Noname manuscript No.
(will be inserted by the editor)

Rasterization-based Progressive Photon Mapping

Iordanis Evangelou · Georgios Papaioannou · Konstantinos Vardis · Andreas A. Vasilakis

Abstract Ray tracing on the GPU has been synergistically operating alongside rasterization in interactive rendering engines for some time now, in order to accurately capture certain illumination effects. In the same spirit, in this paper, we propose an implementation of Progressive Photon Mapping entirely on the rasterization pipeline, which is agnostic to the specific GPU architecture, in order to synthesise images at interactive rates. While any GPU ray tracing architecture can be used for photon mapping, performing ray traversal in image space minimises acceleration data structure construction time and supports arbitrarily complex and fully dynamic geometry. Furthermore, this strategy maximises data structure reuse by encompassing rasterization, ray tracing and photon gathering tasks in a single data structure. Both eye and light paths of arbitrary depth are traced on multi-view deep G-buffers and photon flux is gathered by a properly adapted multi-view photon splatting. In contrast to previous methods exploiting rasterization to some extent, due to our novel indirect photon splatting approach, any event combination present in photon mapping is captured. We evaluate our method using typical test scenes and scenarios for photon mapping methods and show how our approach outperforms typical GPU-based progressive photon mapping.

Keywords Photon mapping · Rasterization · Ray tracing

I. Evangelou · G. Papaioannou · K. Vardis · A. A. Vasilakis
Department of Informatics, Athens University of Economics & Business, Greece

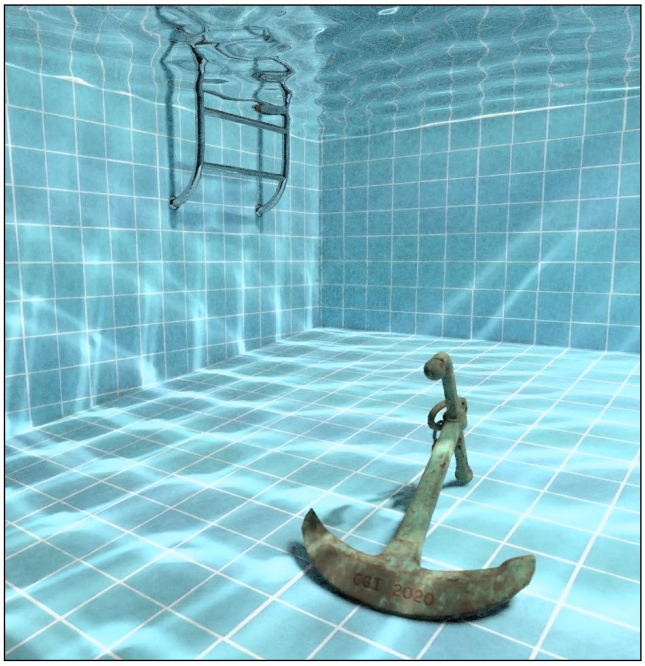

Fig. 1 Converged example of our rasterization-based progressive photon mapping method, using depth 3+3 (light + camera). Iteration time (1M pixel samples): 50ms on an NVIDIA RTX 2080 Ti.

1 Introduction

Photon Mapping [7,8] is a well-known two-stage approximation to bidirectional path tracing, where light-carrying paths or *photons* deposit and cache the carried flux on non-specular surfaces, pre-multiplied with the light path throughput. A data structure, the *photon map*, is responsible for the storage and fast indexing of these particles. Subsequently, for multiple paths traced from the camera, the contribution of photons to hit

points on non-specular interfaces is estimated, converting flux to radiance and modulating it with the combined throughput to the sensor. Since the probability of path vertices from the camera and photon tracing coinciding is zero, photon mapping relies on a kernel function that performs photon flux *density estimation* to integrate the contribution of particles in the vicinity of a camera hit point. As the stored particles in the photon map increase, if a kernel enclosing a fixed number of particles is used, denser areas will result in a tighter and more accurate estimator of scattered radiance, while wider ones may introduce significant bias. The considerable storage required by the photon map(s) and the inability to predict how many photons are adequate to converge to an accurate image estimation for a particular scene led into what is known as the *Progressive Photon Mapping* algorithm [4] (PPM). Its subsequent evolution led to the *probabilistic* approach [10] to PPM that we use in this paper (see related work in Sec. 2).

The general idea in all PPM variants is that instead of tracing and storing a huge amount of particles, photons are iteratively traced in batches and contribute energy to camera path hit points, leading to a tighter and predictable memory budget. At the same time, the radius of the kernel function is gradually tightened as more photons contribute to each hit point. For infinite iterations, the estimator is guaranteed to converge to the expected value.

The motivation behind our work is the definition of a photon mapping method that operates entirely on and benefits from the rasterization pipeline, while it a) correctly captures all photon mapping light transport paths and b) operates at a performance comparable with or better than established GPU-accelerated methods. The rasterization pipeline is universally implemented in hardware. On the other hand, the most efficient to-date frameworks for ray tracing, along with the supporting implementation of acceleration data structures, are hardware-architecture-specific.

