Natural caustics in backward path tracing

Shawn Halayka*

Wednesday 6th September, 2023 14:59

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, ray tracing, and path tracing.

The rasterizer literally converts vector graphics (generally, triangles) into raster graphics (pixels). Generally, a depth buffer is used to discern which triangles to draw, and which to discard, depending on distance to the eye. This depth testing algorithm does the job, but it is just simply not as programmable as a ray tracer or path tracer.

The ray tracer does a similar job, insomuch that it converts triangles into pixels. Rather than using a depth buffer though, one generally uses a bounding volume hierarchy to determine eye ray / triangle intersection.

The path tracer is identical to the ray tracer, except that it also takes global illumination into account. The path tracer uses the exact same bounding volume hierarchy setup as the ray tracer. As we will soon see, caustics occur naturally in the case of the backward path tracer in particular.

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.

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

*sjhalayka@gmail.com

Clearly, the most suitable visualization algorithm for now (and in the future) is the backward path tracer. This 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 Path tracer code

A full Vulkan code exists [2], using a very simple (e.g. not physically accurate) light transport algorithm. The results given here can only get better with the improvement of the light transport algorithm. This code is based off of Sascha Willems' work [3, 4].

3 Acknowledgement

The various Cornell boxes used in this paper were developed by Rob Rau.

References

[1] Kajiya. The rendering equation.

[2] Halayka. Vulkan code. `https://github.com/sjhalayka/cornell_box_textured`

[3] Willems. Path tracer code. `https://github.com/SaschaWillems/VulkanPathTracer`

[4] Willems. Vulkan demo codes. `https://github.com/SaschaWillems/Vulkan`

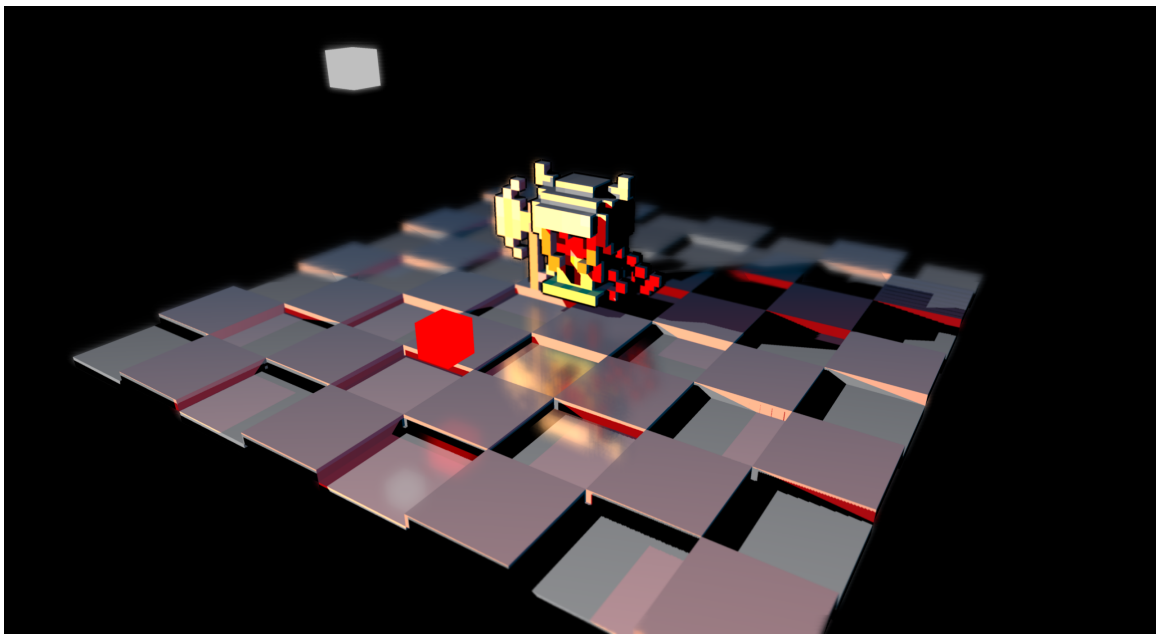

Figure 1: Rasterizer. The surface is lit using Phong shading and omnidirectional shadow maps. The reflections are faked – the camera is flipped upside down, and reflected things masked and drawn. In other words, reflections and shadows are not naturally occurring. Global illumination is not taken into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

