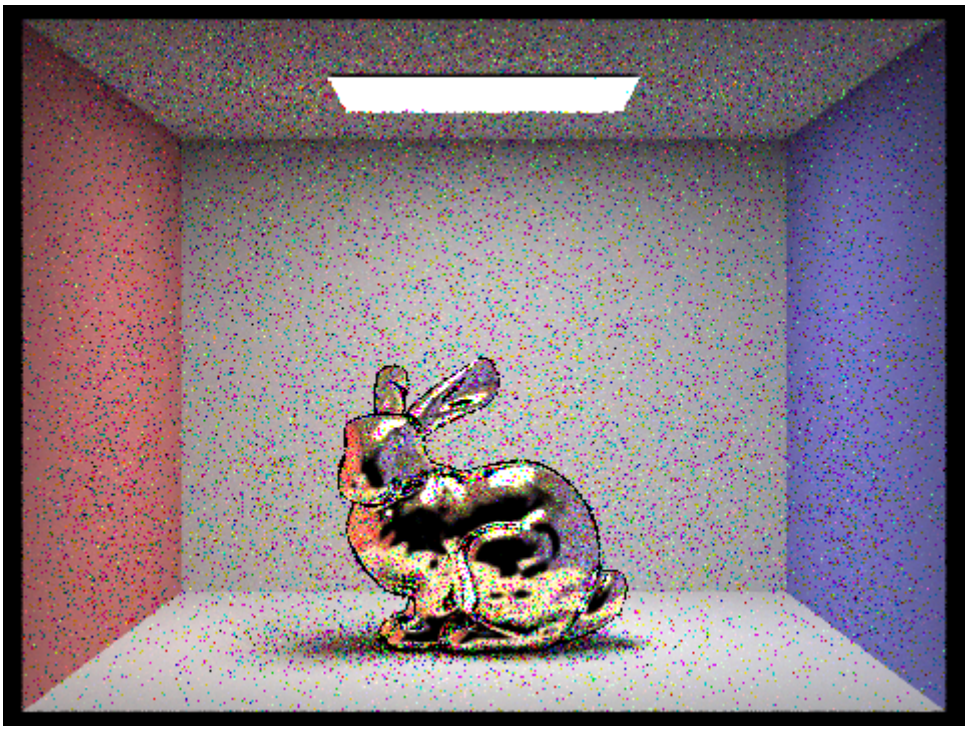
```
Vector3D MicrofacetBSDF::f(const Vector3D wo, const Vector3D wi, double wavelength) {
    Vector3D ans(0.0, 0.0, 0.0);
    if (wo.z >= 0) {
        Vector3D h = ((wo + wi) / 2).unit();
        Vector3D up = F(wi) * G(wo, wi) * D(h);
        double down = 4 * dot(Vector3D(0, 0, 1.0), wo) * dot(Vector3D(0, 0, 1.0), wi);
        ans = up / down;
        auto f = wavelength_dependent_BSDF(wavelength, ans);
        ans = f * wavelength / 300;
    }
    return ans;
}
```

3.2 Sample results

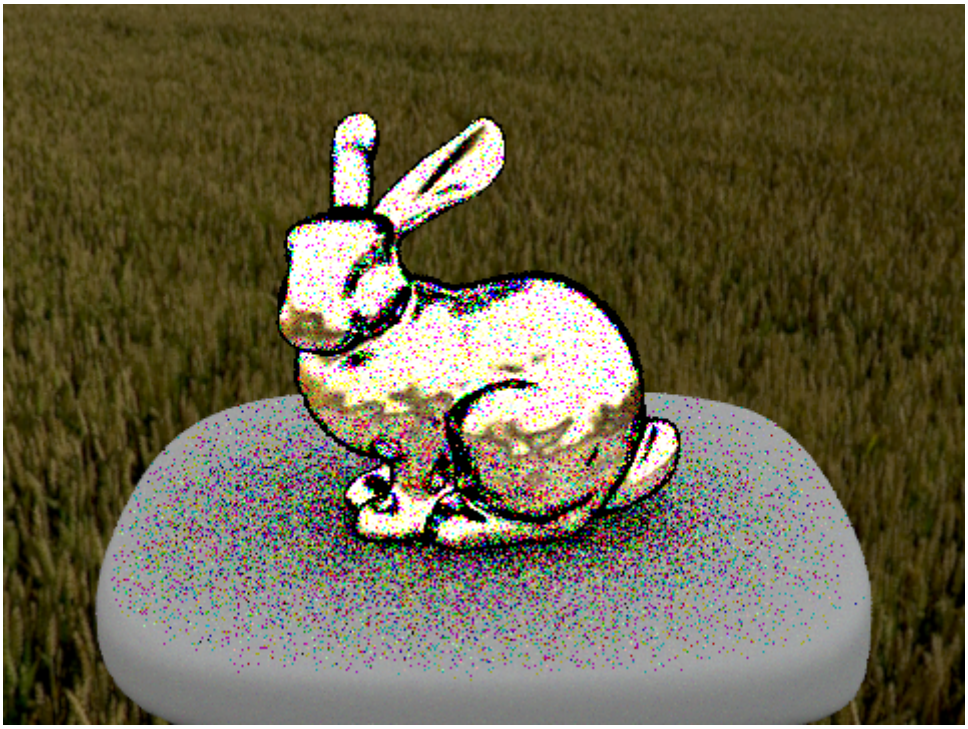
A normal `CBdragon` is rendered with reflectance.