In our probabilistic PPM variant, we perform all ray tracing operations using the DIRT architecture [26], a multi-view image-space approach for unidirectional path tracing using accurate, analytic ray triangle intersection tests. We exploit the intermediate camera rays hit map that is already built as part of the image-space ray traversal, to eliminate the need to build and maintain a separate *importon* (camera path particle) map. Second, we exploit the fact that in DIRT a valid image-space projection can be established for any point in space in order to perform *indirect* splatting to hit points invisible to the primary camera view. These are the hit points of secondary camera path segments. This is actually the first rasterization-based approach to encompass both direct and indirect splatting, capturing all possible photon mapping paths, including L(S∗)D(S+)E ones.

DIRT builds a hierarchical cubemap of deep buffers that index geometry polygons. In terms of performance, since the image-space acceleration data structure (ADS) is built very fast via rasterization, our method is ideal for the interactive preview of photon mapping in scenes with dynamic geometry (see the pool example in Fig. 15). Furthermore, even for static scenes, the elimination of the particle index build step in each iteration of the stochastic PPM provides a significant performance gain compared to a respective general-purpose GPU implementation, as shown in Section 5.

Briefly, the main contributions of this work are:

- Implementation of the full PPM algorithm on the hardware graphics pipeline, taking advantage of rasterization for both the population of the ADS and photon splatting.
- Reuse of the same data structure to perform both tracing and camera particle (importon) storage.
- Reordering of camera path tracing and photon tracing in each progressive iteration, with respect to the original probabilistic PPM approach, in order to efficiently distribute energy from photons to camera path hits at arbitrary trace depths.
- Exploitation of an image-space ADS that encompasses the entire scene geometry to splat photons on camera hits at any camera path depth, not just the directly visible points. In contrast to previous photon splatting techniques, this effectively allows capturing all possible events. We also employ Russian Roulette during splatting, to reduce photon energy distribution cost.
- Support for arbitrary light sources, exploiting light G-buffers for fast first bounce estimation of photons, where possible.

The rest of the paper is organised as follows: Section 2 includes a summary of prior art. Section 3 presents an overview of the stages involved in our method. Details about specific steps introduced in this paper and their implementation are provided in Section 4. Next, in Section 5 we report results and evaluate the efficiency of our method. Finally, Section 6 provides conclusions and future research directions.

2 Background and Related Work

We provide here a brief overview of the literature related to physically-based interactive rendering and, more specifically, progressive photon mapping theory.

Progressive Photon Mapping. The original algorithm, which was introduced by Hachisuka et al. [4], loops through two main steps; first, camera paths are traced, storing the hit point and corresponding path throughput at all non-specular events. Next, multiple photon tracing passes follow, where for each one, the method iterates through all stored camera hits and registers the photons within the density estimation radius of each hit point. The newly discovered photons are used for the progressive refinement of both the radiance estimate and the radius at each camera hit point. After all the photons of the current batch have been processed, they are discarded and a new photon tracing pass commences. This continues until the image has adequately converged. Hachisuka et al. further proved that with the above refinement steps, photon density tends to infinity and the result remains consistent.

The idea was further evolved in *Stochastic Progressive Photon Mapping* (SPPM) [3], where hit points generated from stochastic path tracing from the camera at the same pixel share photon density estimation statistics, at the pixel level, allowing PPM to capture distribution effects in a tractable manner. Later on, Weiss et al. [28] extended SPPM to take into account pre-defined animated scenes when calculating the pixel statistics.

In 2011, Knaus and Zwicker [10] proved that the rate at which the radii are reduced is independent of the local photon density and that the gathering of any local density estimation statistics is unnecessary. The proposed algorithm (abbreviated here as PPPM), which we briefly describe below, exhibits the same convergence behaviour as the original progressive photon mapping approach.

The estimate of the radiance $\hat{L}(\mathbf{x}, \omega_o)$ at a camera hit point $\mathbf{x}$, due to the photons $j = 1 \dots M$ with position $\mathbf{x}_j$ and contribution γ_j in a radius r, prior to multiplication with the camera path throughput up to that point is:

$$\begin{aligned} \hat{L}(\mathbf{x}, \omega_o) &= \frac{1}{M} \sum_{j=1}^{M} k_r(||\mathbf{x} - \mathbf{x}_j||)\gamma_j, \\ k_r(\xi) &= \frac{1}{r^2} k\left(\frac{\xi}{r}\right), \end{aligned} \tag{1}$$

where the photon contribution γ_j is the product of the emitted photon flux and the BSDF at each scattering event up to $\mathbf{x}$ (inclusive), divided by the probability density function of the sampling process that generated the photon path. $k(\xi)$ is a user-defined canonical kernel function and r is the *bandwidth* of the kernel, which defines the area over which the density estimation takes place with non-zero photon contributions. M is the number of emitted photons in the batch. At each step i, the radius is refined according to a constant $\alpha \in (0, 1)$, which controls how fast the density estimation area constricts:

$$r_{i+1} = r_i \sqrt{\frac{i + \alpha}{i + 1}}. \tag{2}$$

Higher α values favour more drastic variance reduction and slower error minimisation, while lower values do the opposite. In the original paper, the authors make a comprehensive study of the error and variance and also provide an estimator for volumetric photon mapping.

