Natural caustics in backward path tracing

Shawn Halayka*

Friday 6th October, 2023 18:03

Abstract

In this paper we introduce a natural method for producing both refraction and reflection caustics in backward path tracing (e.g. bouncing around an eye ray in the scene until a light is found). These caustics do not rely on a light location, and as such, do not rely on bidirectional or forward path tracing. As such, these caustics allow for as many light sources as one would care for; the shapes and positions of the lights are completely arbitrary (can be bunny shaped, etc). We use the standard Cornell box for testing the backward path tracer.

1 Rasterizer versus ray tracer versus backward path tracer

The three main visualization algorithms in contemporary graphics programming are the rasterizer [1, 2], ray tracing [3, 4], and path tracing [5, 6].

The rasterizer literally converts vector graphics (generally: triangles, lines, and points, all collectively known as simplices) into raster graphics (pixels, also known as fragments). Generally, a depth buffer (Z-buffer) [7] is used to discern which fragments for each simplex to draw, or which to discard, depending on distance to the eye and also what fragments that have been previously drawn. This depth testing algorithm is super simple, and it does the job, but it is just simply not as programmable as a ray tracer or backward path tracer.

The ray tracer does a similar job, insomuch that it converts triangles into pixels. Rather than using a depth buffer though, one often uses a more complicated acceleration structure, such a bounding volume hierarchy (BVH) [8], to determine eye ray / triangle intersection.

The backward path tracer is identical to the ray tracer, except that it also takes global illumination into account. The backward path tracer often uses the exact same acceleration structure setup as the ray tracer. It's worth noting that the construction of the acceleration structure is handled for you by the Vulkan / Direct3D 12 driver, which adds to the naturalness of the caustics discussed in this paper.

Here we rate the three main visualization algorithms. We rate them in terms of what is relatively easy, or naturally occurring, in each of the visualization algorithms.

*sjhalayka@gmail.com

Algorithm	Reflections	Shadows	Refraction caustics	Reflection caustics
Rasterizer	No	No	No	No
Ray tracer	Yes	Sharp	No	No
Backward path tracer	Yes	Sharp / soft	Yes	Yes

Clearly, the most suitable visualization algorithm for now (and in the future) is the backward path tracer. This is not to say that, for instance in the case of the rasterizer, there is no such thing as shadows. It's just that the implementation of shadows is a kludgy affair, or not naturally occurring.

Although caustics can be produced using bidirectional or forward path tracing, they suffer from convergence (or lack thereof) problems. These issues do not plague the backward path tracer.

2 The old special backward path tracing algorithm

Each eye ray is sent out s times, each referred to as a sample.

Essentially, each eye ray sample is bounced around pseudorandomly as it hits surfaces within the scene. The bounce count is finite, and bouncing stops after n bounces, or if a light is hit, or if the sky is hit. At each bounce, one takes into account the properties of the surface that is responsible for the bouncing, such as colour.

In the end, the eye ray's final colour is divided by the sample count.

This works great for diffuse, opaque surfaces.

If there is a convergence problem with regard to the backward path tracer's output, it's because there are not enough samples being taken. Samples take time.

3 The new general backward path tracing algorithm

Other than diffuse, opaque surfaces, there are reflective and transparent surfaces to consider. One can generalize the old algorithm by adding in code to handle reflective and transparent surfaces.

Everything is the same, except that each bounce accounts for the extended properties such as transparency and reflectivity.

4 On photon mapping

Comparing this backward path tracer to photon mapping [9,10] is like comparing apples and oranges. For one, path tracing has a future – that is to say that, on an NVIDIA 3060 laptop GPU, the backward path tracer is still at best only an offline renderer. In essence, photon mapping is a kludge when compared to the ease and naturalness of the caustics, etc. in the backward path tracer. It should be noted that photon mapping also suffers from its own inherent convergence problems. These issues do not plague the backward path tracer.

5 Acknowledgement

The code for this paper is based off of Sascha Willems' and NVIDIA's work [11–13].

The various Cornell boxes were developed by Rob Rau.

The code and various Cornell boxes can be found at:
`https://github.com/sjhalayka/cornell_box_textured`

References

[1] Noll. Scanned-display computer graphics.

[2] Wikipedia. Rasterisation.

[3] Appel. Some techniques for shading machine renderings of solids.

[4] Wikipedia. Ray tracing.

[5] Kajiya. The rendering equation.

[6] Wikipedia. Path tracing.

[7] Wikipedia. Depth buffering (Z-buffering).

[8] Wikipedia. Bounding volume hierarchy.

[9] Jensen. Realistic image synthesis using photon mapping.

