CHAPTER 30

REAL-TIME RAY TRACED CAUSTICS

Xueqing Yang and Yaobin Ouyang
NVIDIA

ABSTRACT

We present two real-time ray tracing techniques for rendering caustic effects. One focuses on the caustics around metallic and transparent surfaces after multiple ray bounces, which is built upon an adaptive photon scattering approach and can depict accurate details in the caustic patterns. The other is specialized for the caustics cast by water surfaces after one-bounce reflection or refraction, which is an enhancement of the algorithm of caustics mapping, highly fluidic in sync with the water ripples, and able to cover large scene areas. Both techniques are low cost for high frame rate usages, fully interactive with dynamic surroundings, and ready-to-use with no data formating or preprocessing requirements.

30.1 INTRODUCTION

Caustics are commonly seen phenomenon in scenes containing water, metallic, or transparent surfaces. However, in most of today's real-time renderers, they are either ignored or roughly handled using tricks like decal textures. Although objects casting caustics may only occupy a small portion of the screen in most cases, the delicate optical patterns are very challenging to simulate with a limited time budget. Fortunately, the arrival of GPU ray tracing brings out the possibility of performing photon mapping [6]—the most efficient technique for simulating caustics so far—in real time to accurately rendering these effects.

Noticeably, in the book *Ray Tracing Gems*, Hyuk Kim [7] proposed a simple scheme to execute photon mapping in real time: tracing photons through the scene, blending them directly onto a screen-space buffer, and then applying a spatial denoiser to obtain the final patterns. Albeit easy to implement, the method uses a fixed resolution for photon emission with uniform distribution, which limits its application for large-scale scenes, and the blend-denoise

©NVIDIA 2021

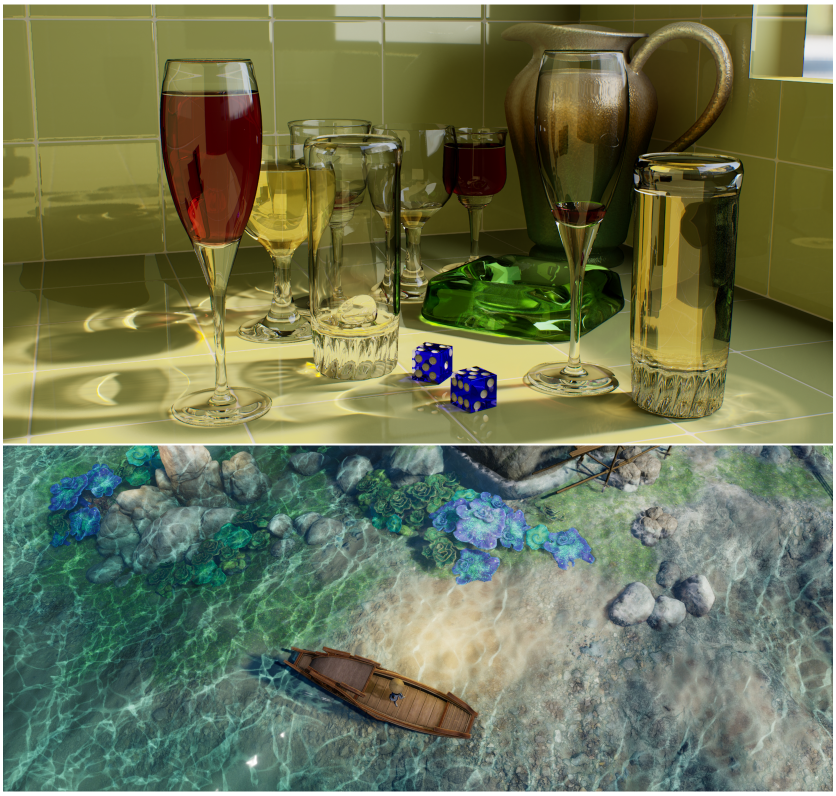


Figure 30-1. *Screenshots of real-time ray traced caustics. Top: the classic "POV-Ray glasses" (courtesy of Gilles Tran). The caustics and glass meshes are ray-traced up to 12-bounce refraction and reflection. Bottom: the undersea water caustic effect from the game JX3 HD Remake developed by Kingsoft Seasun Studio.*

process is prone to exhibit either blurry or noisy results. In the same book, Holger Gruen [4] showed an improved caustics mapping algorithm for underwater caustics, which traced photons from a rasterized water mesh and reconstructed the lighting in screen space. Although being able to eliminate most artifacts in some earlier attempts of rendering underwater caustics, the algorithm still cannot produce sharp but noise-free caustic patterns.

To simulate high-quality caustics in real time by utilizing GPU ray tracing (see Figure 30-1), we present two techniques in this chapter:

1. *Adaptive Anisotropic Photon Scattering (AAPS):* The AAPS technique presented in Section 30.2 is for generating caustics around high-polished metallic and transparent objects. It facilitates traditional

forward photon tracing with photon differentials [5, 10], an anisotropic approach to obtain finer scattering quality and higher efficiency. In addition, inspired by [1, 3, 11], the technique handles photon emission adaptively to generate highly detailed caustic patterns in local regions and to maintain temporal stability. A soft caustic algorithm for area light sources is also provided.

2. *Ray-Guided Water Caustics (RGWC):* The RGWC technique presented in Section 30.3 is for creating caustics above and under water surfaces. It is a continuation of Gruen's work [4] and improves it over several aspects: intensifying the optical details in water caustic by applying photon difference or procedural meshes; supporting most light types and their relevant properties, including textured area lights; being able to cover vast scene regions through cascaded caustics maps; and being flexible on performance-quality trade-off with user-controlled bias-variance preference.

30.2 ADAPTIVE ANISOTROPIC PHOTON SCATTERING

The AAPS method simulates caustics by revamping some techniques from photon splatting, a category of light propagation methods that is a variation of classic photon mapping. In the previous GPU-based method [8], photons are shot and drawn as isotropic particles. AAPS modifies these particles to project elliptical footprints by evaluating photon differentials during hit-bounce time, and then invoking the rasterization pipeline to draw them against the scene depth with additive blending. Such an anisotropic setup can significantly save the bandwidth and bring superior details.

Besides the anisotropic photon setup, our algorithm has two additional novelties:

> A negative feedback loop to distribute photons adaptively into the important areas, i.e., regions close to the camera or parts of the screen where the outcome exhibits temporal instability.

> "Soft caustics" cast by area light sources, which is done by modifying photon differential information at the emission stage.

The algorithm maintains the following buffers:

> *Task buffers:* A set of buffers for guiding the photon emission in the current frame, including the quadtree buffer and the light ID buffer (Section 30.2.1).

- *Photon buffer:* A structured buffer recording photon data, including hit position, photon footprint, and intensity.
- *Feedback buffers:* A couple sets of textures in the light space for tracking feedback information, including the average screen-space area of photon footprints, caustics variance, and ray density (Section 30.2.3).
- *Caustics buffer:* A render target for splatting photons in the screen space.

The workflow is executed in four steps (see Figure 30-2):

1. Emit photons according to the task buffers and trace them through the scene. For any photon hitting an opaque surface, create a record in the photon buffer and add its footprint area to the feedback buffers.
2. Perform photon scattering (splatting) on the caustics buffer: each photon in the photon buffer is drawn as an elliptical footprint against the scene depth. The shape and intensity of the footprint are calculated from photon differentials and the surface normal, respectively.
3. Apply the caustics buffer to the scene, which is usually done by accessing the scene attributes in the G-buffer and performing a deferred lighting pass.
4. Combine the feedback buffers of the previous frame and the current frame to generate the task buffers for the next frame.

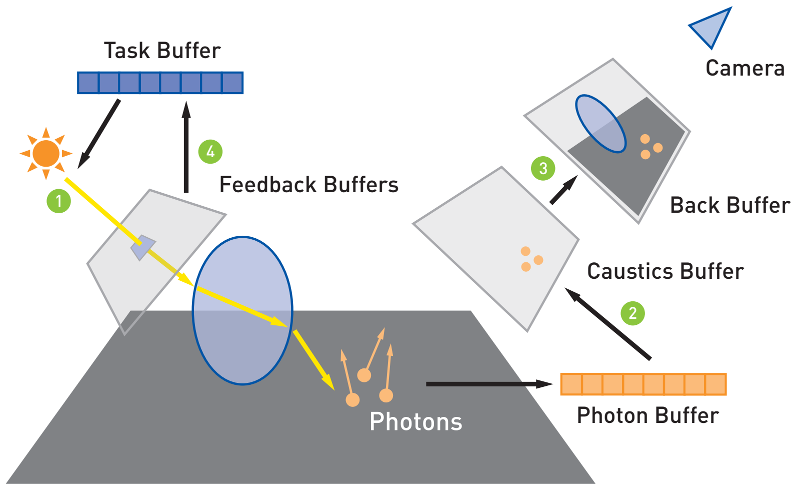


Figure 30-2. *The AAPS workflow. The numbered circles relate to the numbered list in the text.*

In the rest of this section, we first elaborate these four steps in detail (Sections 30.2.1–30.2.3) and describe two approximating methods for simulating dispersion (Section 30.2.4) and soft caustics (Section 30.2.5), respectively. Then, we show our results and the performance tests in typical applications and give some guidance on optimizing performance (Section 30.2.6). Finally, we discuss the algorithm's limitations (Section 30.2.7) and present extended usages (Section 30.2.8).