Interactive Ray Tracing. For the interactive rendering of static scenes, environments with small updates and/or rigid-only motion, any established offline rendering method relying on tracing rays through the environment can nowadays be easily mapped to GPU architectures using any of the available or emerging ray tracing SDKs.

Fast ray traversal typically requires more optimised ADS construction, which translates to increased construction times, practically rendering this stage a preprocessing one, outside the ray shooting and traversal cycles. For a comprehensive performance analysis of this genre of ADSs and corresponding methods, the interested reader may refer to [27].

Despite various ADS readjustment strategies for animated data, interactivity becomes problematic, in general, when geometry is dynamically computed, tessellated or topologically changed very frequently, since the ADS has to be rebuilt from scratch.

Targeting interactive scene updates, one-level or hierarchical uniform grids [9] can be built very fast and offer reasonable traversal performance for relatively uniform primitive distribution in the scene. Perspective grids [2] improved the primitive density per ADS cell and the traversal speed for coherent ray batches, e.g. primary rays. Fragment-based data structures are very fast to build by exploiting the hardware rasterization pipeline. They range from typical rasterized voxel grids, to single-view multi-layer ADSs (e.g. [14]) and finally, to multi-view data structures such as orthogonal A-buffers [5] and cubemap A-buffer [25]. Ray traversal is usually performed with object-space ray marching [22] or image-space ray marching, in a linear [17] or hierarchical manner [24]. While fragment-based ADS methods can very efficiently support dynamic scenes, they generally create sub-optimal structures and often result in poor sampling of oblique or small geometry, leading to ray misses. Thus, to produce more accurate results, a primitive-based ADS can be employed instead.

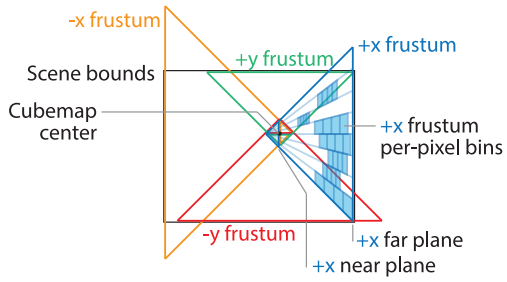

Fig. 2 The deep cubemap data structure used for primitive indexing and ray traversal in the DIRT ray tracing architecture [26]. Here, an xy plane cut-section is shown.

Deferred Image-space Ray Tracing (DIRT). Since for all tracing operations we rely on the DIRT architecture by Vardis et al. [26], we present here a high-level overview of its operation. DIRT introduced a pure rasterization-based approach for analytically tracing rays at interactive frame rates. Primitives are rasterized and their IDs are stored in a user-centred, deep cubemap-like arrangement, encompassing all the scene geometry (Fig. 2). Each multi-layer cubemap side is implemented as an A-buffer linked-list data structure using a per-pixel variable-range, uniform binning to store the primitive information. Ray traversal is performed hierarchically in image space, moving to a different view of the cube-map arrangement, if necessary. Various empty space skipping mechanisms are proposed and implemented, including hierarchical image-space ray marching.

Ray hits are recorded in a sparse *hit buffer* list. The hit buffer is indexed by cubemap pixel coordinates and each cell may contain more than one entry, corresponding to multiple rays intersecting a cubemap pixel. Shading data interpolation is batched for all hits in a separate pass, prior to shading and spawning of new rays. The hit buffer can be easily masked for the efficient execution of the latter stages.

Interactive Photon Mapping. In the interactive rendering domain, the first attempt to perform photon mapping entirely on the GPU, dates back to 2003, with the work of Purcell et al. [21]. Image space photon mapping [16] takes advantage of the light and camera G-buffers to dispense with the tracing of the first light and camera path segments, while continuing the tracing of the intermediate segments in the CPU. Yao et al. [29] performed all tracing operations in image space over multiple single-layer cubemaps in the environment. The success of the approach greatly depended on the ability of the selected number of cubemaps to capture the entirety of the scene. Mara et al. [13] study various strategies for gathering and distributing photon flux and, among other options, optimised the voxel hashing idea of Ma and McCool [12] for the GPU as an indexing scheme for nearest photon search for gathering. Fast gathering of k-nearest photons for interactive rendering has also been investigated, leading to specialised nearest neighbourhood search approaches, such as [11].

The photon density estimation step of photon mapping is often too costly to perform in interactive rendering. An alternative approach is to perform the inverse operation, where energy from each photon is distributed to affected points directly visible to the camera, using a pre-calculated radius of influence. When primitive rasterization is used for covering the area of support of the kernel function in image space, the process is referred to as *photon splatting* [23]. Although simple to implement in the GPU, many works in the literature focus on issues that arise such as excessive overdraw of particles and wasteful computations (e.g. [13]). Moreau et al. [18] implemented a hybrid photon mapping solution, where photons are traced using the NVIDIA OptiX framework [20] and their energy is distributed using a hierarchical frame-buffer-aligned structure for efficiency. Please note that methods in the literature that perform or improve splatting, only handle photons arriving within the camera view, whereas our approach, based on a multi-view buffer, rasterises splats at arbitrary camera sub-path nodes. This allows the capture of all photon density estimation events at any camera depth and any part of the scene.

