CHAPTER 30

Caustics Using Screen-Space Photon Mapping

Hyuk Kim
devCAT Studio, NEXON Korea

ABSTRACT

Photon mapping is a global illumination technique for rendering caustics and indirect lighting by simulating the transportation of photons emitted from the light. This chapter introduces a technique to render caustics with photon mapping in screen space with hardware ray tracing and a screen-space denoiser in real time.

30.1 INTRODUCTION

Photon mapping is a novel global illumination technique invented by Henrik Wann Jensen [2]. It uses a photon to simulate light transportation to obtain global illumination and concentrated light images such as caustics. While it is useful for rendering caustics, traditional photon mapping has not been practical for real-time games. It requires many photons to obtain smooth images. This means that significant ray tracing is required.

McGuire and Luebke have developed *image space photon mapping* (ISPM) [4] in real time. ISPM stores a photon as a volume in the world. Because there are far fewer of these photons than real-world photons, they must be spread (scattered) in some way. In this chapter, a photon is stored as a texel in screen space instead of a photon volume or a surfel in world space. Although this approach, which I call *screen-space photon mapping* (SSPM), still has a few limitations, there are some advantages, such as caustics.

Caustics are a result of intense light. If a photon passes through two media with the different indices of refraction, e.g., from air to glass or from air to water, the photon is refracted and its direction of propagation changes. Refracted photons can be either scattered or concentrated. Such concentrated photons generate caustics. Alternatively, reflections can make caustics, too. Reflected photons can also be concentrated by surrounding objects. Examples of caustics are shown in Figure 30-1; in the top right image, the yellow caustics are generated from reflections off the ring.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_30

Figure 30-1. *Caustics generated by screen-space photon mapping, with 2k × 2k photons for each scene: from top left to bottom right, Conference Room, Ring and Bunny on Conference Room (Ring & Bunny), Bistro, and Cornell Box. Performance measures are shown later in Table 30-2.*

Note that, in this chapter, screen-space photon mapping is used purely for caustics, not for global illumination of the whole scene. For obtaining global illumination, other optimized general-purpose "large" photon-gathering techniques [1, 3] might provide a better solution.

30.2 OVERVIEW

Photon mapping is usually performed in two stages: photon map generation and rendering. I have divided it into three:

- Photon emission and photon tracing (scattering).
- Photon gathering (denoising).
- Lighting with the photon map.

The first stage is *photon emission and photon tracing*, described in detail in Section 30.3.1. Each ray in the DirectX Raytracing (DXR) ray generation shader corresponds to a single photon. When a single photon is traced from a light source to an opaque surface, the photon gets stored in screen space. Note that the emission and tracing of a photon are *not* screen-space operations. With DXR, ray tracing is performed in world space instead of screen space, as done for screen-space reflections. After ray tracing, the photon is stored in screen space. The render target texture storing these photons then becomes a screen-space photon map.

Since a photon is stored as a texel in the screen-space texture, the screen-space photon map has noisy caustics (such as shown in the left part of Figure 30-4). For that reason, the second stage, called *photon gathering* or *photon denoising*, is required to smoothen the photon map, which will be described in Section 30.3.2. After the photons are gathered, we obtain a smooth photon map. Finally, the photon map is used within the direct lighting process, described in Section 30.3.3.

Note that I assume a deferred rendering system. The G-buffer for deferred rendering, including a depth buffer or roughness buffer, is used for storing and gathering photons in screen space.

30.3 IMPLEMENTATION

30.3.1 PHOTON EMISSION AND PHOTON TRACING

The symbols used in this section are summarized in Table 30-1.

Table 30-1. *Summary of symbols.*

Symbol	Quantity	Equation
Φ_e	Radiant flux of the light	30.1, 30.3
l_e	Radiance of the light	30.2
p_w, p_h	Size of width and height of the photons (rays)	
w, h	Width and height of the screen	
a_l	The area of the light area (for a directional light)	
a_p	The area of the pixel	30.4
l_p	Radiance stored to a pixel	30.5

30.3.1.1 PHOTON EMISSION

A photon is emitted and traced in world space. The emitted photon has color, intensity, and direction. A photon that finally stops in the world has only color and intensity. Since more than one photon can be stored in a single pixel, the incoming direction cannot be preserved. Photon *flux* (color and intensity) is stored in a pixel without a direction.