[10] John. Focus on photon mapping.

[11] Willems. Vulkan path tracer.

[12] Willems. Vulkan examples and demos.

[13] NVIDIA. Vulkan mini path tracer.

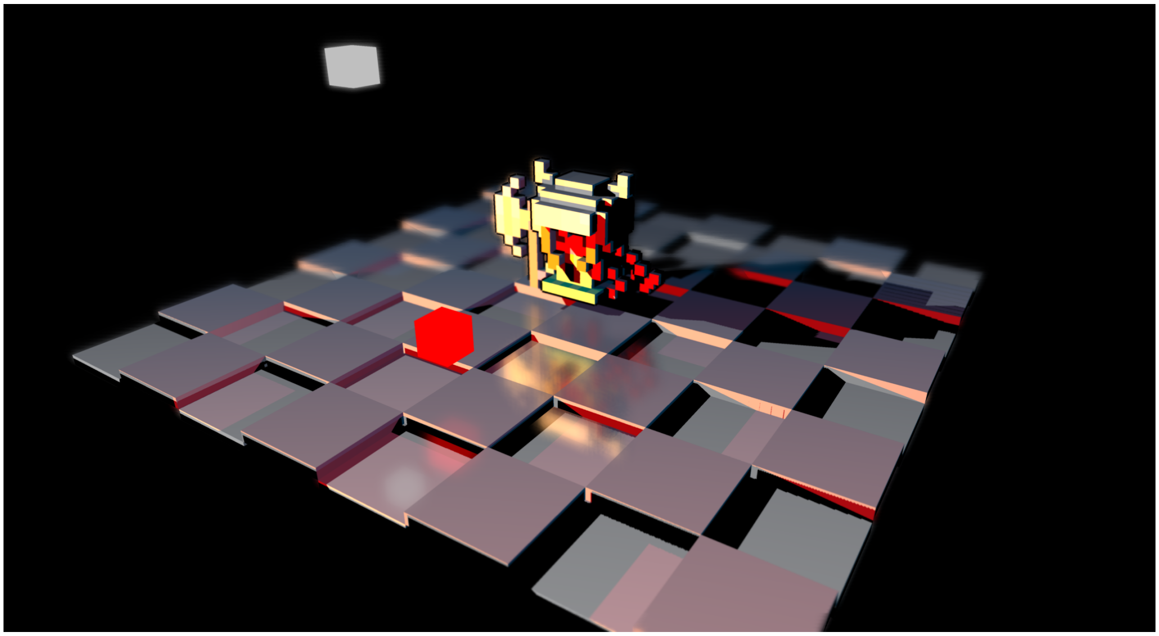

Figure 1: Rasterizer. The surface is lit using Phong shading and omnidirectional shadow maps. The reflections are faked – the camera is flipped upside down, and reflected things masked and drawn. In other words, reflections and shadows are not naturally occurring. Global illumination is not taken into account. This knight model was made by the Twitter user @ephtracy.