Algorithm 1: Rasterization-based PPM

```
AllocMemory();      // Allocate geometry, photon
                    // and camera buffers
Init(); // Populate all buffers, init. PPM radii
i ← 1;
while true do
    for N_ipf iterations do
        DirectCameraHits();     // G-buffer
        DirectPhotonHits();     // Fig. 3, stage 1
        for max photon tracing depth d_photon do
            TracePhotons();     // Fig. 3, stage 2
        end
        DirectSplatting();      // Fig. 3, stage 3
        for max camera tracing depth d_eye do
            TraceImportons();   // Fig. 3, stage 4
            IndirectSplatting(); // Fig. 3, stage 5
        end
        UpdatePPMParams();      // Fig. 3, stage 6
        i ← i + 1;
    end
    if view or scene is invalidated then
        Init();
        i ← 1;
    end
end
```

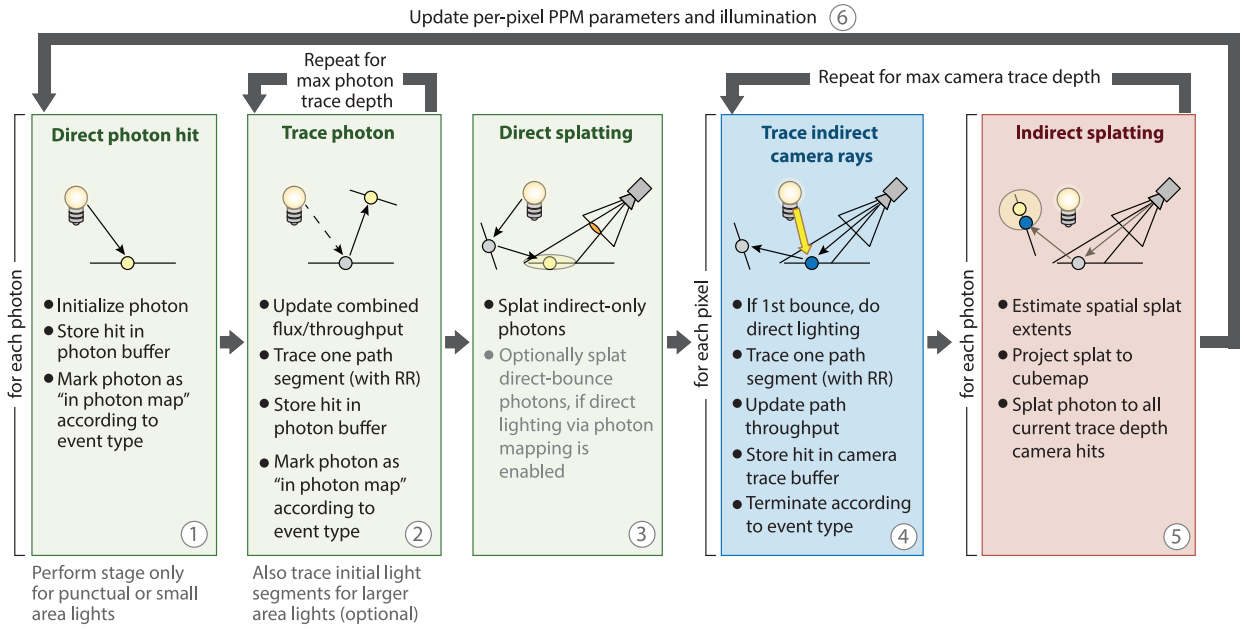

Fig. 3 The sequence of stages in each iteration of our rasterization-based probabilistic PPM.

3 Method Overview

Our probabilistic PPM approach attempts to faithfully encompass all steps in the original algorithm, capturing all L(S|G|D)*E paths, while taking advantage of rasterization stages and their output, where available. It also heavily exploits both the camera G-buffer and the multi-view image-based structure to splat photons, even when not directly visible in the camera view. A high-level outline of our PPM variant is shown in Algorithm 1 and Figure 3. The outer loop of the method is the typical (stochastic) PPM iteration, which is reset (along with the PPM parameters) each time any scene update takes place. A nested loop (not shown in the figure) performs N_{ipf} iterations of PPM, before updating the results to the output frame buffer. Note that this inner loop is only useful for animated scenes and N_{ipf} (iterations per frame) is set to 1, otherwise. For previewing animated geometry it is useful to allow PPM to converge to a better estimate, than using the initial one after the scene changes. The two inner loops correspond to photon and camera tracing iterations for arbitrary, user-defined maximum path lengths (d_{photon}, d_{eye}).

Data Structures and Memory Allocation. The memory for the image-space data structure for primitive storage is pre-allocated, similar to DIRT. In Section 5.1 we discuss the impact of the resolution of the cubemap buffers to the performance.

Next, a light tracing hit buffer (see DIRT overview above) is pre-allocated with a fixed number of entries. For tracing photons at maximum d_{photon} depth, if each photon batch contains N_p photons, shared among the light emitters, $d_{photon} \cdot N_p$ potential photon hit entries are allocated. Note that this buffer includes all light path nodes and not just the deposited photons; the presence of a photon is signified by a suitable flag in each hit record. The depth of each light path hit point is easily identified by its buffer index number and an "active" flag marks any missed photon ray intersections, which signify a discontinued path. The same flag is used when a photon path is terminated due to the Russian Roulette mechanism.