30.2.1 PHOTON TRACING

To implement adaptive photon emission, we maintain a 2D texture called the *ray density texture* to guide photon distribution, in which all light sources' light-space views are tiled together to track the per-pixel ray count that should be traced (Figure 30-3). The texture is then expanded into a sequence of global ray IDs for photon emission that are placed in a 1D buffer called the *quadtree buffer*, in which a quadtree is constructed for accelerating queries and is updated every frame. An additional 2D texture called the *light ID buffer* is also created for light source lookup.

During photon tracing, each ray generation shader thread is dispatched to trace one ray from one of the light sources. The light source ID and the ray's location in the ray density texture are obtained by querying the quadtree buffer, and the ray's origin or direction is determined by the location mapped to one of the light spaces. Figure 30-4 shows an example of this searching process: First, the shader thread searches for the ray's location, starting from the top of the tree and traversing down to the leaf node whose ID range

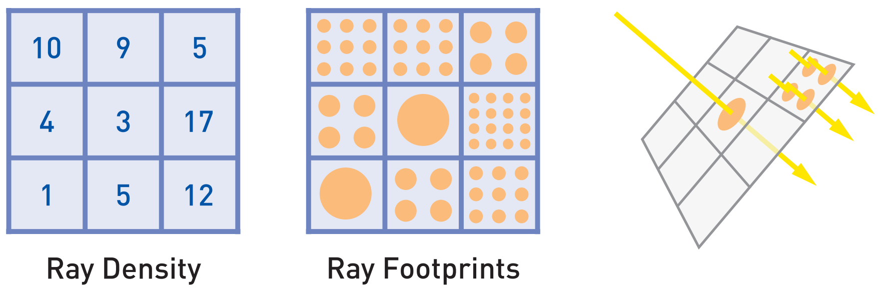


Figure 30-3. *The ray density texture in light space. Left: the per-pixel ray count in the ray density texture. Middle: visualized ray distribution and initial photon footprint size derived from ray density. For each pixel, we set the ray count to the nearest square number less than the number of rays. Right: shooting rays according to the samples' locations in the texture.*

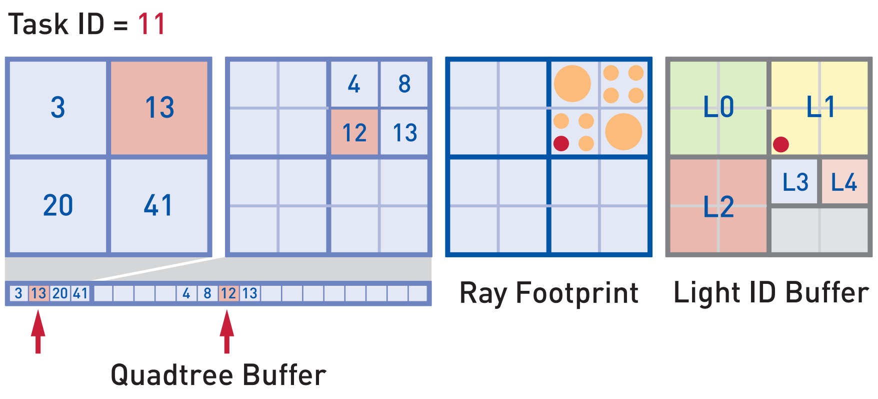


Figure 30-4. *An example of the ray query process: In the quadtree buffer, each quadtree node's ID equals the highest ray ID inside its subtrees, and the ray count in the subtree equals its ID subtracted by its previous sibling's. For a thread with task ID 11, the ray generation shader starts searching from four subtrees 3, 13, 20, and 41 containing 4, 10, 7, and 21 rays, respectively; then, it traverses into subtree 13, which contains four subtrees 4, 8, 12, and 13; finally, it traverses into subtree 12 to find the matching ray ID 11. The 2D location of ray ID 11 is then used for retrieving the light source data from the light ID buffer and generating the UV parameters to sample the light source.*

contains the thread ID. Then, it uses the location to look in the light ID buffer and find out which light source to sample, and it calculates the UV parameters and transforms the location into the light space. Finally, the thread computes the initial size of the photon footprint from the ray density at that position and then sets up a ray to trace the photon. The quadtree query code looks as follows:

```
// taskId is calculated from thread ID.
// sampleIdx is the task offset inside current quadtree node.
uint2 pixelPos = 0;
uint sampleIdx = taskId;
uint4 value = RayCountQuadTree[0];

// Discard threads that don't have a task.
if (taskId >= value.w)
    return;

// Traverse quadtree.
for (int mip = 1; mip <= MipmapDepth; mip++)
{
    pixelPos <<= 1;
    if (sampleIdx >= value.b)
    {
        pixelPos += int2(1, 1);
        sampleIdx -= value.b;
    }
    else if (sampleIdx >= value.g)
    {
        pixelPos += int2(0, 1);
        sampleIdx -= value.g;
    }
    else if (sampleIdx >= value.r)
    {
        pixelPos += int2(1, 0);
        sampleIdx -= value.r;
    }
    // Calculate linear index based on mipmap level and pixel position.
    int nodeOffset = GetTextureOffset(pixelPos, mip);
    value = RayCountQuadTree[nodeOffset];
}

// lightInfo = {light ID, range, anchor point X, anchor point Y}
uint4 lightInfo = LightIDBuffer.Load(int3(pixelPos, 0));
// Get pixel size from ray density texture.
float2 pixelSize; uint2 lightAtlasCoord;
GetRaySample(pixelPos, sampleIdx, lightAtlasCoord, pixelSize);
// Calculate light UV for ray configuration and delta UV for footprint.
float2 lightUV = (lightAtlasCoord - lightInfo.zw) / lightInfo.y;
float2 deltaUV = pixelSize / lightInfo.y;
```

Tracking the photon footprints during the ray tracing is done by estimating photon differentials [10]. Supposing a ray shot from a light has two positional parameters $\mathbf{p} = \mathbf{p}(u, v)$ in the case of a directional light source, or two directional parameters $\mathbf{d} = \mathbf{d}(u, v)$ in the case of a point light, the photon's hit position $\mathbf{p}'$ after the ray tracing is determined by all parameters $\mathbf{p}' = \mathbf{p}'(u, v)$. The algorithm generates two small perturbations for each ray, updates them using the chain rule when the photon hits a surface, and uses the perturbations of photon position $\Delta\mathbf{p}'$ to determine the new size of the photon footprint:

$$\Delta\mathbf{p}' = \frac{\partial\mathbf{p}'}{\partial u}\Delta u + \frac{\partial\mathbf{p}'}{\partial v}\Delta v. \tag{30.1}$$

The photon differentials are evaluated in closest-hit shaders. Once reaching a visible, opaque, and rough surface, the photon's attributes are recorded in the photon buffer, including the hit position, intensity, incident direction, and final footprint, which is determined by the partial derivatives $\partial\mathbf{p}'/\partial u$ and $\partial\mathbf{p}'/\partial v$ at the last hit point.

30.2.2 PHOTON SCATTERING

In the photon scattering step, we construct the photon footprints as quadrilaterals based on their differentials and then draw them into the caustics buffer using additive blending. During the rendering, each footprint's lighting result is computed by the photon's intensity, incident angle, and

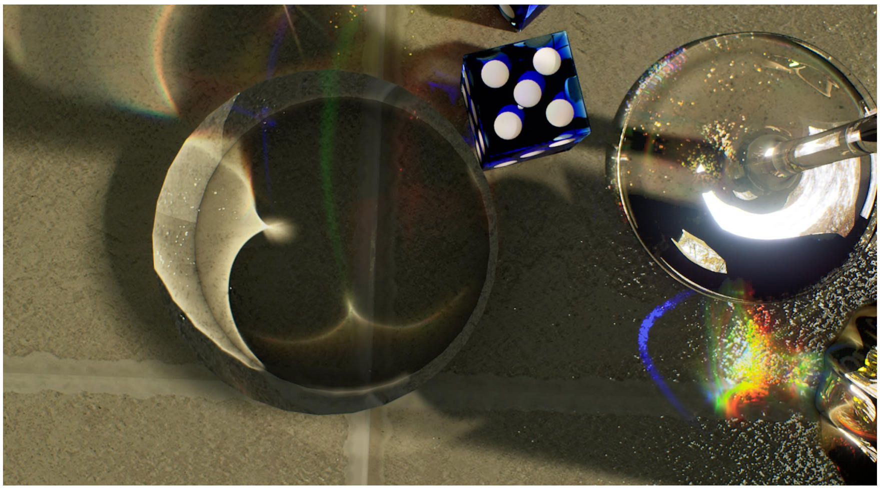


Figure 30-5. *Mesh caustics cast on a glossy surface. A GGX shading pass is performed for each photon footprint.*

surface attributes at the hit point retrieved from the G-buffer. Figure 30-5 shows an example in which the caustics are cast on a glossy surface, where the incident angle and intensity from the photon, the local geometry, and the material information from the G-buffer are collected to perform a GGX shading pass.