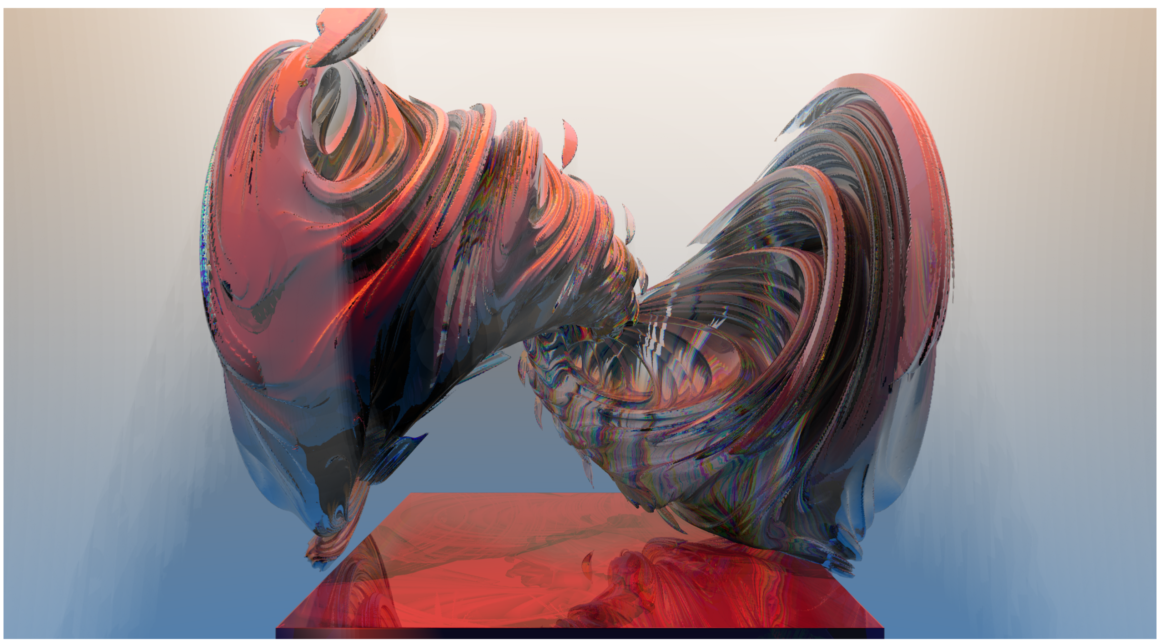

Figure 2: Ray tracer, taking into account transparency. The surface is lit using Phong shading and sharp shadows. In other words, reflections and shadows are naturally occurring. Global illumination is not taken into account.

```
o = hitPos + rayPayload.normal * 0.01;
d = cosWeightedRandomHemisphereDirection(rayPayload.normal, prng_state);
```

Figure 3: Old backward path tracer code, always using a pseudorandom cosine-weighted direction vector (e.g. scattering). Here the o variable is the ray origin, and d is the ray direction. Works great for opaque diffuse surfaces.

```
vec3 o_reflect = hitPos + rayPayload.normal * 0.01;
vec3 d_reflect = reflect(d, rayPayload.normal);

vec3 temp_o = o_reflect;
vec3 temp_d = d_reflect;

if(rayPayload.reflector < 1.0) // if less than fully reflective, do scattering
{
    vec3 o_scatter = hitPos + rayPayload.normal * 0.01;
    vec3 d_scatter = cosWeightedRandomHemisphereDirection(rayPayload.normal, prng_state);

    temp_o = mix(temp_o, o_scatter, 1.0 - rayPayload.reflector);
    temp_d = mix(temp_d, d_scatter, 1.0 - rayPayload.reflector);
}

if(rayPayload.opacity < 1.0) // if partially transparent, do refraction
{
    vec3 o_transparent = vec3(0.0);
    vec3 d_transparent = vec3(0.0);

    // Incoming
    if(dot(d, rayPayload.normal) <= 0.0)
    {
        o_transparent = hitPos.xyz - rayPayload.normal * 0.01f;
        d_transparent = refract(d, rayPayload.normal, eta);
    }
    else // Outgoing
    {
        vec3 temp_dir = refract(d, -rayPayload.normal, 1.0 / eta);

        if(temp_dir != vec3(0.0))
        {
            o_transparent = hitPos.xyz + rayPayload.normal * 0.01f;
            d_transparent = temp_dir;
        }
        else
        {
            // total internal reflection
            o_transparent = hitPos.xyz - rayPayload.normal * 0.01f;
            d_transparent = reflect(d, -rayPayload.normal);
        }
    }

    temp_o = mix(temp_o, o_transparent, 1.0 - rayPayload.opacity);
    temp_d = mix(temp_d, d_transparent, 1.0 - rayPayload.opacity);
}

o = temp_o;
d = normalize(temp_d);
```

Figure 4: New backward path tracer code, taking reflective and transparent surfaces into consideration. This solution for caustics performs well on modern graphics provessing units – disabling caustics causes a practically imperceptible gain in frame rate. While not as simple as the old backward path tracer code, it is still not very computationally expensive – just the right amount of complexity leads to much better results.