For a point light, photons are emitted from the position of the light. Equation 1 shows the emission flux Φ_e of a photon has the intensity of a light ray,

$$\Phi_e = \frac{l_e}{4\pi} \frac{1}{p_w p_h}, \tag{1}$$

where p_w and p_h are the sizes of photon rays and l_e is the radiance of the light from the light source:

$$l_e = \text{light color} \times \text{light intensity}. \tag{2}$$

Equations for a directional light are similar. Unlike a point light, photons of a directional light spread to the whole scene without attenuation. Since each photon corresponds to multiple ray traces, reducing wastage of photons is important for both quality and performance. To reduce such wastage, Jensen [2] used projection maps to concentrate photons on significant areas. A *projection map* is a map of the geometry seen from the light source, and it contains information on whether the geometry exists for a particular direction from the light.

For simplicity and efficiency, I used a bounding box called a *projection volume* for a directional light. Its purpose is the same as the cell in Jensen's projection maps. A projection volume is a bounding box in which there exist objects generating caustics, as shown by volume V in Figure 30-2. By projecting the box to the negative direction of the directional light, we can obtain a rectangular light area shown as area A. If we emit photons only to the projection volume, photons can be concentrated on objects generating caustics for the directional light. Moreover, we can control the number of rays to obtain either consistent quality or consistent performance, i.e., constant photon emission area or constant ray tracing count.

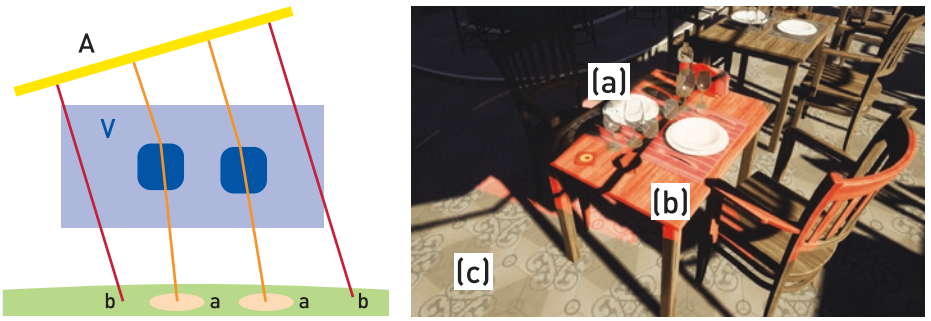

Figure 30-2. *Left: Projection volume V and corresponding light area A. Right: the projection volume is placed onto the table encompassing transparent objects. (a) Rays that generate caustics tracing through transparent objects are marked in orange. (b) Photons corresponding to direct light are marked in red. These photons are discarded. (c) Outside of the projection volume, no rays will be emitted.*

For the resolution of light area A created with projection map V, $p_w p_h$ is the size of the ray generation shader passed to dispatched rays such as a point light. Each ray of the ray generation shader carries a photon emitted from area A. Since each photon corresponds to a portion of the light area, each photon has area $1/(p_w p_h)$ for light area a_l. The emission flux of a directional light is

$$\Phi_e^{\mathrm{D}} = l_e \frac{a_l}{p_w p_h}. \tag{3}$$

However, it should be noted that a_l is the area of light area A in world-space units.

30.3.1.2 PHOTON TRACING

After a photon is emitted, the photon is ray traced in the world until the maximum number of ray tracing steps is reached or a surface hit by the photon is opaque. In more detail, after a ray traces and hits some object, it evaluates material information. If the surface hit by the ray is opaque enough, the photon will stop and the scene depth will be evaluated to check whether the surface would store the photon in screen space or not. If no object is hit, or the photon's scene depth from the camera is beyond that stored in the depth buffer, the photon will be discarded. Another condition for stopping the photon is when photon intensity is negligibly small, i.e., the photon is diminished while passing through transparent objects.