Finally, we also allocate a frame-buffer-sized *PPM buffer*, which holds the updated per-pixel data needed by progressive photon mapping, such as the current estimate of the gathered pixel radiance and r_i (see Eq. 2).

Initialisation. The camera G-buffer is prepared as usual. Shadow maps are also constructed per light source, if used in direct lighting estimation and first-bounce photon splatting. For large area lights that cannot be adequately approximated by punctual sources and shadow maps, the primary light tracing hit for direct lighting are employed instead. Alternatively, we can emit each photon batch from different locations sampled on the emitters, thus exploiting the shadow maps at the jittered points. For each pixel, the gathered radiance is zeroed and the initial per-pixel radius r_1 is set in the PPM buffer (see Sec. 4.1).

Direct Photon Hit Pass. This stage detects the first hits of the photons emitted from the light sources and records them in the photon hit buffer. Depending on

the approach used, this first bounce can use the shadow map(s) to directly determine the world position of M photons jittered on the parametric space of the shadow map using a compute shader, or trace the photons from random positions on the emitters using DIRT. While the first approach is faster, tracing the photons allows for arbitrary emission patterns and surfaces.

Photon Caching and Tracing. At each photon hit, we stochastically, determine the scattering event (D for diffuse, G for glossy, S for specular) via importance sampling. Next we choose the appropriate sampler for the next path segment and modify γ_j for the current photon j. We mark a photon hit as a stored photon on a diffuse event or if the roughness exceeds a predefined value for a glossy scattering event.

All secondary photon path segments are traced in image space using the ray traversal of DIRT and analytic ray-primitive intersections. Note that for efficiency and to sample as many paths as possible with the current photon batch, all photons keep scattering up to the maximum photon tracing depth d_{photon} or until their path is terminated according to Russian Roulette, i.e. they do not stop at the first diffuse surface.

Direct Splatting. Each photon deposited due to any indirect light tracing bounce is splatted on the camera image plane and the photon contributes to all camera pixels whose density estimation area of radius r_i around the first visible point includes the photon. These visible points can be easily obtained from the camera G-buffer, so this pass is performed before tracing any camera rays, similar to prior methods using photon splatting in the bibliography. The splat radius, which determines what primary camera hit points the photons contribute to, is determined globally and adjusted in each PPM iteration. If direct lighting via photon splatting is enabled, we also splat primary photon hits.

Camera Path Tracing and Indirect Splatting. After the direct splatting of photons, camera hit points that were not marked as density estimation points sample a new ray direction and start tracing paths using DIRT. For each new wavefront of hit points at a given tracing depth k and after deciding on the type of event (D, G, S) for each point, all currently deposited photons are splatted against the current list of non-specular camera hits at this depth. However, while in the direct photon splatting step the camera G-buffer was used for this task, here the photons are splatted on the views of the DIRT cubemap, since there is no guarantee that the density estimation points are located within the camera frustum. The resolution of the DIRT deep cubemap does not affect the quality of the photon energy distribution operation, since photons directly interact with the lists of registered camera hits in the DIRT hit map. Resolution only affects the sparsity and length of the hit record lists and thus performance (see Fig. 8).

This process is repeated for each camera bounce. Camera rays are culled using Russian Roulette and at each photon density estimation event, the estimated radiance $\hat{L}(\mathbf{x}, \omega_o)$ is multiplied with the camera path throughput and temporarily stored in the PPM buffer.

PPM Parameters Update. At the end of each PPM iteration the radius r_i is updated based on Equation 2 and so does the splat radius. Finally, the radiance estimate of the primary camera rays is incrementally averaged over the i-th iterations so far, after adding the local illumination, if computed separately.

4 Method Details

4.1 Per-pixel Bandwidth Estimation

The density radius r_1 for the importons is initialised per pixel and stored in the pixel buffer, in order to be updated in every progressive iteration. Alternatively, a global value can be set by the user. However, it is non-trivial to manually achieve a tight upper bound for the entire scene, due to depth differences, perspective and scattering of the camera paths.

We follow the un-projection strategy described in [10], where a global image-space radius scale (in pixels) s_r is given and then the corresponding per-pixel radius r_1 in world coordinates is computed by back-projecting onto the visible geometry:

$$r_1 = s_r \sqrt{\frac{4 \tan^2(\theta_{fov}/2) ||\mathbf{p} - \mathbf{c}||^2 \, (\mathbf{r} \cdot \mathbf{d}_{front})^3}{\pi h^2 |\mathbf{n} \cdot \mathbf{r}|}}, \qquad (3)$$

where $\mathbf{n}$ is the normal vector at the world-space position $\mathbf{p}$ of the visible fragment, $\mathbf{c}$ is the ray origin, $\mathbf{r}$ is the ray direction (normalised $\mathbf{p} - \mathbf{c}$). The parameter h is the height of the image in pixels, θ_{fov} the vertical aperture and $\mathbf{d}_{front}$ is the look-at camera direction. The value of s_r is typically set to 10-15 for an 1MPixel frame buffer, while r_i is computed for every pixel independently according to Equation 2.