After photon scattering, a compute shader back-projects the current screen pixels to the previous frame to calculate the caustics variance between the two frames, and it stores the result in the alpha channel of caustics buffer. The variance values are used for ray density calculation, which is a part of the feedback mechanism (described in the next section).

30.2.3 FEEDBACK BUFFERS

The AAPS technique features a mechanism of a negative feedback loop to determine the photon distribution adaptively. At its core, a couple of textures, together called *feedback buffers*, are placed in the light spaces and updated in each frame to evaluate the spatiotemporal importance of the traced photons (Figure 30-6).

The *projected area texture* contains the average screen-space area of the photon footprints. At the end of photon tracing, each photon's final footprint is calculated and its area in the screen space is added to the light-space location in this texture where the photon was emitted.

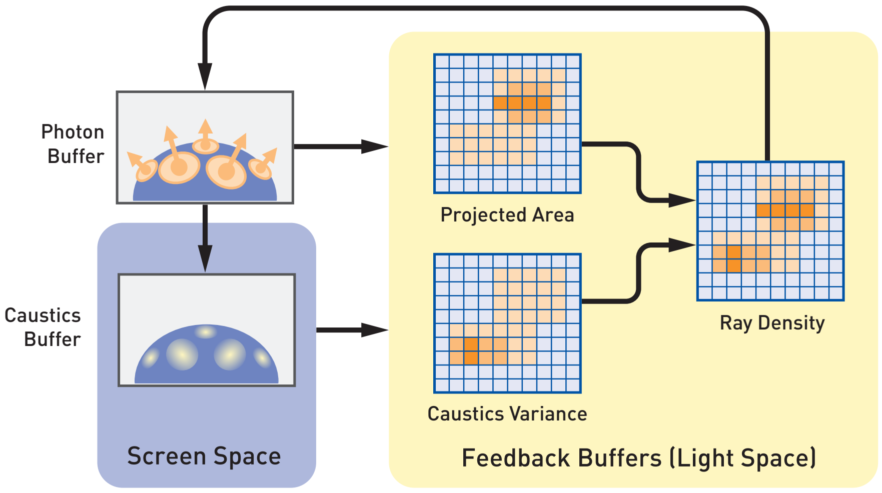


Figure 30-6. *The mechanism of feedback buffers. Top left: at the end of the photon tracing stage, each photon footprint's screen-space area is calculated and added to the projected area texture in light space. Bottom left: the caustics variance between the current and previous frames is stored in the caustics buffer. At the end of the photon tracing stage, each photon samples variance from the caustics buffer and accumulates it into the caustics variance texture in light space. Right: the projected area and caustics variance are combined to generate the ray density.*

The *caustics variance texture* contains the average color variance of the screen-space pixels covered by the photons. At the end of photon scattering, each photon collects the variance value from the caustics buffer's alpha channel at its footprint's center and writes the value into the pixel of this texture where the photon was emitted. This looks as follows:

```
// At the end of photon tracing, calculate screen-space coordinates and
    footprint area.
float3 screenCoord;
float pixelArea = GetPhotonScreenArea(photon.position, photon.dPdx, photon.
    dPdy, screenCoord);
// Read variance value from caustics buffer.
float variance = GetVariance(screenCoord);
// Write feedback buffers; lightAtlasCoord is the 2D coordinate in light
    space.
uint dummy;
InterlockedAdd(ProjectedArea[lightAtlasCoord], uint(pixelArea), dummy);
InterlockedAdd(Variance[lightAtlasCoord], uint(variance), dummy);
InterlockedAdd(PhotonCount[lightAtlasCoord], 1, dummy);
```

The *ray density texture* is computed for the next frame by combining the two previous textures to compute a suggested ray density, which is used as a

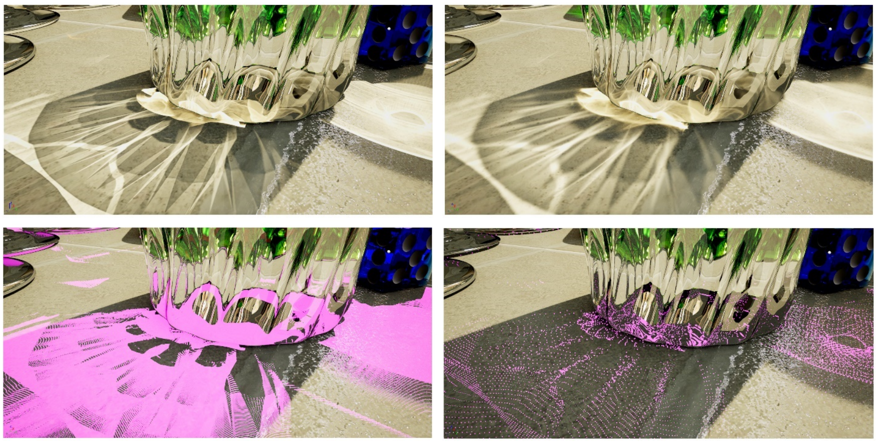


Figure 30-7. *Comparison of target projected area of photon footprint set to 20 (upper left) and 80 (upper right). The second row shows the point visualization of the photons. Detailed caustics patterns are well captured without apparent noise.*

guide for the per-pixel ray count:

$$d' = d\frac{a}{a_t} + vg, \tag{30.2}$$

where d' is the suggested ray density, d is the previous ray density, a is the average screen-space projected area, a_t is the target projected area, v is the caustics variance, and g is the variance gain. In order to create sharper details, we can set a_t to a smaller value to restrict photon size (Figure 30-7 shows a comparison between two a_t values); and to suppress temporal flickering, we can set g to a higher value.

Applying d' directly for subsequent usage may cause the small local features to be unstable if the change is too steep. To avoid this, we filter d' by blending with the neighboring pixels' current ray density:

$$d_{\text{final}} = w_t d' + (1 - w_t)\frac{\sum_i w_i d_i}{\sum_i w_i}, \tag{30.3}$$

where d_{final} is the final ray density, d' is the suggested ray density, the d_i are the current ray densities of the pixel and its neighbors, and w_t and w_i are temporal and spatial weights, respectively, which should be set between 0 and 1. A higher value of w_t enables faster updates but less stable results.

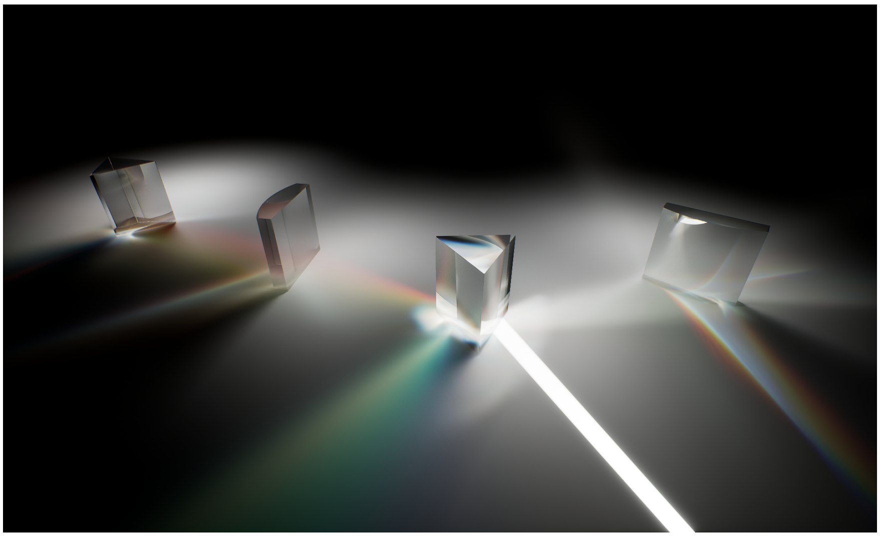


Figure 30-8. *Light dispersion though prisms.*

30.2.4 DISPERSION

Accurately simulating light dispersion (as shown in Figure 30-8) requires computing spectral ray differentials, as done by Elek et al. [2], plus the emission spectrum of light sources and the absorption spectrum of materials. To avoid such complicated input data and huge computational cost, we employ a perturbation-based approach instead.

First, at each refraction point, an index of refraction (IOR) perturbation is calculated based on the thread ID:

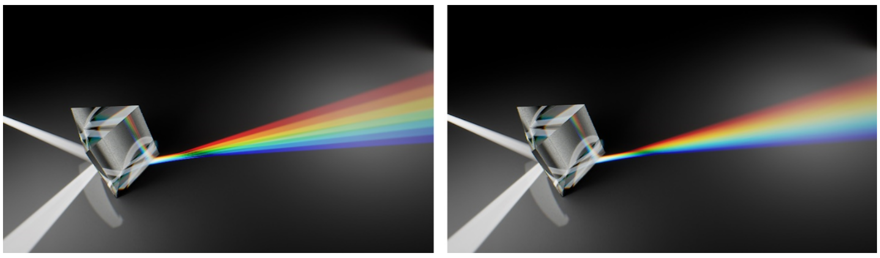
$$\Delta i = 2\frac{t \bmod s_d}{s_d - 1} - 1, \tag{30.4}$$

where Δi is the IOR perturbation in the range $[-1, 1]$, t is the thread ID, and s_d is the number of separated monochromatic colors. In Figure 30-9a, s_d is set to 7, thus the white light is split into seven monochromatic colors. Then, Δi is applied to modify the IOR and generate the refraction ray.