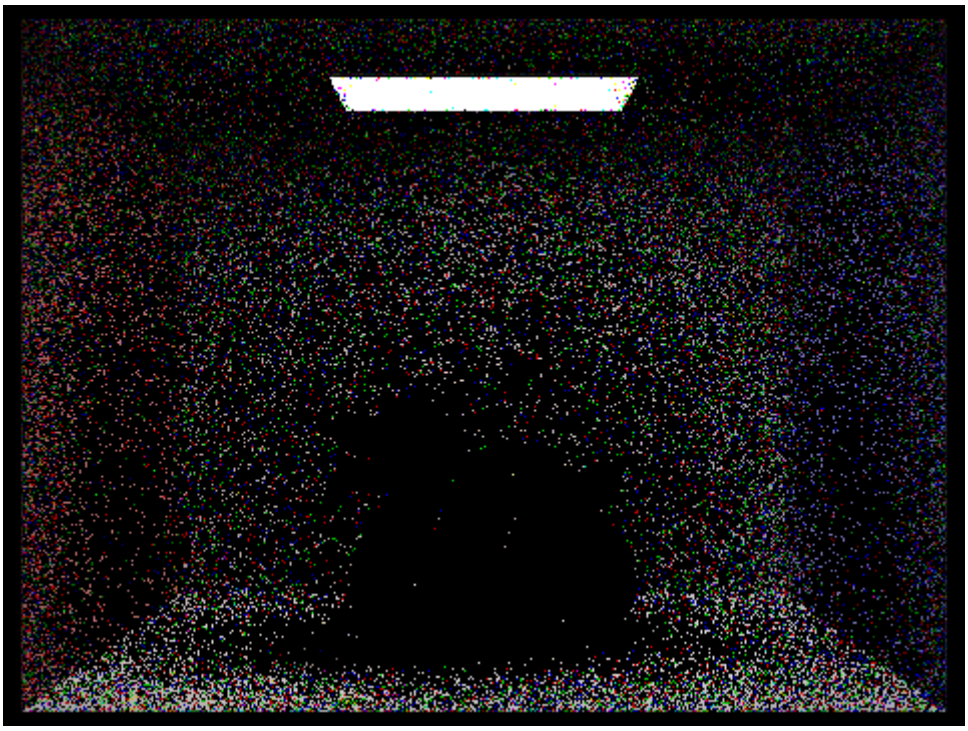
For microfacet objects, `bunny_cu` can be rendered with a black contour.



For field environmental light, it still can be rendered.



However, it would become complete black during rendering `CBdragon_au`. I think it may result from the different facet material properties. It is worth further investigation in the future.



Summary

From this project, I got a better understanding of ray tracing. During the stage of topic selection, I searched for a lot of information like bidirectional path tracing and spectral ray tracing and have been overwhelmed by the ingenuity of researchers in Computer Graphics.

Also, programming process gave me a precious chance to temper myself. Solving those bugs and errors in the template is really painful. Fortunately, I made it through.

To implement my project, I've read several academic papers to extend my knowledge and learned a lot from them. This would be much useful for me in my future research career.

Acknowledgement

This is the last work in this term for me. It is a long and fruitful term indeed. Thanks for all work done by professor and TAs, so I can explore a tip of the iceberg of Computer Graphics. I also need to thank my buddies taken this course as well. They are persons I probably could not have done it without! This experience of studying Computer Graphics is pretty valuable to me, and it is hard to forget the professor's warm and patient office hour meeting. That's all! Thank you all of you!