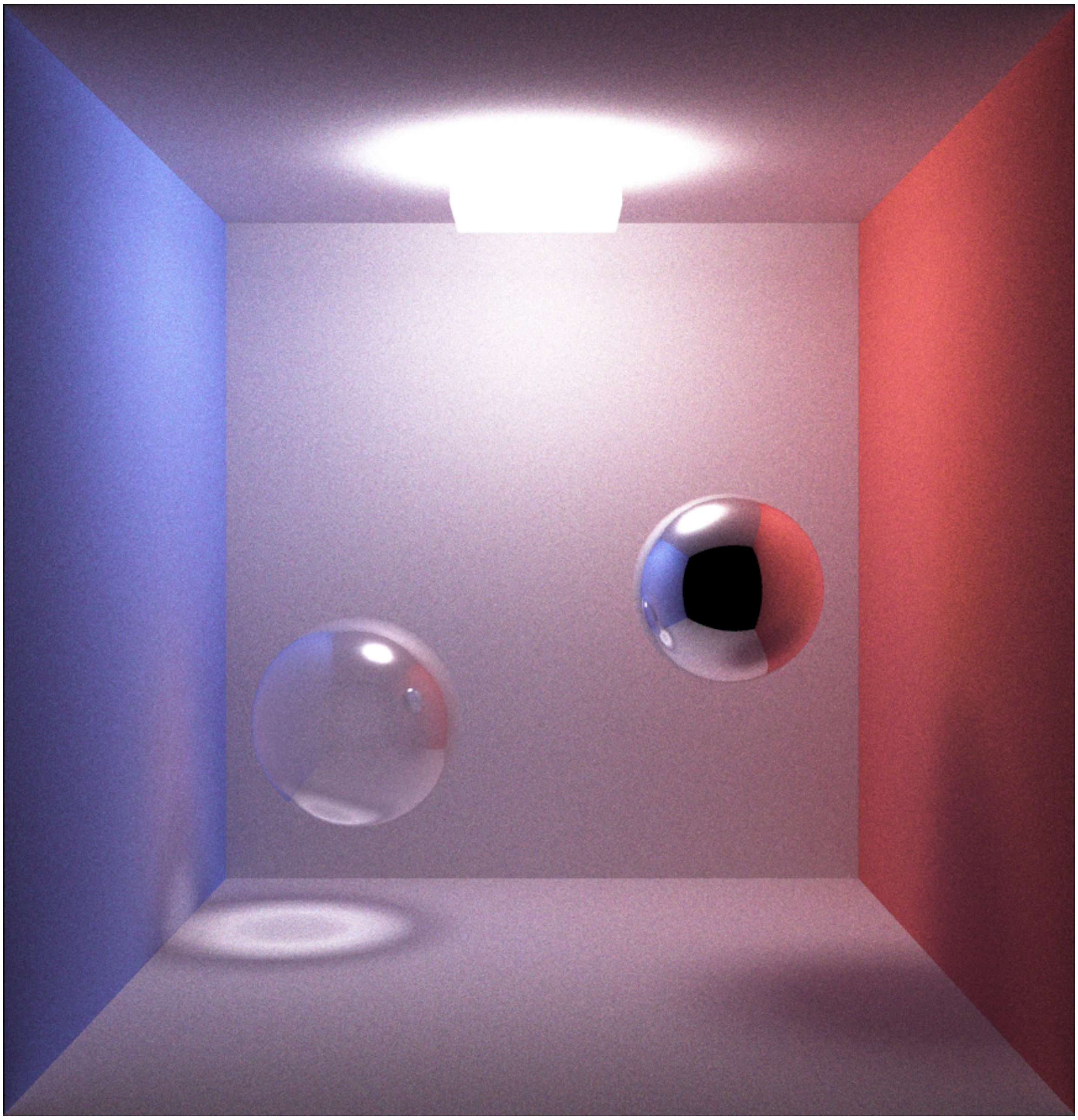

Figure 5: Backward path tracer, taking transparent surfaces into consideration. Note the naturally occurring refraction caustic. Note the soft shadows. Global illumination (e.g. colour bleeding / indirect lighting) is taken into account.