The red rays in Figure 30-2 correspond to direct light and will not be stored. It is redundant to store a photon from direct lighting since we process direct light in a separate pass. Removing all direct light might create some artifacts around shadow edges in the denoising stages, but these are not very noticeable.

Because tracing a ray through transparent objects is not a special part in photon mapping, I have skipped those details.

In short, there are four conditions under which a photon will not be stored:

1. The photon's intensity is negligibly small.
2. The location to be stored is out of the screen.
3. The photon travels beyond the stored depth buffer values.
4. The photon is part of direct lighting.

The code for generating photons is in the function `rayGenMain` of PhotonEmission.rt.hlsl.

30.3.1.3 STORING PHOTONS

A significant part of the screen-space photon mapping process is that a photon is compressed into a single pixel. This might be wrong for a given pixel, but energy will be conserved as a whole. Instead of a photon spreading out (scattering) into neighboring pixels, the compressed photon is stored in a single pixel. After being stored, the pixel's photon will be scattered onto its neighbors (i.e., it gathers from its neighbors) in the following denoise stage.

The area a_p of a pixel in world-space units is

$$a_p = \left(\frac{2d\tan\left(\theta_x / 2\right)}{w} \right) \left(\frac{2d\tan\left(\theta_y / 2\right)}{h} \right), \tag{4}$$

where w and h are the width and height, respectively, of the screen; θ_x and θ_y are the field-of-view x and y angles, respectively, of the field of view; and d is the distance from the eye to the pixel in world space.

Since Φ_e is not radiance, but flux, a photon must be turned into proper radiance. The radiance l_p to be stored in a pixel is

$$l_p = \frac{\Phi_e}{a_p} = \left(4 \cdot 4\pi \tan\left(\frac{\theta_x}{2}\right) \tan\left(\frac{\theta_y}{2}\right) \right)^{-1} \frac{l_e}{d^2} \left(\frac{wh}{p_w p_h} \right). \tag{5}$$

Note that one of the advantages of screen-space photon mapping is that the photon can use the eye vector when being stored. This means that the photon can have specular color as well as diffuse color evaluated from the BRDFs of the surfaces.

Figure 30-3 shows comparisons on different numbers of the photons before denoising. The code for storing photon implementation is in the function `storePhoton` of PhotonEmission.rt.hlsl.

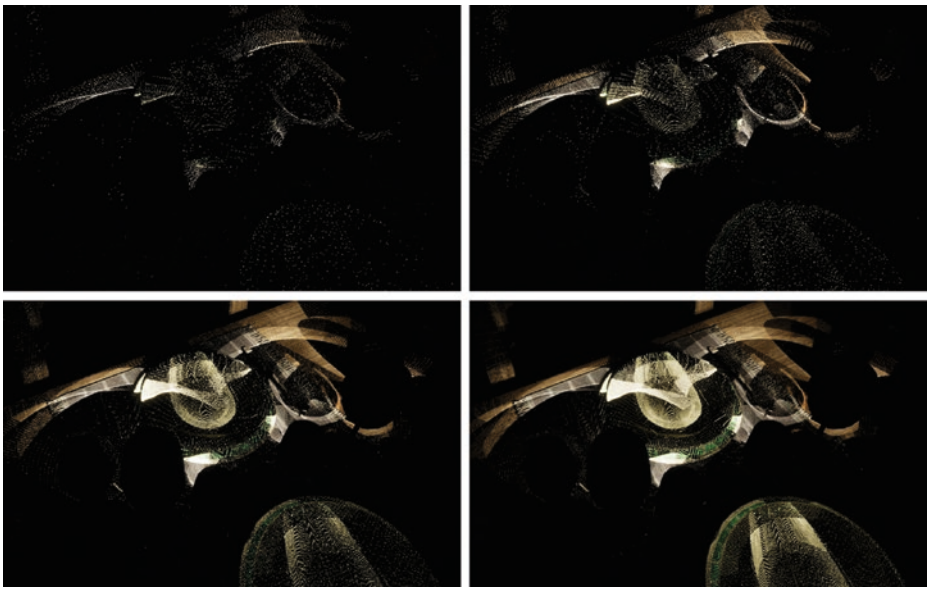

Figure 30-3. *From top left to bottom right, 500 × 500, 1000 × 1000, 2000 × 2000, and 3000 × 3000 photons with directional light in the Bistro scene.*