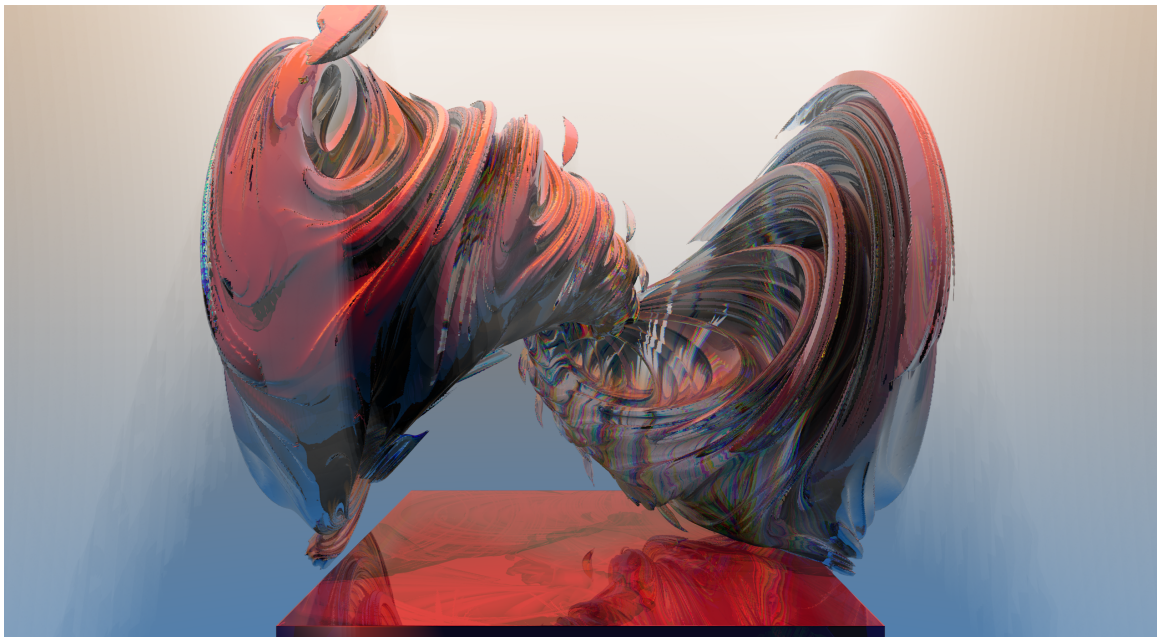

Figure 2: Ray tracer, taking into account transparency. The surface is lit using Phong shading and sharp shadows. In other words, reflections and shadows are naturally occurring. Global illumination is not taken into account.

```
vec3 hitPos = o + d * rayPayload.distance;

// If partially transparent
if(rayPayload.opacity != 1.0)
{
    // Incoming ray
    if(dot(d, rayPayload.normal) <= 0.0)
    {
        o = hitPos.xyz - rayPayload.normal * 0.01;
        d = refract(d, rayPayload.normal, eta);
    }
    else // Outgoing ray
    {
        vec3 temp_dir = refract(d, -rayPayload.normal, 1.0/eta);

        if(temp_dir != vec3(0.0))
        {
            o = hitPos.xyz + rayPayload.normal * 0.01;
            d = temp_dir;
        }
        else
        {
            // Total internal reflection
            o = hitPos.xyz - rayPayload.normal * 0.01;
            d = reflect(d, -rayPayload.normal);
        }
    }
}
else // Fully opaque
{
    o = hitPos + rayPayload.normal*0.01;
    vec3 d_cos = cos_weighted(rayPayload.normal, prng_state);
    d = mix(d_cos, reflect(d, rayPayload.normal), rayPayload.reflector);
}
```

Figure 3: Taking transparent objects into consideration. In essence, instead of always producing a pseudorandom cosine-weighted reflection vector, refraction occurs for transparent objects. Note that reflection caustics are created by reflective surfaces. In the case where the surface is both transparent and reflective, a pseudorandom number should be generated to see which path to take.

