

Natural caustics in backward path tracing

Shawn Halayka*

Friday 13th October, 2023 09:47

Abstract

In this paper we introduce a natural method for producing both refraction and reflection caustics in the 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 Introduction

In this paper we discuss the following:

- Section 2: The difference between the rasterizer, ray tracer, and backward path tracer
- Section 3: Bidirectional or forward path tracers, and convergence problems
- Section 4: The old special backward path tracer algorithm (without caustics)
- Section 5: The new general backward path tracer algorithm (with caustics)
- Section 6: Various imagery produced by the new general backward path tracer
- Section 7: A review of the discussion
- Section 8: Some acknowledgements
- Appendix A: contemporary graphics research on caustics
- Appendix B: classical photon mapping

*sjhalayka@gmail.com

2 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. See Fig. 1 for an example render performed by the rasterizer.

The ray tracer does a similar job, insofar 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. See Fig. 2 for an example render performed by ray tracing.

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 / DXR 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.

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 shadows are not naturally occurring.

3 On bidirectional or forward path tracing

Although caustics can be produced using bidirectional or forward path tracing, thanks to the Helmholtz reciprocity, these caustics often suffer from convergence problems (e.g. low photon count). These particular issues do not plague the backward path tracer.

4 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 scattered (e.g. 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, 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 so-called convergence problem with regard to the backward path tracer's output, it's because there are not enough samples being taken. Samples take time, but they produce beautiful results.

See Fig. 3 for the old code, in GLSL.

5 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 as the old special backward path tracer, except that now each bounce accounts for the extended properties such as reflectivity and transparency. That is, rather than always pseudorandomly scattering the eye ray around, one will also be able to reflect the eye ray if the surface is reflective, and one will refract the eye ray if the surface is transparent.

See Fig. 4 for the new code, in GLSL.

6 Results of the new code

See Figs. 5 - 10 for the results of the new code. Each image is rendered at a size of 7680 x 4204, at $s = 25$ samples per eye ray with maximum $n = 5$ bounces, then cropped and downsized to like 1000 x 1000. The full size images are available upon request.

7 Review

In this paper we have introduced a rendering method that accounts for colour bleeding, indirect lighting, and most importantly, caustics – collectively known as the phenomena of global illumination. This method runs on modern graphics processing units, although 4K output is still produced using offline rendering.

It should be noted that the GLSL code to enable caustics is only around 30 lines longer than the code without caustics – again, see Fig. 3 versus Fig. 4. The C++ codes for the old special backward path tracer and the new general backward path tracer are identical – the C++ code is literally zero lines longer. This low line count is a testament to the naturalness of the solution.

8 Acknowledgements

Many thanks to those who gave me feedback, and to those who replied to my emails. Much appreciated.

The code for this paper is based off of Sascha Willems' and NVIDIA's work [15–17].

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

A Caustics especially: contemporary approaches versus the backward path tracer

We should consider the work of others, in regard to caustics.

See [11–14], which are all completely different modern approaches to solving the problem of caustics. Our paper stands apart from these others because the rendering method that it introduces is 1) simple, 2) fast, and 3) free of the traditional convergence problems. That is, we do not use buffers or maps or all kinds of convoluted programmatic incantations.

In essence, these other approaches are not natural, especially when compared to the ease and naturalness of the caustics in the backward path tracer.

B Global illumination in general: classical photon mapping versus the backward path tracer

Given that photon mapping [9, 10] is the *de facto* standard method for producing global illumination, we must give it some consideration.

Comparing photon mapping to this backward path tracer is like comparing apples and oranges.

For one, photon mapping generally uses a k-d tree for storing photon data, and the backward path tracer does not. It takes time and space to store and retrieve data to and from the photon map.

For another, backward path tracing has a future – this is to say that, on an NVIDIA 3060 laptop graphics processing unit, the backward path tracer is still at best only an offline renderer at 4K resolution. Its day will come.

As well, it must be noted that the backward path tracer is strictly a view space effect. This is unlike photon mapping and its 2-pass solution, insofar that the caustics and other lighting effects are all accomplished in one pass when using the backward path tracer.

Of course, it should be noted that photon mapping also suffers from its own inherent convergence problems (e.g. low photon count). These particular issues do not plague the backward path tracer.

Finally, it is stated in [9] that $s = 1000$ samples per eye ray are necessary to get caustics that are not noisy. Here we only used $s = 25$ samples. This likely means that the path tracer used in [9] was not a backward path tracer – it's no wonder why it could be stated as a matter of fact in [9] that photon mapping is hundreds of times faster than the path tracer!

This is obviously no longer the case, because 1) we use a *bona fide* backward path tracer, 2) we use a sample count s that is orders-of-magnitude smaller than what is used in [9], and 3) the backward path tracer is now accelerated by the graphics processing unit in Vulkan / DXR.

In essence, photon mapping is not natural, especially when compared to the ease and naturalness of the global illumination in the backward path tracer.