30.3.2 PHOTON GATHERING

Traditional photon mapping uses several techniques to get an accurate and smooth result. To obtain real-time performance, one of the easiest ways to gather photons in screen space is by using the reflection denoiser from NVIDIA GamesWorks Ray Tracing [5]. With a reflection denoiser, we can think of photons as reflections with some tricks.

The denoiser receives the camera data (matrix), depth buffer, roughness buffer, and normal buffer as inputs. It constructs a world from the camera matrix and depth, then gathers neighbor's reflections based on normal and roughness. Keep in mind that the denoiser receives a buffer containing hit distances from the pixel to the reflection hits.

In a photon denoiser, hit distance becomes the distance from the last hit position to the pixel, fundamentally the same as for reflections. Unlike reflections, however, a photon map does not need to be sharp. As a small hit distance prevents photon maps from blurring, I clamped the distance to a proper value that varies based on the scene.

The normal and roughness are static values for the photon denoiser. On one hand, photons would not gather well if the original normal of an object is used. On the other hand, roughness is a crucial part of denoising reflections and so original values should ideally be retained. After some experimentation, I set the normal as the direction to the camera from the pixel and the roughness as some value around 0.07. These values might change for different scenes or for different versions of the denoiser. I set roughness as a parameter for global blurriness of the scene and adjusted blurriness per photon by hit distance.

See the comparison of before and after denoising in Figure 30-4. Here is a summary of the denoiser parameters for the photon denoiser:

- *Normal:* Direction from the pixel to the camera.
- *Roughness:* Constant parameter; 0.07 worked well for me.
- *Hit distance:* Last traced distance clamped with minimum and maximum values. In my scene, 300 and 2000 were the minimum and maximum values, respectively. I recommend that you make a distance function suitable for your scene.

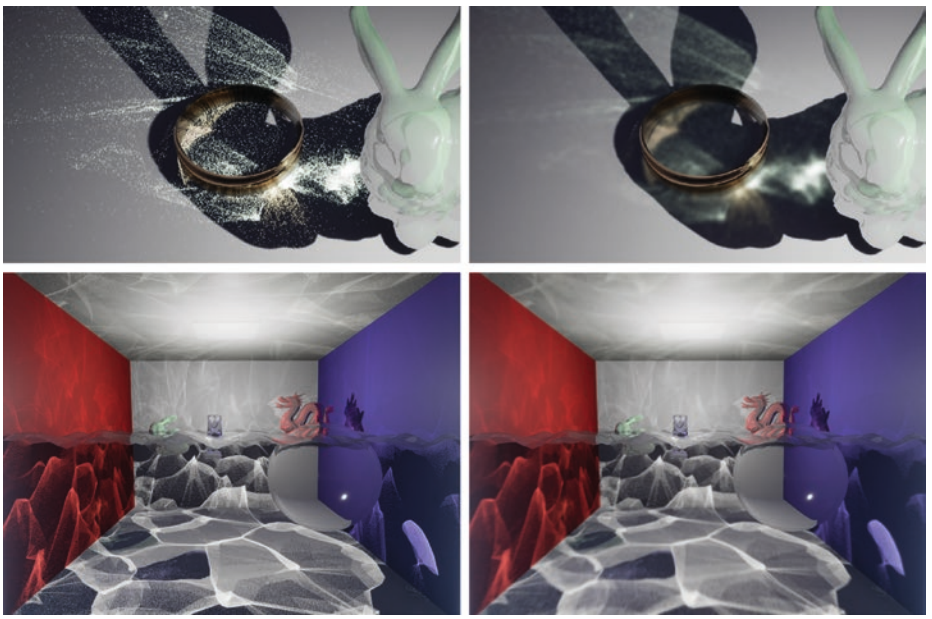

Figure 30-4. *Left: before denoising. Right: after denoising. Top: Ring & Bunny. Bottom: Cornell Box. Both scenes are rendered with 2k × 2k photons from point lights.*

The implementation is in the class `PhotonGather::GameWorksDenoiser` in PhotonGather.cpp.