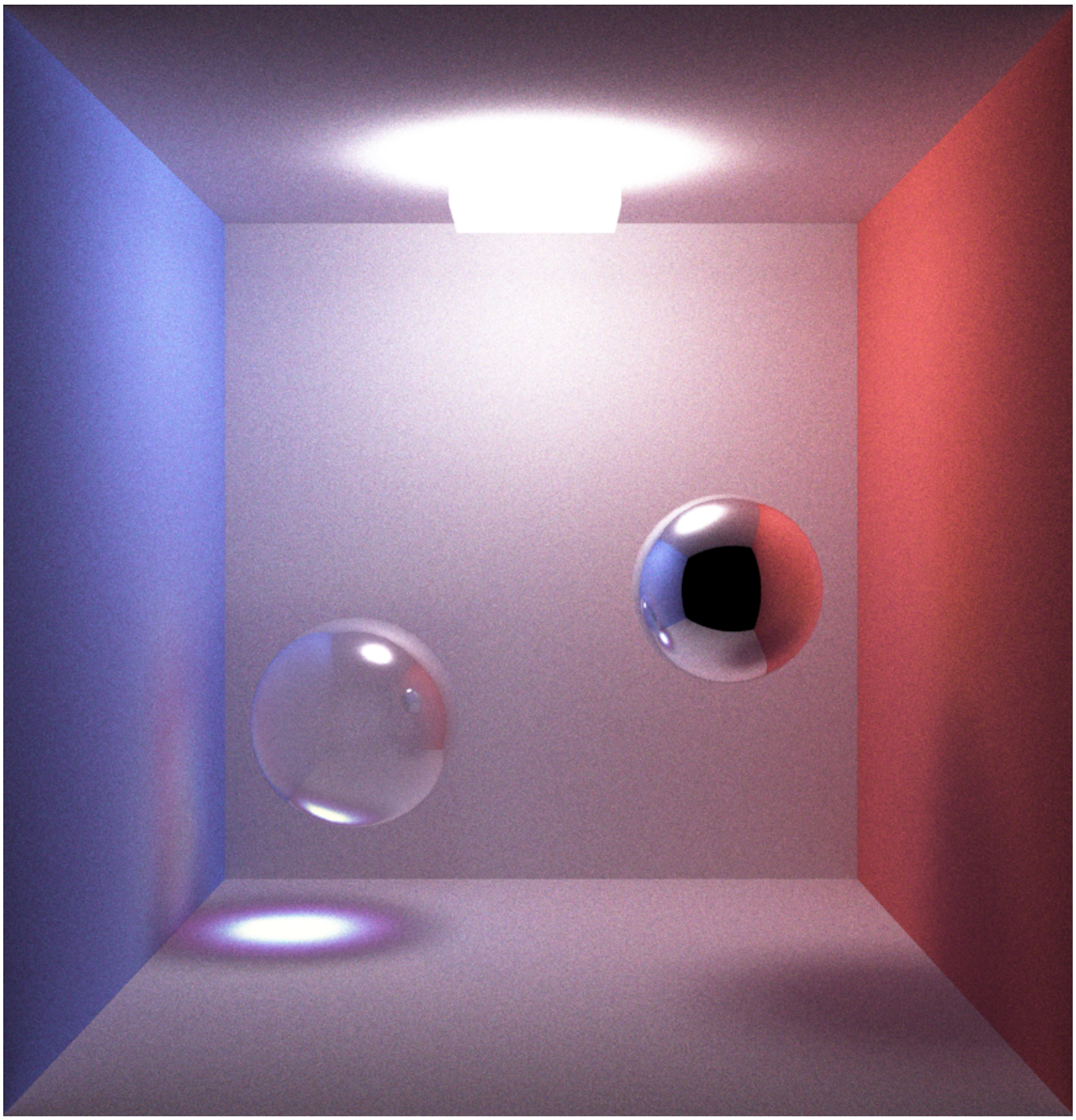

Figure 6: Note the refraction caustic, with 20-channel chromatic aberration.