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] Moreau, et al. Real-time rendering of indirectly visible caustics.
- [12] Komarov, et al. Rendering the real-time caustics with DirectX raytracing.
- [13] Gruen. Ray tracing gems – Ray-guided volumetric water caustics in single scattering media with DXR.
- [14] Yang, et al. Ray tracing gems II – Real-time ray traced caustics.
- [15] Willems. Vulkan path tracer.
- [16] Willems. Vulkan examples and demos.
- [17] 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 special 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 diffuse, opaque surfaces. The `cosWeightedRandomHemisphereDirection` function returns a pseudorandomly-generated direction vector in the same hemisphere as the surface normal.

```

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(stepAndOutputRNGFloat(prng_state) > rayPayload.opacity)
{
    doing_caustic = true;

    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 = o_transparent;
    temp_d = d_transparent;
}
else
{
    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);
}

o = temp_o;
d = normalize(temp_d);

```

Figure 4: New general backward path tracer code, taking reflective and transparent surfaces into consideration. This solution for caustics performs well on modern graphics processing 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. It’s not perfect, but it’s a good start. The stepAndOutputRNGFloat function returns a pseudorandom float in the range of 0.0 to 1.0.



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. Also note that the smaller the light source, the sharper the caustic. Finally, it should be noted that the backward path tracer fails when the light source has zero volume. The solution to this problem is to model one's lights according to reality – having non-zero volume – and so to abandon the simplification used in the shell theorem as applied to point lights.

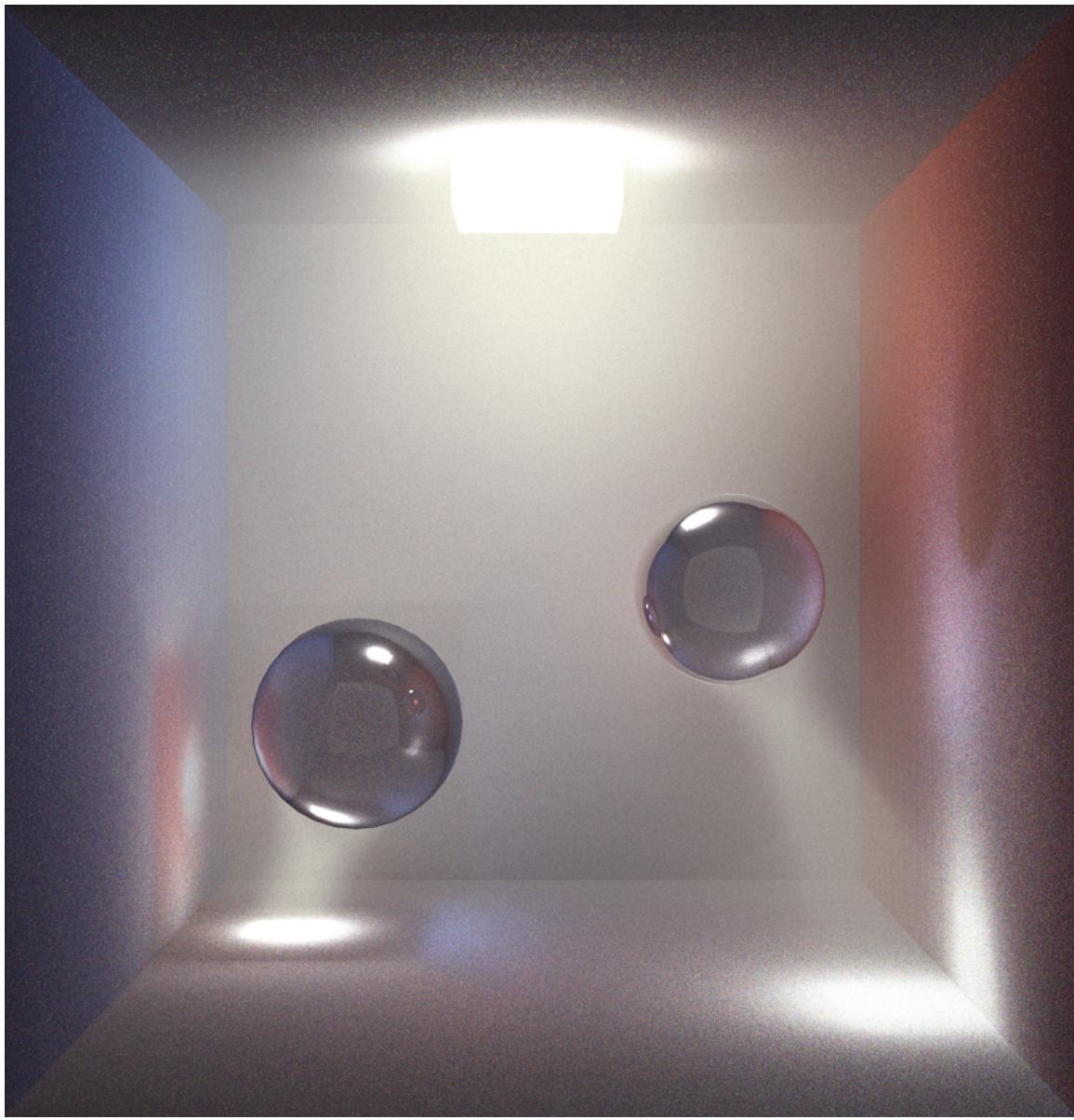


Figure 9: Here, both spheres are transparent. Note that the fog lights up because of the caustics. Note all kinds of caustics – red, blue, white. Note no chromatic aberration.



Figure 10: Stanford bunny light, with 20-channel chromatic aberration.