Note that while using the GameWorks denoiser works quite well, it is only one of the good methods for denoising. Since the GameWorks ray tracing denoiser is bilateral filtering for ray tracing, readers may want to implement a denoiser specifically for photon noise to obtain fine-tuning and efficiency.

In addition to the GameWorks denoiser, a bilateral photon denoiser is also provided in the class `PhotonGather::BilateralDenoiser` in PhotonGather.cpp. The bilateral photon denoiser consists of two parts: downsampling a photon map and denoising a photon map. A downsampled photon map is used for near-depth surfaces (which is not discussed here; see detailed comments in the code). There are also good references on bilateral filtering [3, 6].

30.3.3 LIGHTING

Lighting with a photon map is simple. Photon map lighting is just like screen-space lighting on deferred rendering system. A photon presented in a photon map is considered as an additional light for the pixel. The center of Figure 30-5 shows this photon map only.

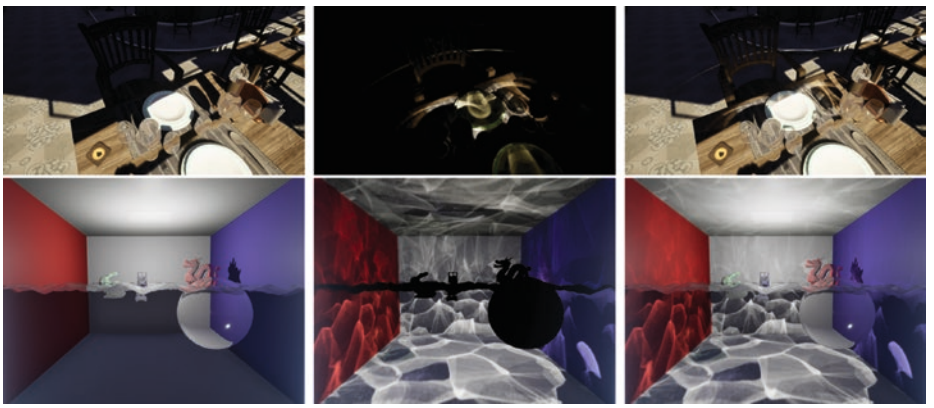

Figure 30-5. *Left: Without SSPM. Center: Photon map only. Right: Composited. Top: Bistro. Bottom: Cornell Box.*

30.4 RESULTS

With the help of Microsoft's DXR and NVIDIA's RTX, practical caustic rendering in real time with some limitations can be achieved. Figure 30-1 shows results of caustics and their performance measures are listed in Table 30-2. All the performance measurements include time spent denoising. The cost of the denoiser is about 3–5 ms. Note that using the reflection denoiser is not an optimized solution for photon denoising.

Table 30-2. *Performance for the scenes in Figure 30-1 on a GeForce RTX 2080 Ti with 1920 × 1080 pixels. All measures are in milliseconds and each number in parentheses is the difference from the cost without SSPM.*

Scene	No SSPM	1k × 1k Photons	2k × 2k Photons	3k × 3k Photons
Conference Room	4.79	9.39 (4.6)	11.94 (7.15)	16.20 (11.41)
Ring & Bunny	4.13	9.24 (5.11)	11.44 (7.31)	15.15 (11.02)
Bistro	10.98	11.04 (<1)	12.27 (1.29)	17.15 (6.17)
CornellBox	4.18	9.20 (5.02)	12.62 (8.44)	18.23 (14.05)

All the figures in this chapter were rendered with Unreal Engine 4; however, the accompanying code is based on NVIDIA's Falcor engine. The timing results for no SSPM versus 1k × 1k SSPM show little difference for the Bistro because this scene has many objects.