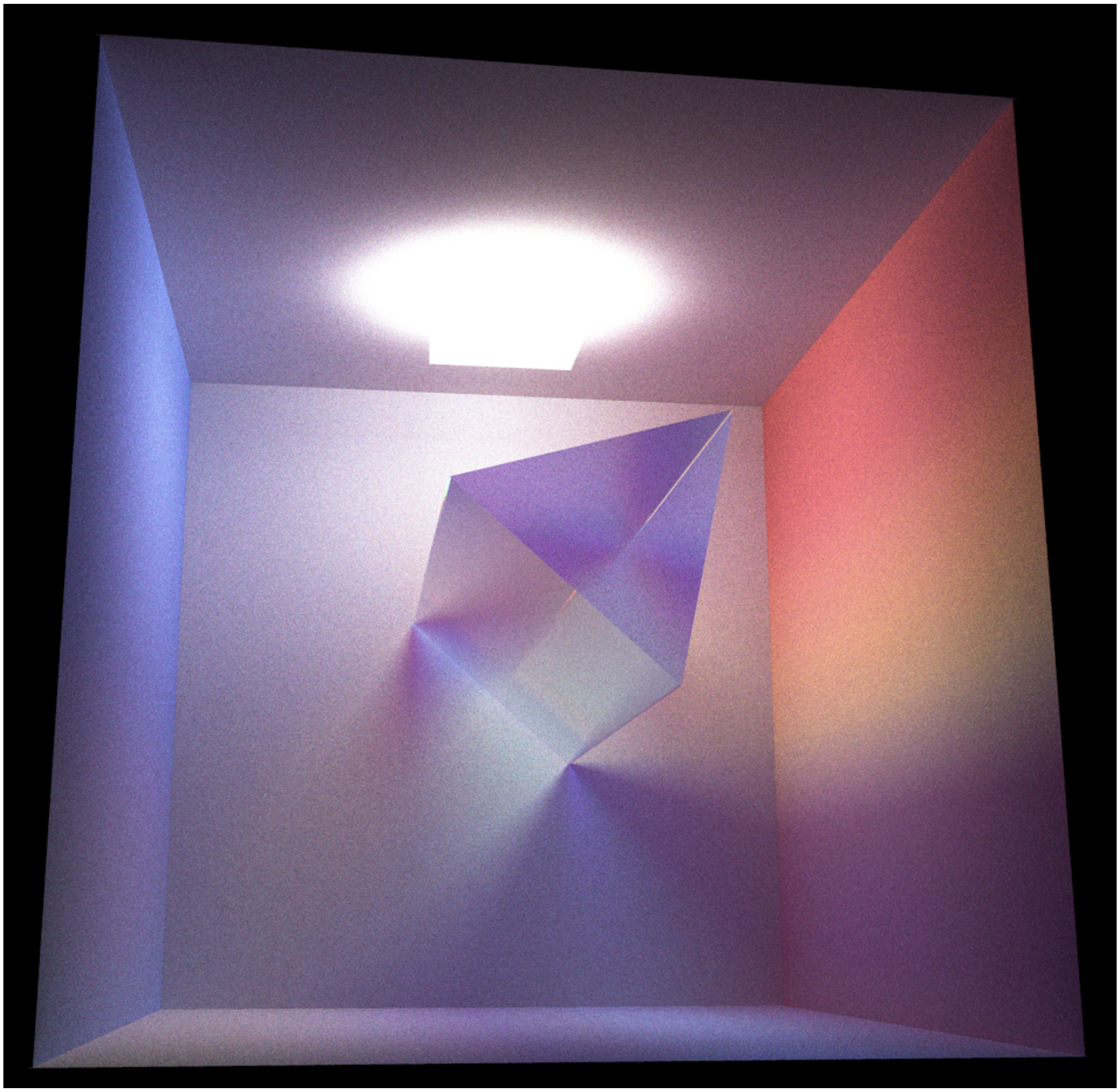

Figure 7: Note the refraction caustic, with 20-channel chromatic aberration.