Next, at the end of the photon tracing stage, Δi is used for calculating the modulation color. The RGB triplet of modulation weights can be computed by

$$C_f = saturate\left(-\Delta i, w_g(1 - |\Delta i|), \Delta i\right), \tag{30.5}$$

where C_f is the modulation color in RGB channels and w_g is the weight factor



(a) *Seven color bands* **(b)** *Jittered IOR perturbation*

Figure 30-9. *Monochromatic color separation: (a) The white light is split into seven color bands. (b) Continuous color separation with jittering is applied.*

for the green channel, which ensures that all modulation colors can be combined into grayscale colors:

$$w_g = \frac{s_d + 1}{2s_d - 2}. \tag{30.6}$$

Finally, each photon's color is multiplied by its modulation color C_f. The dispersion effect now looks like a series of colorful bands. To make the color distribution smoother over the spectrum, the IOR perturbation Δi can be jittered. Figure 30-9b shows the results with jittering.

30.2.5 SOFT CAUSTICS

AAPS can simulate soft caustics cast by area light sources. Unlike directional or point light source, an area light source can emit photons with independently varied position and direction. Thus, we need to formulate a proper method on estimating photon differentials based on the four perturbations.

Suppose that a photon emitted from an area light has two positional parameters $\mathbf{p} = \mathbf{p}(u, v)$ and two directional parameters $\mathbf{d} = \mathbf{d}(p, q)$. The final hit point of the photon $\mathbf{p}'$ is determined by all parameters $\mathbf{p}' = \mathbf{p}'(u, v, p, q)$. Adding either a positional or a directional perturbation to the ray's origin will raise a shift to the hit point (Figure 30-10). Our solution for area light sources is to treat all four perturbations $\Delta u, \Delta v, \Delta p, \Delta q$ as independent random variables obeying standard normal distribution and the photon footprint as the significant area of resulting probability distribution. The perturbations of the

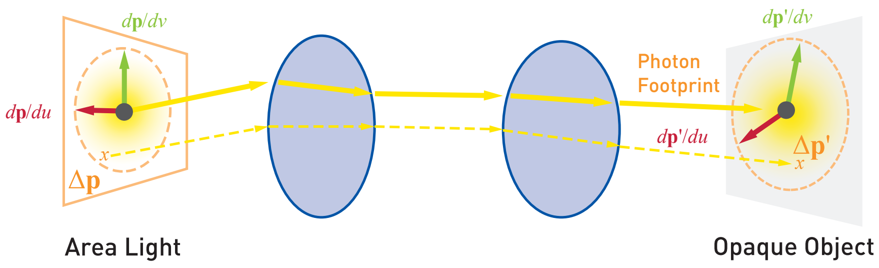


Figure 30-10. *Adding a positional perturbation $\Delta\mathbf{p}$ on the ray's origin will raise a positional perturbation $\Delta\mathbf{p}'$ on the hit point.*

photon position are $\Delta\mathbf{p}' = \Delta\mathbf{p}'_\mathbf{p} + \Delta\mathbf{p}'_\mathbf{d}$, in which

$$\Delta\mathbf{p}'_\mathbf{p} = \frac{\partial\mathbf{p}'}{\partial u}\Delta u + \frac{\partial\mathbf{p}'}{\partial v}\Delta v, \tag{30.7}$$

$$\Delta\mathbf{p}'_\mathbf{d} = \frac{\partial\mathbf{p}'}{\partial p}\Delta p + \frac{\partial\mathbf{p}'}{\partial q}\Delta q. \tag{30.8}$$

Here, $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$ are 2D photon perturbation vectors in the local *xy*-coordinate frame of the photon, raised by positional and directional perturbations, respectively. Because all perturbation inputs are normally distributed, both $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$ obey normal distribution: $\Delta\mathbf{p}'_\mathbf{p} \sim N(\mathbf{0}, \mathbf{C}_\mathbf{p})$ $\Delta\mathbf{p}'_\mathbf{d} \sim N(\mathbf{0}, \mathbf{C}_\mathbf{d})$ in which

$$\mathbf{C}_\mathbf{p} = \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial u} & \frac{\partial\mathbf{p}'}{\partial v} \end{pmatrix} \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial u} & \frac{\partial\mathbf{p}'}{\partial v} \end{pmatrix}^T, \tag{30.9}$$

$$\mathbf{C}_\mathbf{d} = \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial p} & \frac{\partial\mathbf{p}'}{\partial q} \end{pmatrix} \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial p} & \frac{\partial\mathbf{p}'}{\partial q} \end{pmatrix}^T. \tag{30.10}$$

Because $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$ are independent, the probability distribution of $\Delta\mathbf{p}'$ is the convolution of the probability distributions of $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$. Note that the convolution of two normal distributions is still a normal distribution, with the mean value and the covariance matrix being the sum of the two respectively:

$$\Delta\mathbf{p}' \sim N(\mathbf{0}, \mathbf{C}), \tag{30.11}$$

where $\mathbf{C} = \mathbf{C}_\mathbf{p} + \mathbf{C}_\mathbf{d}$. To calculate photon differentials from $\mathbf{C}$, we can find two vectors $\Delta\mathbf{p}_1$ and $\Delta\mathbf{p}_2$ satisfying

$$\mathbf{C} = \begin{pmatrix} \Delta\mathbf{p}_1 & \Delta\mathbf{p}_2 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{p}_1 & \Delta\mathbf{p}_2 \end{pmatrix}^T. \tag{30.12}$$

But, such a parameterization is not unique. We just assume that $\Delta\mathbf{p}_1$ is along the *x*-axis of the local coordinate frame and solve for $\Delta\mathbf{p}_2$ accordingly.

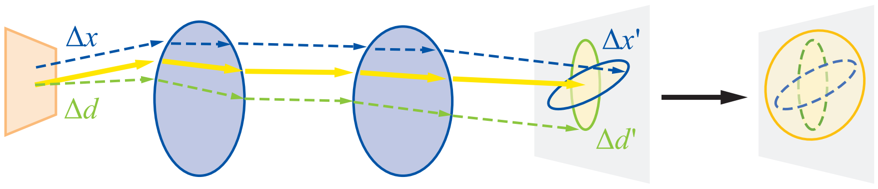
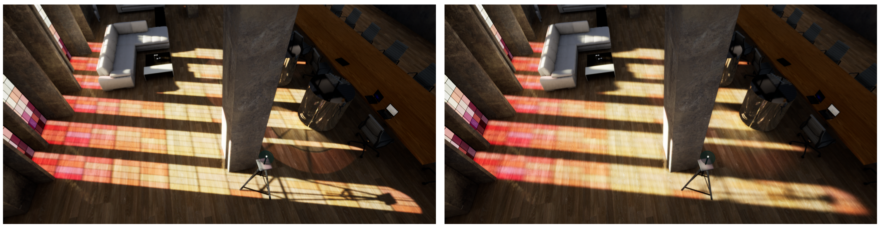


Figure 30-11. *Our soft caustic algorithm tracks photon differentials for both ray position and direction, combines them using convolution and generates a two-dimensional photon footprint.*



(a) *Soft caustics disabled* **(b)** *Soft caustics enabled*

Figure 30-12. *Use soft caustics to simulate soft transparent shadows. (a) The colorful shadow is produced by caustics with softness set to 0. (b) With softness set to 0.15, the shadow is softened. Notice that the caustics generated by a metallic cylinder on the right side are also softened.*

SUMMARY First, our soft caustics implementation estimates photon differentials for both positional and directional perturbations. Then, it constructs the covariance matrices $\mathbf{C_p}$ and $\mathbf{C_d}$ from the differentials and adds both matrices to get the covariance matrix $\mathbf{C}$ for the composite footprint. Finally, it calculates the combined photon differentials $\Delta\mathbf{p_1}$ and $\Delta\mathbf{p_2}$ from $\mathbf{C}$ and applies them to the photon. Figure 30-11 shows the process, and Figure 30-12 shows a scene with and without soft caustics.

30.2.6 RESULTS

The AAPS technique can easily achieve high frame rates for real-time usages while producing accurate, noise-free images. By efficiently applying adaptive photon distribution and anisotropic footprints, the total photons emitted can be kept at a much lower level than traditional offline renderers with similar image quality. Figures 30-13 and 30-14 show two scenes running in real time in which the number of photons is around 50,000 to 100,000.