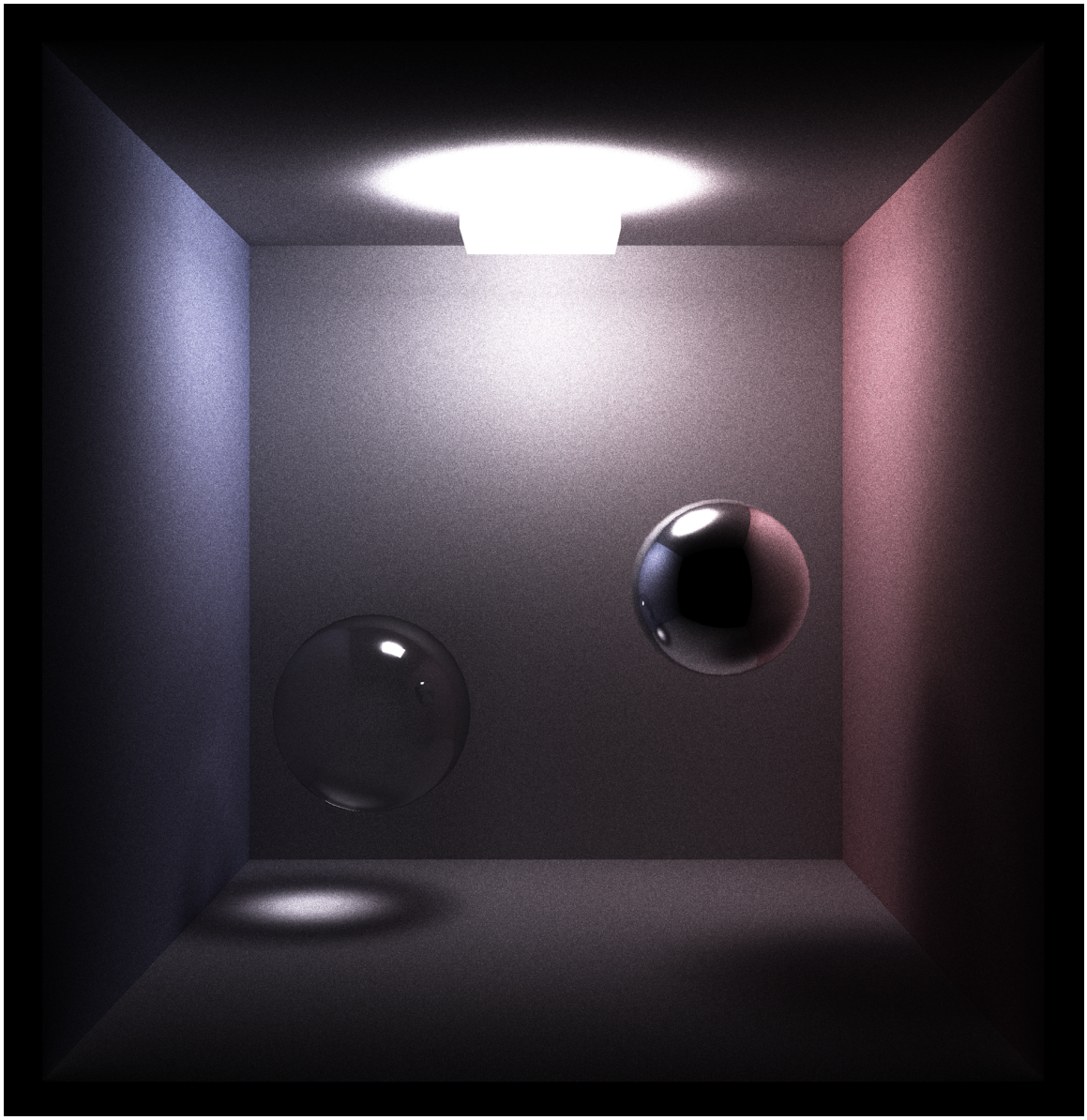

Figure 4: Backward path tracer, taking transparent objects into consideration. Note the naturally occurring refraction caustic. Note the soft shadows. Global illumination (e.g. colour bleeding / indirect lighting) is taken into account.