Since the photon density estimation does not only occur at the primary camera ray hits, the radius at the k-th depth $r_i^{(k)}$ must be evaluated based on the current radius r_i. To properly determine this, the ray differentials [6] up to the hit location must be taken into account as suggested in [10]. To simplify computations and reduce propagated data per ray, we estimate the radius $r_i^{(k)}$ for the hit point encountered at depth k of the camera path by expanding the solid angle through the pixel as being scattered with no distortions (Fig. 4); we

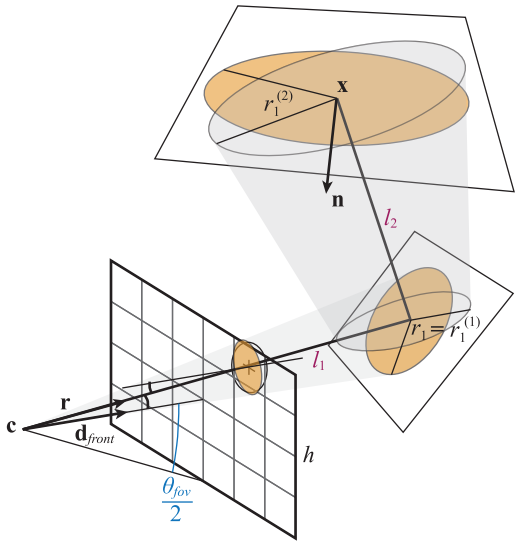

Fig. 4 Projected initial photon density estimation radius $r_1^{(k)}$ for the k-th camera bounce (k=1, 2 here). See Eq. 3, 4.

only need to keep track of the cumulative path length including the k-level segment and the primary hit distance $l_0 = ||\mathbf{p} - \mathbf{c}||$:

$$r_i^{(k)} = \frac{r_i}{l_1} \sum_{l=1}^{k} l_k, \; k > 1. \tag{4}$$

Knaus et al. [10] also set a minimum and a maximum world-space radius r_{min}, r_{max} to avoid corner cases. We respect the same limits, which are reduced by the same shrinkage ratio in each iteration. Maintaining r_{max} is important to our implementation, as we set the photon splat radius equal to it, so that the splatted photons are guaranteed to contribute to the correct importons, whose radius is $r_{min} \leq r_i^{(k)} \leq r_{max}$. This serves as an upper bound for the splat radius but there is no other way to estimate it without actually querying the nearest importons, an operation that should be clearly avoided.

4.2 Photon Splatting

As mentioned in Section 3, photon splatting uses either the camera view or the cubemap side of DIRT that the photon is projected on (Fig. 6a), depending on whether the splatting corresponds to the primary camera rays (direct splatting - Fig. 6b) or a secondary camera path segment (Fig. 6c). The splatting mechanism is almost identical in the two cases: a quad centred at the photon hit location is drawn using a geometry shader and the list of camera hit points associated with the current view is iteratively processed to locate hit points

Fig. 5 Fragment reduction with Russian Roulette (RR) during splatting, trading variance for a substantial decrease in both direct and indirect splatting times.

whose r_i include the photon location. In the case of direct splatting, the process is simplified as the generated splat fragment directly maps to the unprojected visible point in the G-buffer and no search for candidate hit points is needed. In the indirect splatting, in order to determine which DIRT view the photons must attempt to splat on, the latter are culled on each frustum, according to their position and splat radius.

Primarily to accelerate initial iterations, where the splat radius is large, we optionally introduce a Russian Roulette splat fragment rejection mechanism with probability 0.5. This effectively introduces some variance but shortens splatting times about 20%-30% for indirect splatting and 40%-50% for direct splatting (Fig. 5).

The splat radius is globally decreased in each iteration according to the rate of Equation 2. This avoids wasting GPU computations and decreases splatting cost over time. It is also possible to employ a better splat radius estimation, as proposed by Frisvad et al. [1].

4.3 Light Sources

Our method can support multiple light sources by splitting the number of photons M according to user-defined balance ratios. Even though typical automatic heuristics could be used, their study is outside the scope of this paper. For omnidirectional light sources we prefer to compute direct lighting via photon tracing instead of cubemaps or paraboloid maps, since the additional passes and memory cost are hardly justified, while exhibiting the typical shadow map artefacts. The Bathroom, Pool and Fireplace scenes in Figure 7 are rendered this way.

5 Evaluation

We ran experiments with scene complexity ranging from a few thousand to over a million triangles (Table 1) and photon batches of 262K photons, unless otherwise stated. For most scenes, we allowed a large number of

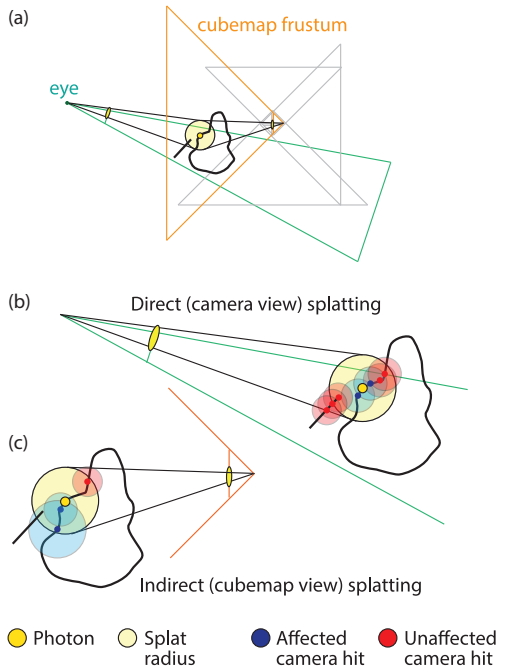

Fig. 6 Photon Splatting. (a) Photon projection on camera and cubemap frusta for direct and indirect splatting, respectively. Below, details showing the splatting on the (dense) primary camera path hits (b) and the indirectly traced importons on a DIRT ADS view (c).

light-to-eye path segments (≥ 3) in order to capture all intricate global illumination effects and also stress-test our method. Our most elaborate examples are presented in Figure 7.