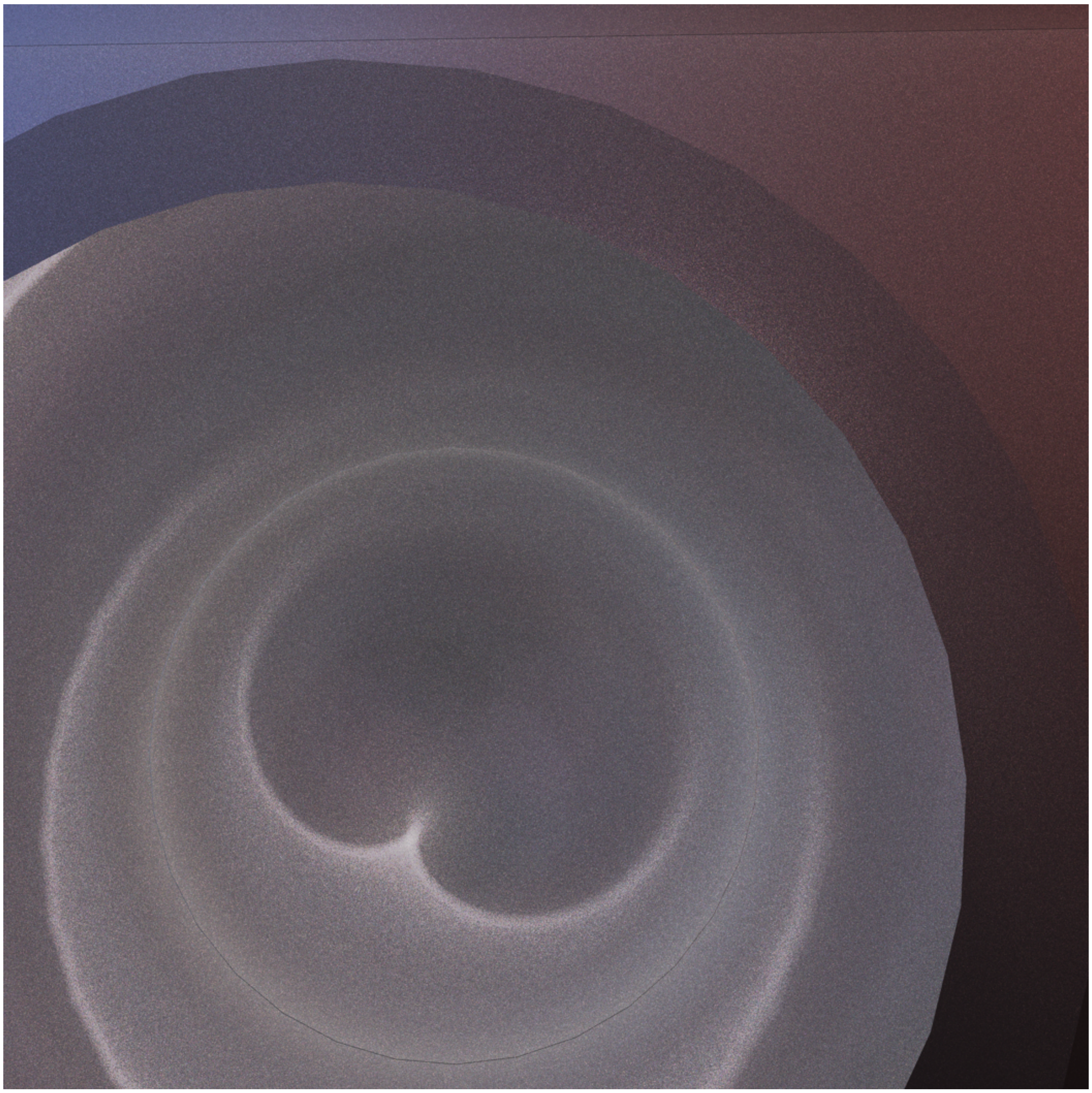

Figure 8: Note the nephroid reflection caustic at the bottom of a hollow cylinder.

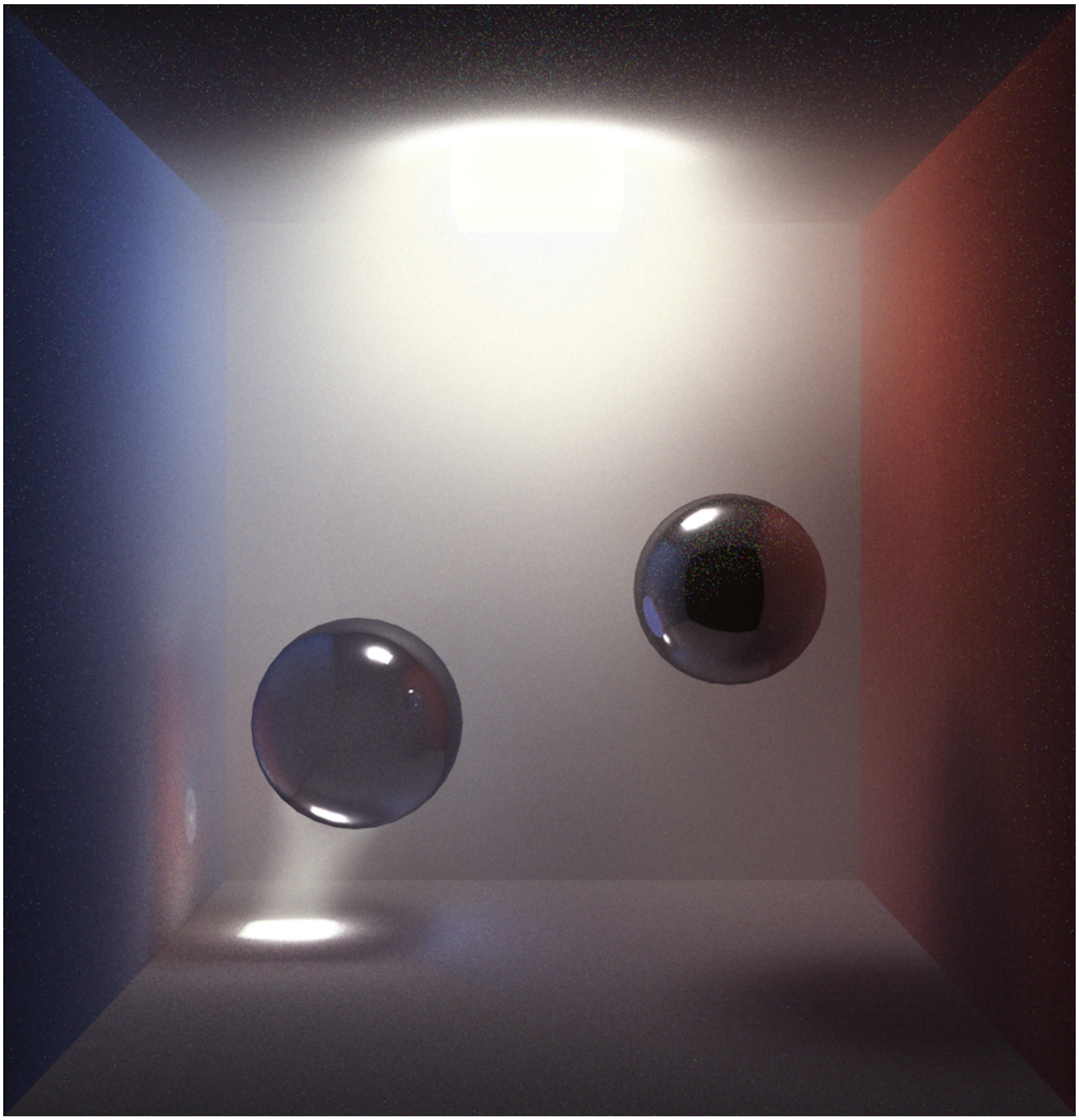

Figure 9: Note that the experimental fog lights up because of the caustic.