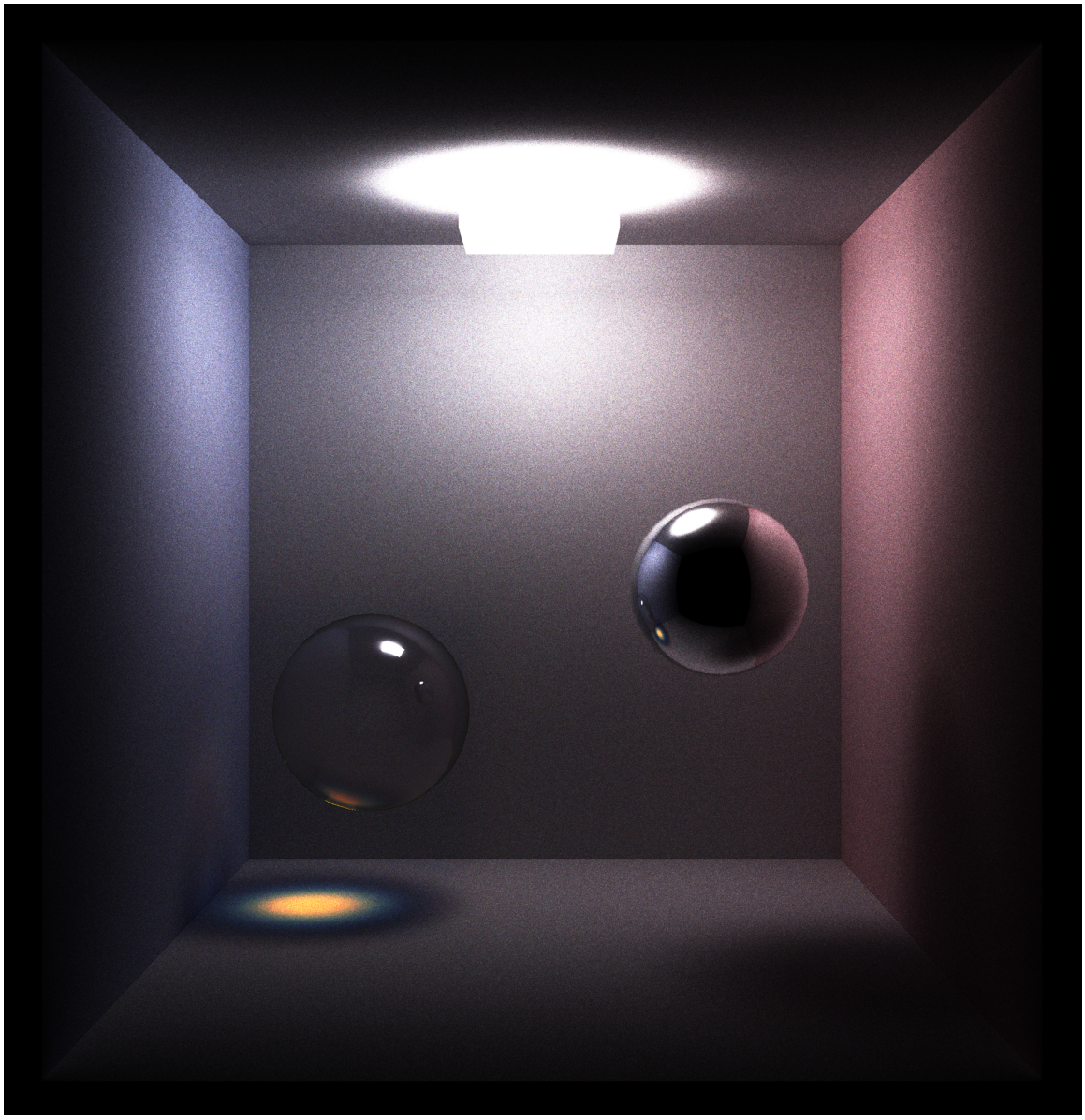

Figure 5: Note the refraction caustic, with 3-channel chromatic aberration.