All experiments were performed on an NVIDIA RTX 2080 Ti graphics card with 12GB of video memory. The host system is irrelevant since all computations are executed on the GPU. All scenes include LS+D(D|G|S)*E paths to some degree so that caustics and specular (indirect) shadows could be formed. We avoided including large open environments, as the photon mapping algorithm is not the most suitable approach for them. Finally, unless otherwise stated, all our test scenes are evaluated at 1MPixel resolution except from the Fireplace scene, which is rendered at 2MPixel.

In order to maximise ray tracing performance on the DIRT ADS, we have evaluated the impact of the cubemap resolution and have tuned the number of bins per buffer cell independently for each scene between 12 and 16.

5.1 Memory Usage

Memory consumption is directly affected by the cubemap resolution, however the smaller the resolution, the higher the number of primitives per cell. This in turn, translates to more ray-triangle intersections and longer list traversals for the fetching of primitive shading properties (as indicated by the authors of the DIRT method). Figure 8 clearly demonstrates this behaviour and also shows that choosing a resolution of 256^2 for the cubemap buffers is generally a good trade-off between memory consumption and ray tracing speed, while for higher performance a resolution of 512^2 is preferred, at the expense of an increased memory total cost. Going to a finer discretisation not only increases the memory excessively, but also hinders traversal due to longer ray marching and too many hierarchical level hops, which amplifies incoherent memory accesses.

It is worth mentioning that the proposed method, contributes zero memory overhead to the total ADS memory consumption pool, since the importon storage is implicitly handled by the DIRT update chain in every iteration. The total size of the PPM buffers, i.e. photon buffer and PPM parameters as well as radiance estimate, depends on the image resolution R and the number of photons in a batch M, times the number of light tracing segments L ($L = 1$ indicates direct photon hits). Specifically, this equals to 16 bytes per pixel radiance and radius estimate, an equal size intermediate buffer to accumulate splat estimations and 64 bytes per photon entry, which yields a total cost of $32R + 64ML$ bytes of GPU memory.

5.2 Performance

Figure 9 shows a breakdown of the rendering time per iteration for three different scenes, while Figure 10 presents average rendering times on the pool scene with different eye and light path lengths. We observe that two factors predominately affect the performance of the method: i) the number of fragments generated during the indirect photon splatting, which in turn corresponds to excessive searches in the hit buffer for scattering events from focused L(S*)D(S+)E paths and ii) the number of secondary rays, for both the camera and light paths. On the other hand, even though direct splatting may generate a vast number of fragments, it does not incur any significant performance hit, as these splats are resolved using only the camera G-buffer. Primary ray generation from both the camera and light sources can be fully optimised through rasterization with minimal overhead.

Another notable factor impacting performance is the splat radius (bandwidth) for the indirect splatting, as for each reconstructed location corresponding to a splat fragment, the entire list of camera path hits in the same view frustum of the cubemap must be traversed. Large splat radii increase the rendering time but the

Fig. 7 Test scenes used for the experimental validation of our method, encompassing complex light transport events. Numbers in the parentheses indicate the light and eye path lengths respectively. From top left to bottom right, both low- and high-frequency illumination effects are accurately captured on Bathroom (4+4), Fireplace (5+5), Bunny (2+2), Pool (4+4), Sponza (3+3), Ring (3+3), and Glass (5+5) scenes (see Tab. 1 for details).

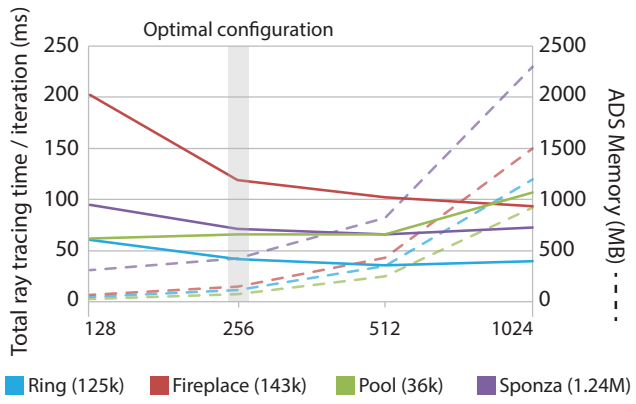

Fig. 8 Memory consumption and ray tracing time versus cubemap image resolution, for constant photon batch size. The numbers in parentheses denote number of polygons.