We picked two scenes for a performance test. The first scene is the classic POV-ray glasses scene shown in Figure 30-1, in which all transparent objects

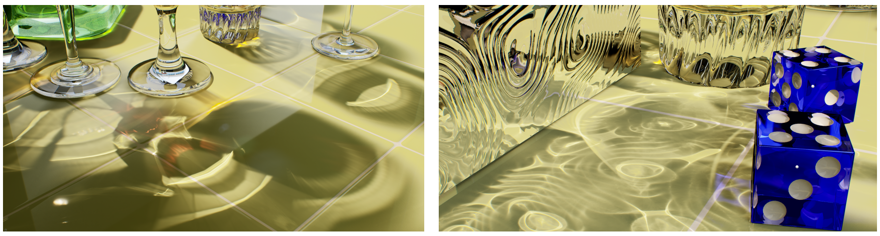


Figure 30-13. *Two views of the POV-ray glasses scene. Left: accurate refractive caustic patterns. Right: reflective caustic patterns cast by normal mapping on the planar metal surface.*

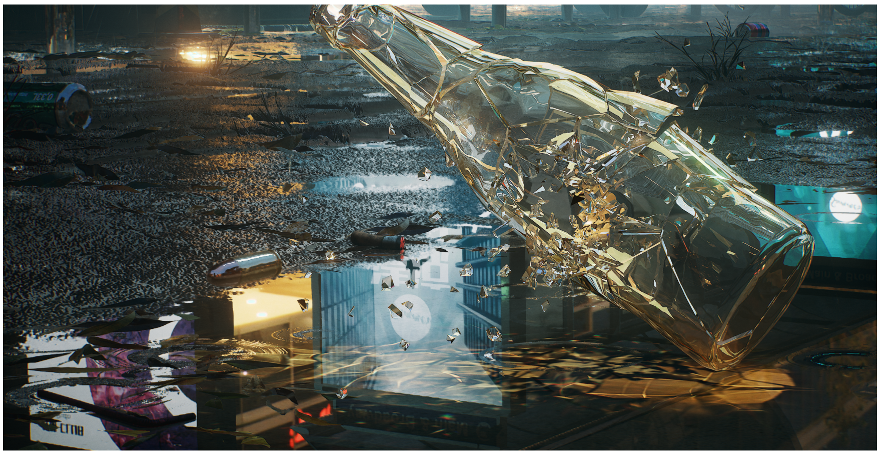


Figure 30-14. *AAPS in real-world applications: a cutscene from the game Bright Memory: Infinite featuring the mesh caustics cast by a shattered glass bottle.*

have both reflected and refracted caustics enabled, and the number of ray bounces is up to 12. The second scene is from the game *Bright Memory: Infinite* and contains a shattered glass bottle (Figure 30-14), where the caustic photons bounce up to eight times before hitting the ground. Beside mesh caustics, both scenes contain large amounts of ray traced refractions and reflections. The tests were performed against 1920 × 1080 and 2560 × 1440 resolutions on selected GPUs. All caustics are rendered at full resolutions without any upscaling technique involved. The frame time breakdowns are listed in Figure 30-15.

Based on the performance chart, we can expect that in a usual setup where caustic effects cover a large portion of the view, the cost of rendering caustics

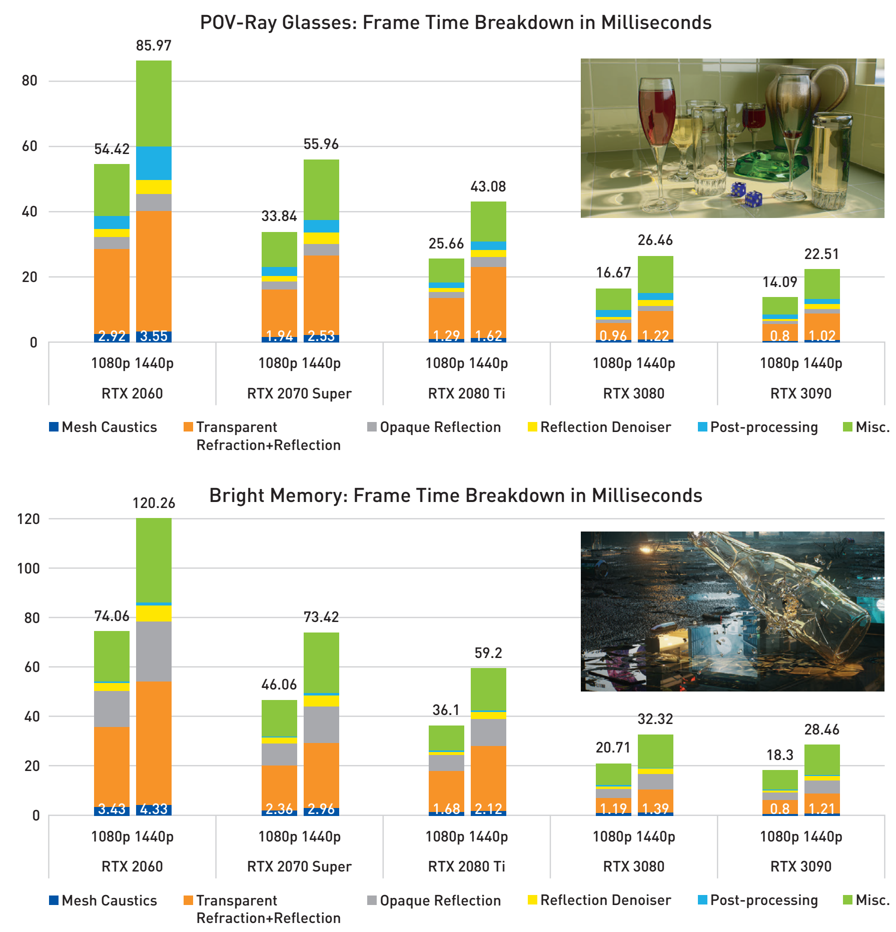


Figure 30-15. *Rendering time breakdown for the POV-ray glasses (Figure 30-1) and Bright Memory (Figure 30-14) scenes at resolutions* 1920 × 1080 *and* 2560 × 1440.

is on the level of 3–4 ms on a performance GPU (RTX 2060) and 1–2 ms on an enthusiast GPU (RTX 3090). These data also show that the cost of caustics is fractional in comparison to other ray tracing regimes. For example, the multi-bounce reflection and refraction on the glass objects take more than 60% of the frame time but are essential for visual quality in a caustic-rich scene. That said, for better performance, the user may have to put more effort toward finding the acceptable appearance of translucent materials other than tweaking caustic parameters.

Further investigation based on profiling tools shows that the AAPS process has two main computational hot spots:

> *Photon tracing:* In the POV-ray glasses scene, the photon tracing stage takes roughly 60% of the caustic rendering time, in which photons bounce up to 12 times before reaching their final hit points. To keep this part of cost under control, the key is to reduce the unnecessary photons, such as excluding materials that can only cast obscure caustics, finding out the maximum acceptable target photon footprint area, culling out photons whose energy falls under a certain threshold, and so on.

> *Photon scattering:* The scattering takes about 20% of the caustic rendering time, which blends all photon footprints into the caustics buffer. To reduce the overhead of blending, the resolution of the caustics buffer can be downscaled to quarter screen size; or consider using an upscaling technique, for example, Deep Learning Super Sampling (DLSS) to boost the frame rate.

30.2.7 LIMITATIONS

In our implementation, the photon tracing step does not use the roughness value in a physically accurate way, thus caustics are still sharp for rough surfaces. We assume that a surface has zero roughness value when calculating photon differentials. This drawback can be relieved by modifying the differentials according to the roughness term. The main challenge is to construct a good approximation to capture reflection and refraction lobes well.

Reflected and refracted caustics are generated in separated threads, which means that having both types of caustics for one object will nearly double the cost of photon tracing.

The calculation of dispersion is not based on continuous spectrum data, thus it is not physically accurate.

Caustic effects seen through reflection and refraction are limited to screen space. For example, if a mirror is placed near a surface that receives caustics, we can only observe the caustics that fall in the current main viewport through the mirror.

The support for area light sources is not optically accurate. For performance consideration, the "soft caustics" cast by area light sources are done by modifying the photon differentials on both ray origins and directions, which means it may not produce the correct contact hardening look in the way of ray traced soft shadows.

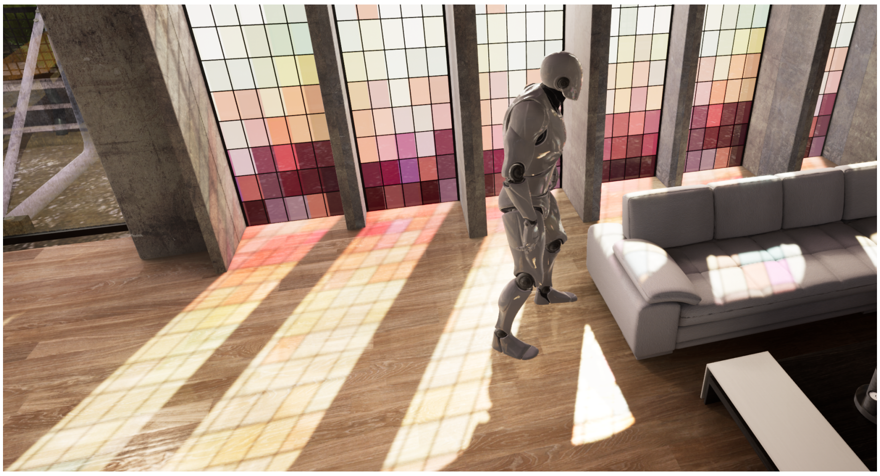


Figure 30-16. *Extended usage of AAPS: simulating transparent shadows. The colorful shadows through the stained-glass windows are one-bounce refractive caustics.*

30.2.8 EXTENDED USAGES

For extended usages other than regular caustic effects cast by glass or metal surfaces, the mesh caustics integration also works for rendering transparent shadows and light spots through textured windows. Figure 30-16 shows a scene with stained glass. The light cast through the windows can be efficiently simulated as a one-bounce refractive caustic. The cost is minimal even if the effect covers a large portion of the screen.