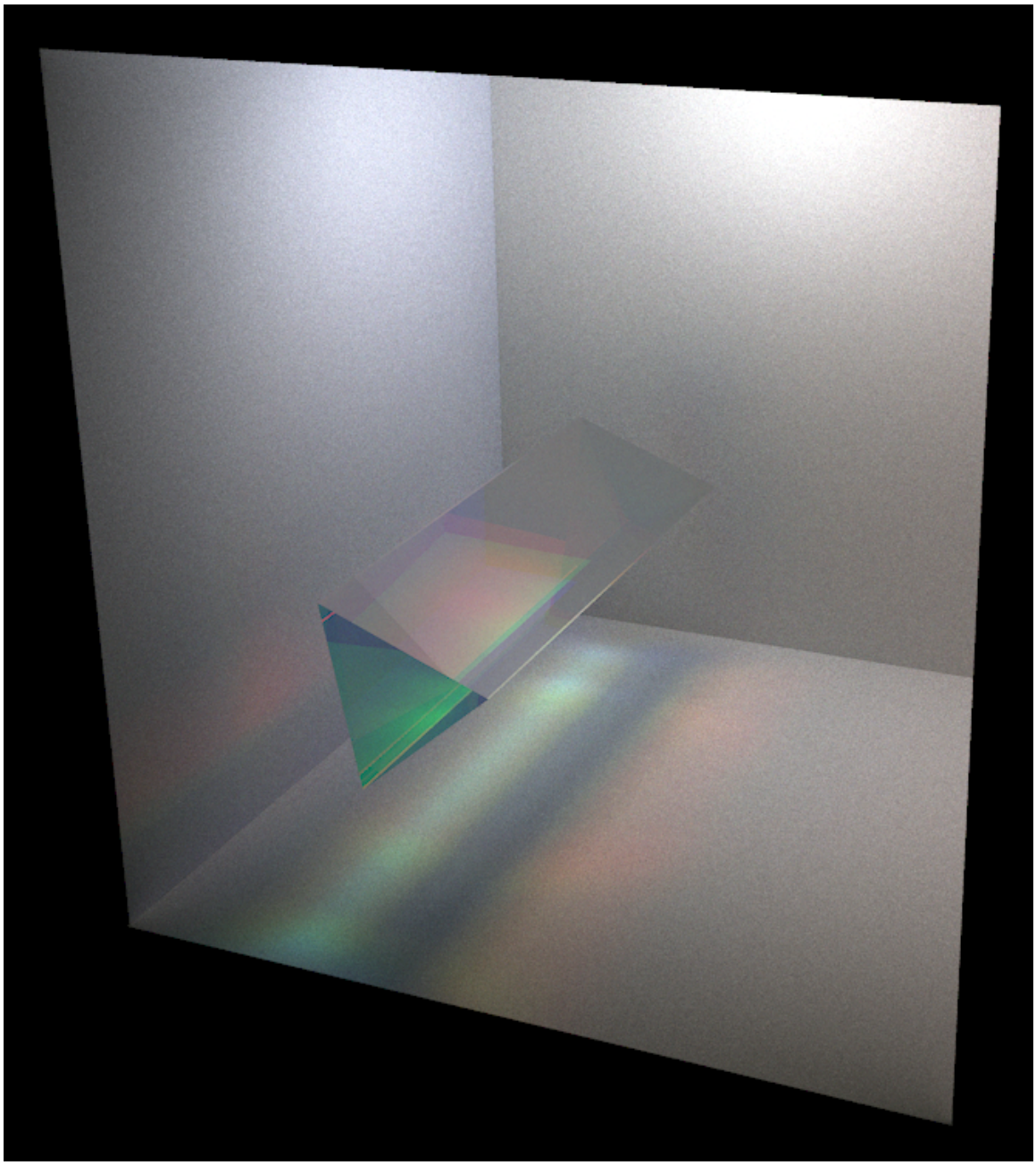

Figure 6: Note the refraction caustic, with 3-channel chromatic aberration.

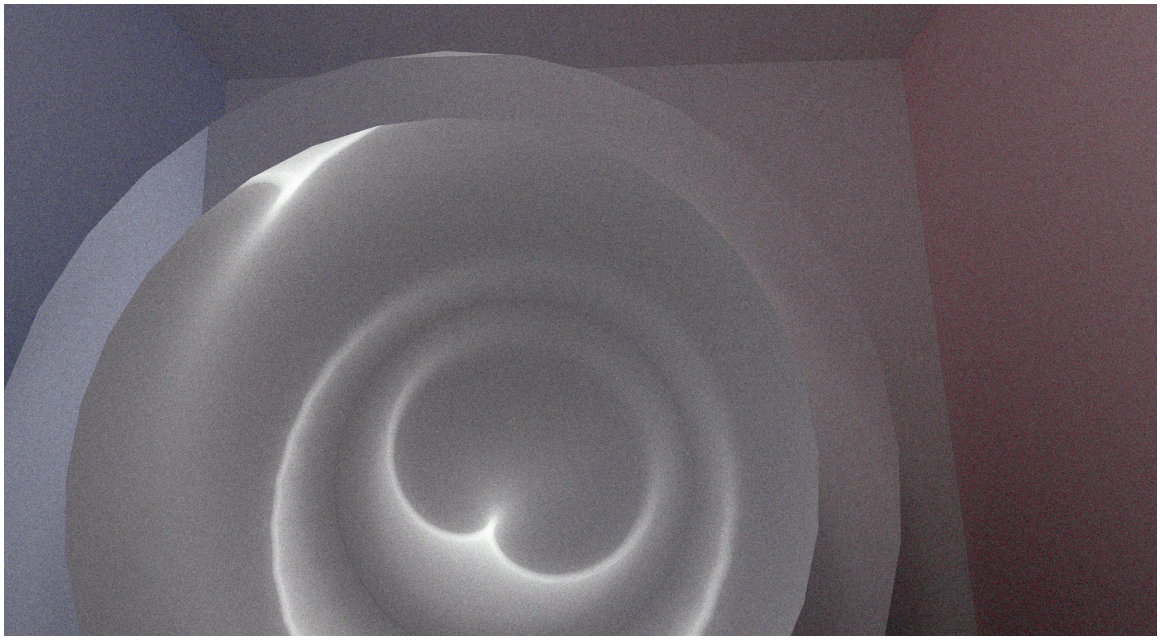

Figure 7: Note the sharp nephroid reflection caustic at the bottom of a hollow cylinder. Note that caustic sharpness is related to the size of the light – smaller light sources produce sharper caustics.