30.4.1 LIMITATIONS AND FUTURE WORKS

While screen-space photon mapping is practical in real time, there are some limitations and artifacts produced. First, due to the lack of an atomic operation when writing a pixel to a buffer in the ray tracing shader, there might exist some values being lost when two shader threads write the same pixel simultaneously. Second, because pixels near the screen frustum's near depth are too high resolution for photons, photons do not gather well when the camera approaches the caustic surfaces. This might be improved by using other blurring techniques with a custom denoiser for future works.

30.4.2 TRANSPARENT OBJECTS IN THE DEPTH BUFFER

In this chapter, transparent objects are drawn to the depth buffer, just like opaque objects. The translucency of transparent objects is performed by ray tracing, starting from the surfaces of these objects. While this is not the usual implementation of deferred rendering, it has some pros and cons for caustics. Caustics photons can be stored on transparent objects when they are drawn in the depth buffer. However, we cannot see caustics beyond transparent objects. This limitation can be seen in the Cornell Box scene in Figure 30-1.

30.4.3 PRACTICAL USAGE

As mentioned previously, some of the photons are lost when they are written in a buffer. Besides, the number of photons is far less than what we would need for real-world representations. Some of the photons are blurred out by the denoising process. For practical and artistic purposes, one can add additional intensity to caustics. This is not physically correct but would complement some loss of photons. Note that the figures presented here have not had additional intensity applied in order to show precise results.

There is one more thing to consider. As you know, caustics generated from reflection and refraction should be used under restricted conditions. The current implementation has been chosen to have transparent iterations as the main loop. Rays causing reflection caustics are generated in each transparent loop. This is shown in Section 30.5. If the scene is affected by reflection caustics more than refractions, a reflection loop might be more suitable.

30.5 CODE

The following pseudocode corresponds to photon emission and photon tracing, including storing a photon. Real code can be found in PhotonEmission.rt.hlsl.

```
1 void PhotonTracing(float2 LaunchIndex)
2 {
3     // Initialize rays for a point light.
4     Ray = UniformSampleSphere(LaunchIndex.xy);
5
6     // Ray tracing
7     for (int i = 0; i < MaxIterationCount; i++)
8     {
9         // Result of(reconstructed) surface data being hit by the ray.
10        Result = rtTrace(Ray);
11
```

```
12          bool bHit = Result.HitT >= 0.0;
13          if (!bHit)
14              break;
15
16          // Storing conditions are described in Section 30.3.1.2.
17          if(CheckToStorePhoton(Result))
18          {
19              // Storing a photon is described in Section 30.3.1.3.
20              StorePhoton(Result);
21          }
22
23          // Photon is reflected if the surface has low enough roughness.
24          if(Result.Roughness <= RoughnessThresholdForReflection)
25          {
26              FRayHitInfo Result = rtTrace(Ray)
27              bool bHit = Result.HitT >= 0.0;
28              if (bHit && CheckToStorePhoton(Result))
29                  StorePhoton(Result);
30          }
31          Ray = RefractPhoton(Ray, Result);
32      }
33  }
```

REFERENCES

[1] Jendersie, J., Kuri, D., and Grosch, T. Real-Time Global Illumination Using Precomputed Illuminance Composition with Chrominance Compression. *Journal of Computer Graphics Techniques 5*, 4 (2016), 8–35.

[2] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[3] Mara, M., Luebke, D., and McGuire, M. Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2013), pp. 71–78.

[4] McGuire, M., and Luebke, D. Hardware-Accelerated Global Illumination by Image Space Photon Mapping. In *Proceedings of High-Performance Graphics* (2009), pp. 77–89.

[5] NVIDIA. GameWorks Ray Tracing Overview. https://developer.nvidia.com/gameworks-ray-tracing, 2018.

[6] Weber, M., Milch, M., Myszkowski, K., Dmitriev, K., Rokita, P., and Seidel, H.-P. Spatio-Temporal Photon Density Estimation Using Bilateral Filtering. In *IEEE Computer Graphics International* (2004), pp. 120–127.