30.3 RAY-GUIDED WATER CAUSTICS

Water caustics have the characteristics of being highly dynamic and interactive, usually covering large areas, and only requiring one-bounce light reflection or refraction, all of which lead us to research in specialized rendering methods. Before describing our latest algorithm improvements, we will give a brief recap to Gruen's method [4]. The method involves mainly two sets of buffers: the *caustics map* stores rasterized water geometry information (positions and normals) from the view of the light source, which may consist of two textures in practice, and the *caustics buffer* accumulates photon footprints in screen space. The workflow consists of four steps:

1. Render water surfaces into the caustics map from the light view, recording the positions and normals of the water surface (Figure 30-17).

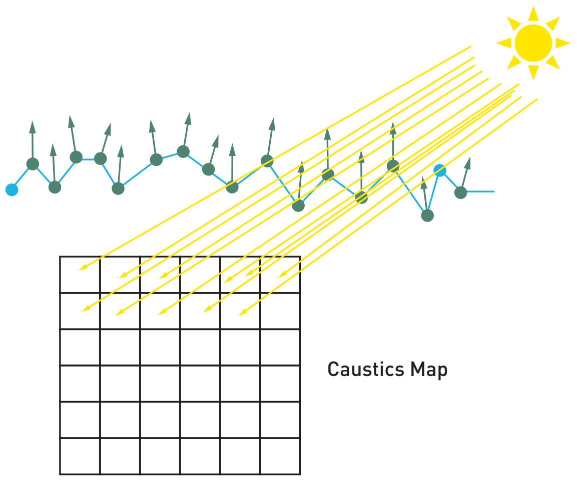


Figure 30-17. *Rendering water geometries into the caustics map from the light view. (From [4].)*

2. Generate rays from positions recorded in the caustics map, and trace them along the reflected or refracted directions calculated from the surface normals. Once the rays hit the scene, record the information of the hit points.
3. Render caustics into the caustics buffer, which is placed in screen space, using the data of the hit points obtained in step 2.
4. Perform denoising on the caustics buffer, and composite the result with the scene.

Our improvements focus on the surface caustics in step 2 and step 3. In step 2, instead of only outputting the intensity of the ray hit point, we count the number of valid hit points and record more data, including position and direction. In step 3, we developed two independent approaches for generating better caustic patterns: *Photon Difference Scattering* (notice that this is not photon differentials), which treats each ray hit point as a photon and renders it as a decal against the scene depth, and *Procedural Caustic Mesh*, which reconstructs the caustic network as a triangular mesh, and then blends it with the scene. As both approaches have pros and cons, users can switch between the two options based on their preference of better performance or higher quality. Besides these overhauls, we also introduce *cascaded caustic maps*, an analog to cascaded shadow maps, to cover mass water bodies by multi-scale rendering.

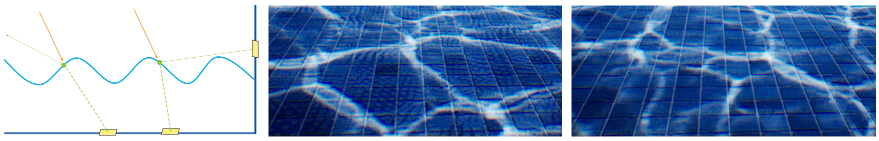


Figure 30-18. *Photon Difference Scattering. Left: creation of quadrilateral photon footprints at ray hit points. Middle: photon footprints visualized as being rendered with equal size; notice the mosaic artifacts. Right: water caustics rendered using PDS with no denoising applied.*

30.3.1 PHOTON DIFFERENCE SCATTERING

The Photon Difference Scattering (PDS) technique, similar to photon differentials, uses the finite difference of nearby rays' data to compute photon coverage, which can give more accurate results than using local perturbations as in photon differentials. Unlike photon differentials, PDS does not need to access the geometry data of the caustics receivers, so it is easier to implement in game engines. With PDS, we can achieve the same quality of the original caustics mapping method by casting much sparser rays, thus greatly improving the ray tracing performance. The brightness adjustment by footprint coverage ensures the correct intensity distribution from all incident angles, while some slope angles may raise artifacts in Gruen's method.

With the PDS approach, we treat ray hit points as photon footprints and render them as decal sprites against the scene depth. Figure 30-18 (middle) shows the photon footprints being rendered at a fixed size, which forms correct caustic envelopes but leaves gaps in between. To fill the scene surface with compact quads, we need to find a proper size for each footprint.

Fortunately, for water caustics each ray is cast from one single reflection or refraction, which means that we can easily backtrace to the ray's origin in the caustics map and access its adjacent rays' origins and directions. The right size of the footprint is then estimated using these ray data around the hit point (Figure 30-19).

The initial size of the footprint is determined by the resolution of the caustics map and the desired precision set by the user, then its scaling factor is derived from the current hit point and the estimated hit points:

$$\text{Scale} = \frac{\text{TriangleArea}(\text{hit0}, \text{hit1}, \text{hit2})}{\text{TriangleArea}(\text{pos0}, \text{pos1}, \text{pos2})}, \tag{30.13}$$

where *hit*0 is the position of the current hit point, *hit*1 and *hit*2 are the

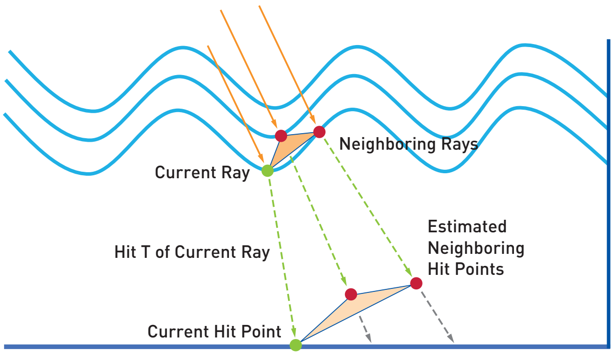


Figure 30-19. *Calculating the size of a photon footprint by combining the ray's origin, direction, and hit point data with the adjacent ray's origins and directions. The hit points of adjacent rays (red) are estimated from the current ray's hit T (green).*

estimated positions of neighboring hit points, and *pos*0, *pos*1, and *pos*2 are the corresponding original positions in the caustics map.

Before finally rendering the photon footprint onto the caustics buffer, the intensity, size, and orientation of the quad sprite are also adjusted by the photon direction and the surface normal: the quad is placed perpendicular to the ray direction and then projected to the opaque surface (Figure 30-20). Because the quad is smeared over the receiving surface, its intensity is cosine weighted by the incident angle. The lighting result is calculated using the scene materials during the scattering.

Choosing a proper shape for footprints can significantly improve the quality. Figure 30-21 shows that the elliptic footprints provide better result than the rectangular ones. In Figure 30-18 (right), after PDS was applied, the footprints formed into continuous patterns without any denoising involved.

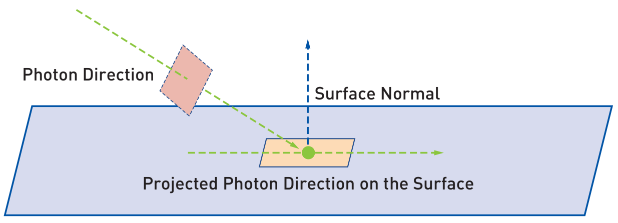


Figure 30-20. *The intensity, size, and orientation of the quad sprite are also adjusted by the photon direction and the surface normal.*

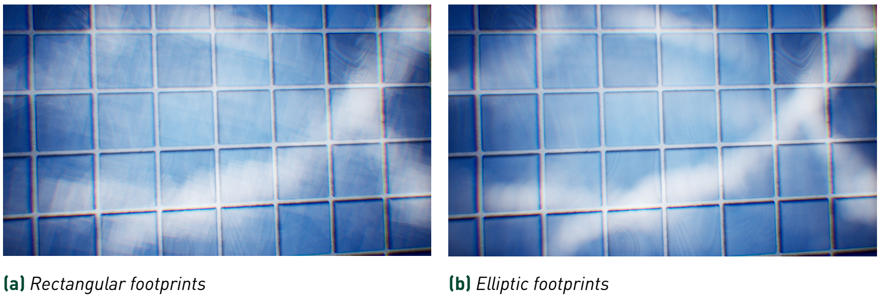

(a) *Rectangular footprints* **(b)** *Elliptic footprints*

Figure 30-21. *Comparison of caustics footprints: (b) elliptic footprints provide sharper and clearer caustics than (a) rectangular footprints.*

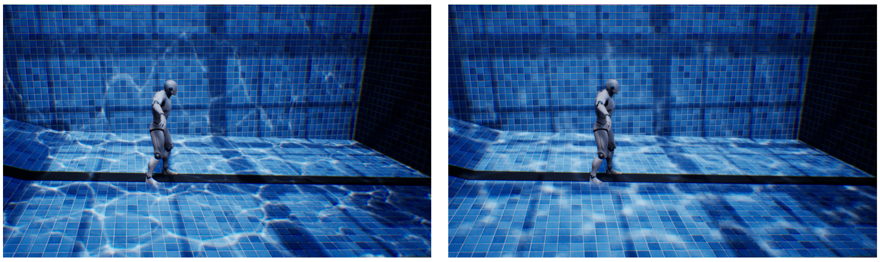


Figure 30-22. *The results of using different caustics map resolutions with photon difference scattering. Left: resolution* 2048 × 2048 *covering a* 30 *m* × 30 *m area. Right: resolution* 512 × 512 *covering the same area.*

On the pro side, PDS can render high-quality water caustics with low cost and can easily extend the supports for many types of light sources, including area light. On the con side, it is sensitive to the caustics map resolution related to the covering range—applying a low-resolution caustics map to a large scene area may result in very blurry caustic patterns (Figure 30-22).