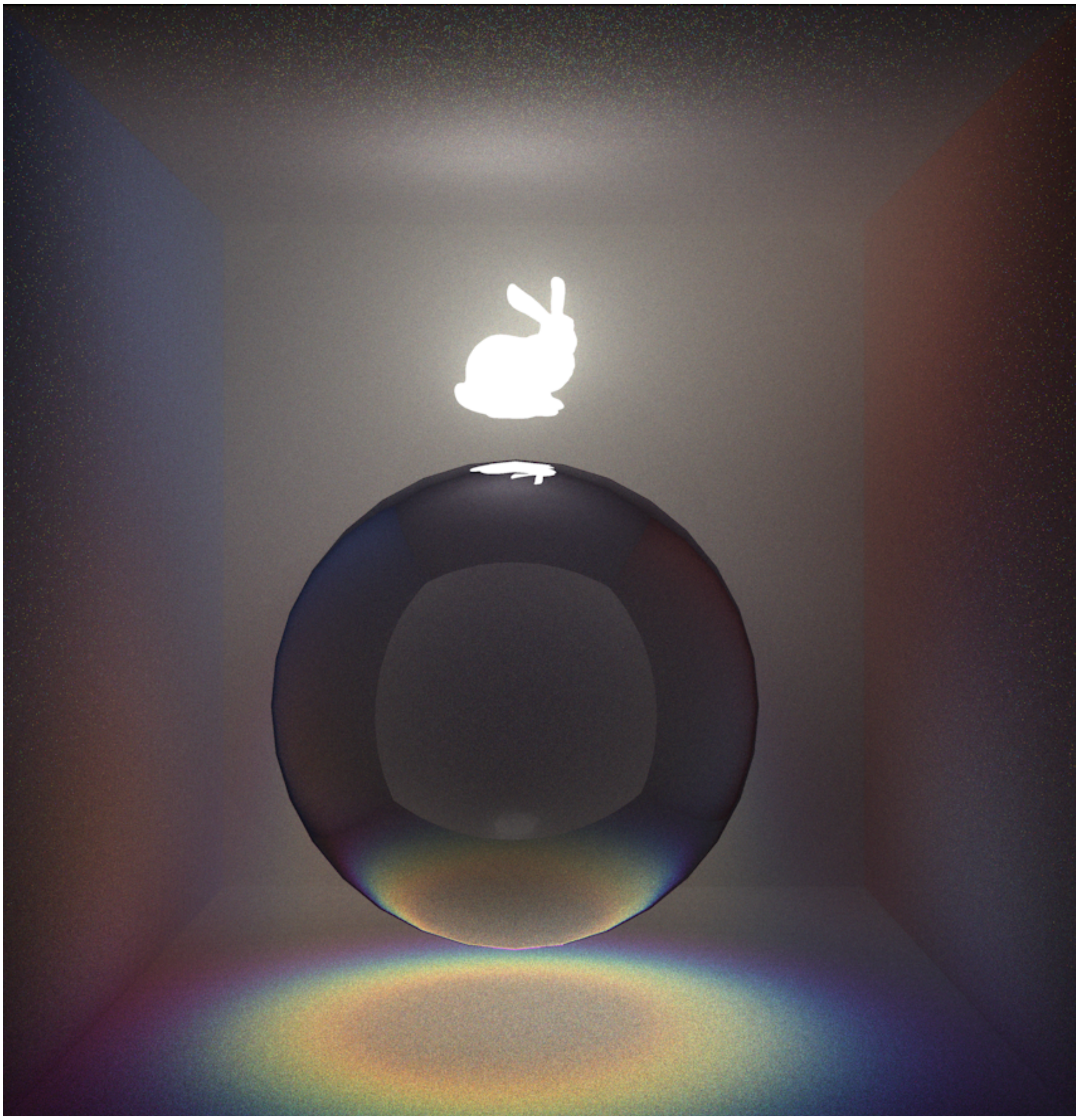

Figure 10: Stanford bunny light.