30.3.2 PROCEDURAL CAUSTIC MESH

The other approach for reconstructing caustic patterns is Procedural Caustic Mesh (PCM), which converts hit points into an intermediate mesh: each hit point is mapped to a vertex in the mesh whose topology is a triangle list that maps to the regular grids in the caustics map. After the ray tracing pass, a compute shader fetches the hit point data, evaluates the contribution and intensity of each primitive according to its world-space area, discards invalid primitives, and generates the index buffer. The mesh is then rendered onto the caustics buffer in a rasterization pass (Figure 30-23). In practice, we build

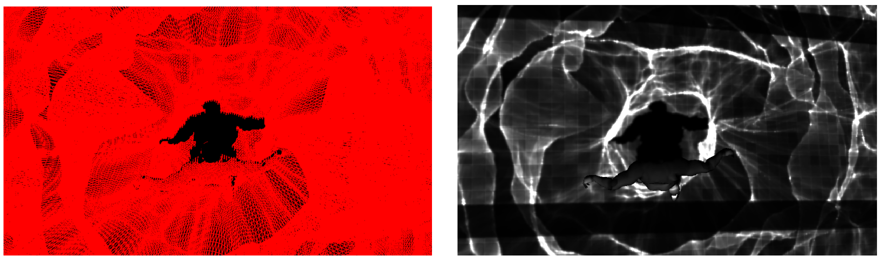


Figure 30-23. *Procedural Caustic Mesh. Left: the refracted caustic mesh; notice that the shadow area is culled from the mesh. Right: the rendered mesh modulated by scene materials.*

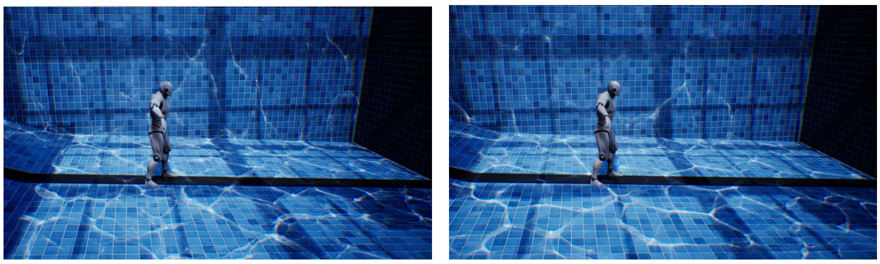


Figure 30-24. *The results of using different caustics map resolutions with PCM. Left: resolution* 2048×2048 *covering a* 30 *m* × 30 *m area; Right: resolution* 512×512 *covering the same area. Notice that the qualities of the two are almost the same.*

two procedural caustic meshes for every water object, one for reflection and the other for refraction.

The advantage of this approach is that it always produces sharp caustic patterns even if the caustics map resolution is very low (Figure 30-24). However, it also raises the "black edge" artifact at object corners where the mesh triangles span over the culling region (Figure 30-25). For the best result, users can choose between PDS and PCM for better quality. In general, PDS is more flexible and well suited for water areas in a confined space like swimming pools, whereas PCM is more efficient when coupling with large water bodies, such as lakes and oceans.

30.3.3 CASCADED CAUSTICS MAPS

To generate caustics for large water bodies like ocean surfaces, we have implemented *cascaded caustics maps* (CCM) that work in the same way as cascaded shadow maps (CSM). Figure 30-26 shows a configuration of four

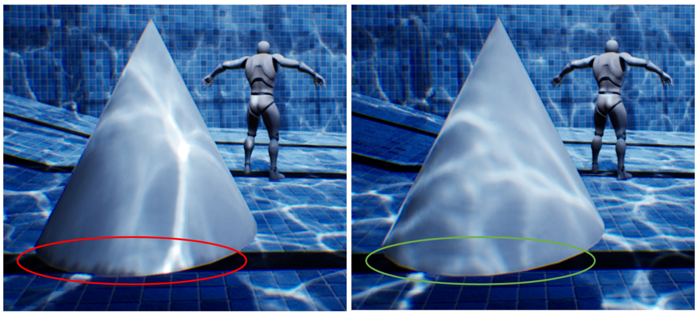


Figure 30-25. *Left: artifact raised by PCM along the discontinuous edge of the scene geometry due to inaccurate culling of mesh primitives. Right: in comparison, PDS has no such artifact.*

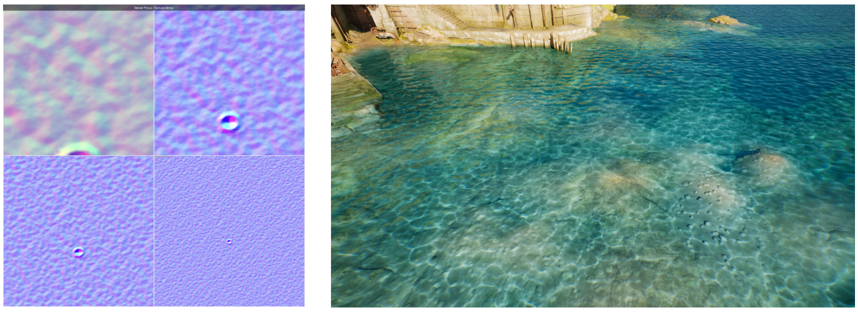


Figure 30-26. *Left: a CCM configuration of four cascades. Right: an aerial view of the ocean surface with reflective and refractive caustics rendered by CCM.*

cascades, where each cascade contains a caustics map at a user-selected resolution. CCM can be coupled with PDS to mitigate blurry results when rendering with limited photon budget but higher details are desired at near sight, as it allows us to distribute more photons at the innermost cascade to lift the quality and less photons at the outer cascades to keep the cost down.

Unlike CSM, to implement cascaded maps in a ray tracing pipeline, we need to determine the number of shader threads for dispatching rays and the scheme of assigning data to the threads. The following formula gives the threads needed for the CCM:

$$N_{\text{thread}} = W \times H \times \left[1 + \left(1 - \frac{1}{\text{Scale}^2}\right) \times \left(N_{\text{level}} - 1\right)\right], \tag{30.14}$$

where W and H are the width and height of the caustics map, Scale is the

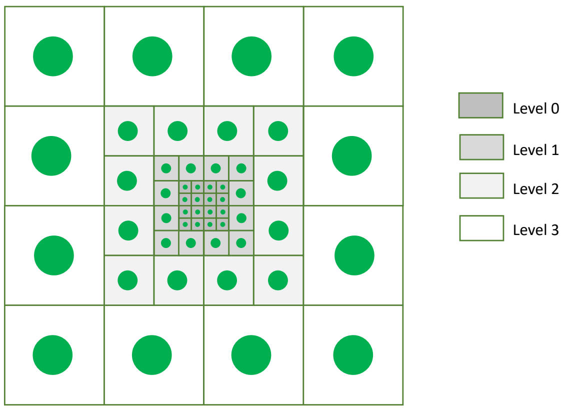


Figure 30-27. *Assigning CCM data to ray generation shader threads: A setup with four cascades, each cascade at* 4 × 4 *resolution, with size scaling* 2 *between the cascades, where the gray areas are overlapped. The cascaded setup only dispatches* 52 *threads in total, while a uniform setup requires* 1024 *threads.*

length ratio between two adjacent cascade levels, and N_{level} is the number of cascades. Figure 30-27 demonstrates an example on the mapping between the CCM and the shader threads.

30.3.4 SOFT WATER CAUSTICS BY AREA LIGHTS

We have extended the PDS method to simulate soft water caustics cast by area light sources. Currently, only textured rectangular lights are supported, but the technique can be easily expanded to accommodate more complicated area lights. Applying a 2D texture to a rectangular light to define the surface intensity and to emit photons accordingly allows us to simulate any planar luminaries. Unlike with other light types, the photons leaving an area light are going in all directions (Figure 30-28). Thus, water caustics exhibit "softness" in the way of soft shadows.

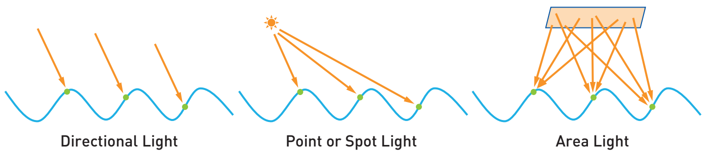


Figure 30-28. *Only one ray hits each point on the water surface with directional, point, and spot lights. But with an area light, each point on the water surface may be hit by many rays.*

To simulate this soft effect in real time, we developed a temporal method to reuse the caustic patterns over multiple frames. It consists of two steps:

1. For each valid point in the caustics map in the current frame, we select an incident ray from a random sampling point on the rectangular light, and we trace it through the water surface.
2. We accumulate the caustics over several frames with a temporal filter to output the soft caustics.

Performance-wise, we can produce acceptable soft water caustics by any rectangular lights at merely the same cost as other light types.

30.3.5 RESULTS

The RGWC technique rasterizes the water mesh into a caustics map, avoiding ray tracing between light sources and water surfaces, which greatly helps the rendering efficiency. Meanwhile, the two caustic reconstruction methods, PDS and PCM, allow the creation of high-quality patterns while covering large water bodies. In addition, CCM and soft water caustics greatly expand the usability of RGWC.

The performance of RGWC is affected by many factors, such as the resolution of the caustics map and the caustics buffer, the water coverage in the view, and the number of light sources that affect the caustics. We performed some testing using the two scenes shown in Figure 30-29.

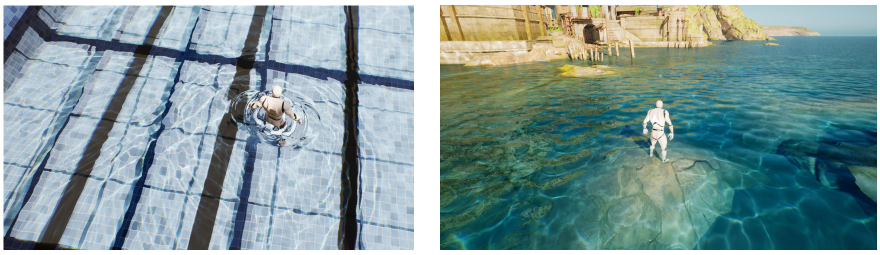
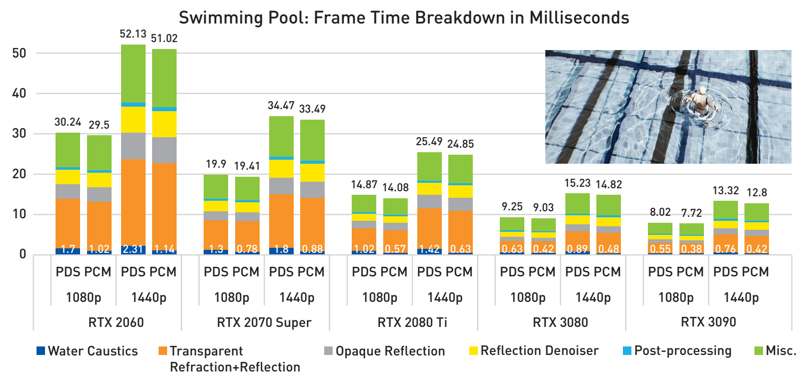


Figure 30-29. *Two scenes selected for RGWC performance tests. Left: swimming pool lit by one directional light source. Right: seaside town lit by one directional light and having a four-cascade CCM configuration. Both scenes have above and under water caustics enabled. The resolution of the caustics buffer is set to full screen size. The caustics map is set to* 1024×1024.



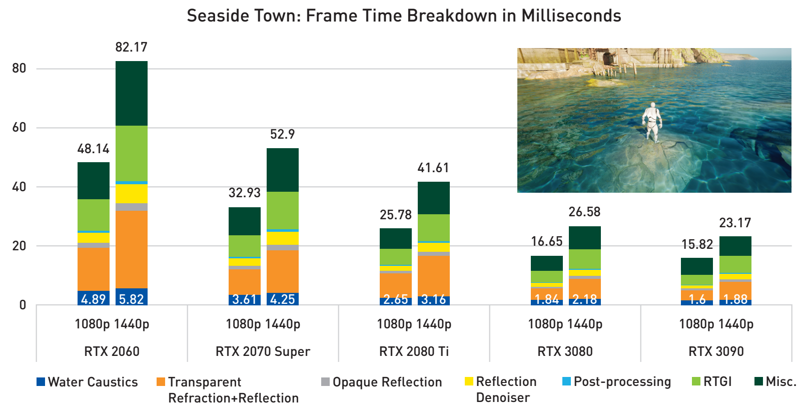


Figure 30-30. *Water caustics rendering time breakdown for the scenes Swimming Pool (top) and Seaside Town (bottom) at screen resolutions* 1920 × 1080 *and* 2560 × 1440*.*

Figure 30-30 (top) compares the performance of PDS and PCM on selected GPUs using the test scene Swimming Pool (Figure 30-29, left). In general, PCM is faster than PDS. Further investigations using profiling tools reveal that part of the overhead of PCM comes from the workload that reconstructs the new index buffer, which is dependent on the vertex number (same as the size of the caustics map) but less dependent to the size of the caustics buffer. And rendering the mesh avoids rendering overlapping sprites in PDS. This explains why PCM is more efficient than PDS and less impacted by the screen resolutions.

Figure 30-30 (bottom) lists the cost of CCM using the test scene Seaside Town (Figure 30-29, right). The scene has four cascades of a 1024 × 1024 × 4 caustics map to cover the sea surface of 240 × 240 m^2. Because CCM does not work with PCM, only PDS numbers are shown here. The usage of CCM significantly increases the cost of RGWC because all four caustics maps are ray traced. However, the aerial view of the sea caustics cannot be easily handled by PCM. Thus, CCM is the only option here to produce high-quality caustic patterns for large scene coverage.

30.3.6 LIMITATIONS

In our implementation, we only render the reflected and refracted caustics generated by the first bounce of photons. Photons bouncing multiple times among the water ripples are not captured.

CCM only works for directional lights and does not support PCM.

Water caustics cannot be seen within ray traced reflections. For example, when underwater, the total internal reflection on the water surface does not show the underwater caustics.

30.4 CONCLUSION

In this chapter, we introduced two techniques to render caustics effect for translucent or metallic objects and water surface. Adaptive Anisotropic Photon Scattering can produce high-quality caustics effects with a reasonable number of rays each frame, which insures the high performance required by many games. As a bonus effect, this technique can also be used to produce shadows cast by translucent objects, such as colored glass windows. Ray Guided Water Caustics is a highly specialized technique to simulate one-bounce water caustics, being very efficient and versatile to handle all sorts of water bodies. Also, it does not need to track light propagation with photon differentials that require additional hit shader code in all materials, which makes it easy to integrate into commercial products.

We have integrated both techniques into NVIDIA's customized Unreal Engine 4 branch. The source code can be obtained at the repository [9].

ACKNOWLEDGMENTS

We thank Nan Lin for providing reference materials and helping edit this chapter. Thanks to Jiaqi Wang for his numerous work on creating and polishing test scenes, performance tuning, and rendering quality control. Thanks to Tao Shi for measuring performance statistics on various hardware and configurations. We would also thank Evan Hart for his advice on CPU-side performance optimization.

REFERENCES

[1] Boksansky, J., Wimmer, M., and Bittner, J. Ray traced shadows: Maintaining real-time frame rates. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 159–182. Apress, 2019.

[2] Elek, O., Bauszat, P., Ritschel, T., Magnor, M., and Seidel, H.-P. Spectral ray differentials. 33(4):113–122, 2014. DOI: 10.1111/cgf.12418.

[3] Estevez, A. C. and Kulla, C. Practical caustics rendering with adaptive photon guiding. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks*, pages 1–2, 2020.

[4] Gruen, H. Ray-guided volumetric water caustics in single scattering media with DXR. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 183–201. Apress, 2019.

[5] Igehy, H. Tracing ray differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 179–186, 1999. DOI: 10.1145/311535.311555.

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

[7] Kim, H. Caustics using screen-space photon mapping. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 543–555. Apress, 2019.

[8] 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*, pages 71–78, 2013. DOI: 10.1145/2448196.2448207.

[9] NVIDIA. NVIDIA RTX experimental branch of Unreal Engine 4. https://github.com/NVRTX/UnrealEngine/tree/NvRTX_Caustics-4.26, 2020. (Registration required to access link. See https://developer.nvidia.com/unrealengine).

[10] Schjoth, L., Frisvad, J. R., Erleben, K., and Sporring, J. Photon differentials. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, pages 179–186, 2007. DOI: 10.1145/1321261.1321293.

[11] Wyman, C. and Nichols, G. Adaptive caustic maps using deferred shading. *Computer Graphics Forum*, 28(2):309–318, 2009. DOI: 10.1111/j.1467-8659.2009.01370.x.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.