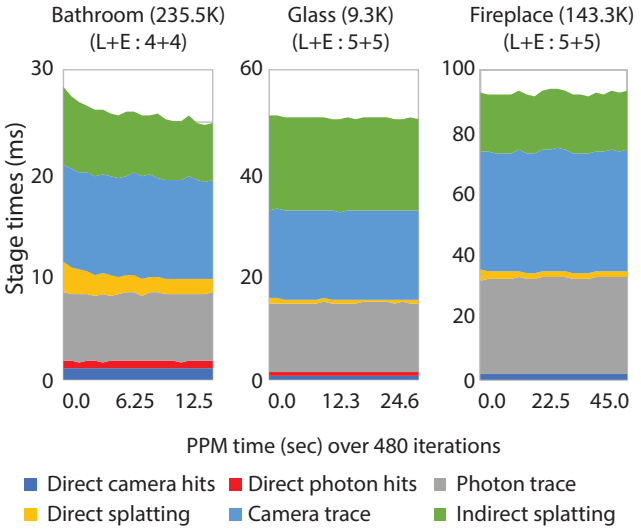

Fig. 9 Individual stage times for the first 480 iterations of our method on three scenes of different geometric and PPM complexity.

iterative bandwidth reduction (Eq. 2) and the hard radius limit both guarantee that the overall performance rapidly stabilises to a good level (Fig. 9).

We also evaluate the performance of our method against a typical general-purpose GPU implementation of the PPM pipeline using the NVIDIA OptiX framework for ray tracing and CUDA FLANN [19] for GPU-accelerated nearest neighbours search of particles (Fig. 11). In every frame, we construct a new CUDA kd-tree for the photon particles, with buffers shared with OptiX in order to have zero data transactions with the host system. In our experiments, every kd-tree is initialised with leaf size equal to 64, the sorted output of each query is disabled and the number of maximum leaves to be visited during traversal is set to unlimited. Since CUDA FLANN requires pre-allocated GPU

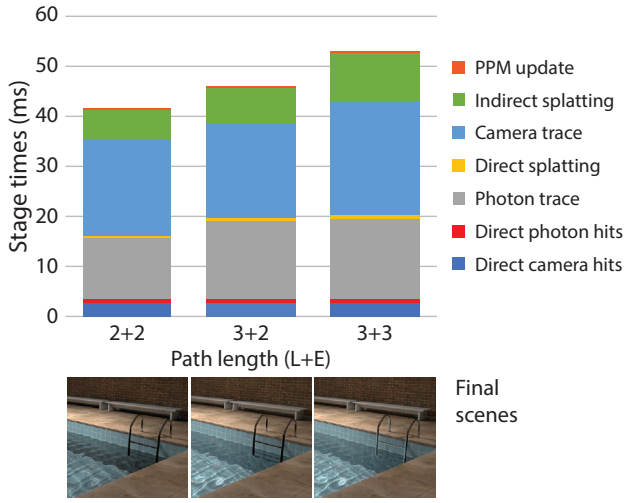

Fig. 10 Individual stage execution times for the pool scene under different number of (maximum) path lengths, averaged over 1000 PPM iterations. 2 MPixel image buffer.

buffer pointers as input to the gathering phase, we invoke the radius search for a fixed maximum capacity of 100 photons, a parameter we found optimal between quality and performance.

Despite the comparatively slower ray traversal of the implemented DIRT ADS compared to the optimised OptiX ray tracing times, our method has a significantly better overall performance due to two important factors: First, our PPM variant dispenses with the particle ADS construction in each frame, which in contrast, burdens the OptiX-CUDA FLANN implementation (see Tab. 1). Second, the introduction of photon splatting instead of gathering at *all* camera path depths inherently outperforms a tree search due to its trivially parallel fragment-based execution. During the first 50 frames, the measured times were approximately 5% and 3% higher than the reported average for our method and the OptiX-CUDA FLANN implementation, respectively.

Finally, we measured the performance of our method against the OptiX-CUDA FLANN implementation for different photon batch sizes (Fig. 12) and different resolutions up to 4K (Fig. 13), on scenes with different photon and camera path distributions. As the photon batch size increases, the image-space photon tracing takes a significant part of the overall PPM cycle. However, for the corresponding number of photons, the FLANN ADS build time and gathering also has a significant overhead, resulting in noticeably higher cycle times, in all cases. The same behaviour was observed, when increasing the frame buffer resolution, with any gain in camera path tracing times being overshadowed by the queries in the OptiX-CUDA FLANN implementation.

Table 1 ADS construction time for the example scenes using a cubemap side resolution of 512^2 cells for our method vs the OptiX-CUDA FLANN implementation.

Scene	Primitives	ADS Construction (ms) Our method (Image-space)	OptiX-CUDA FLANN (BVH + kd-tree)
Glass	9.3K	1.5	76.0 (2.9 + 73.1)
Pool	35.7K	1.6	48.0 (3.6 + 44.4)
Bunny	72.9K	4.0	57.0 (7.5 + 49.5)
Ring	124.8K	2.8	41.8 (5.1 + 36.7)
Bathroom	235.5K	5.0	73.6 (5.0 + 68.6)
Fireplace	143.3K	5.5	66.1 (4.3 + 61.8)
Sponza	1,249.3K	11.1	84.2 (23.7 + 60.5)

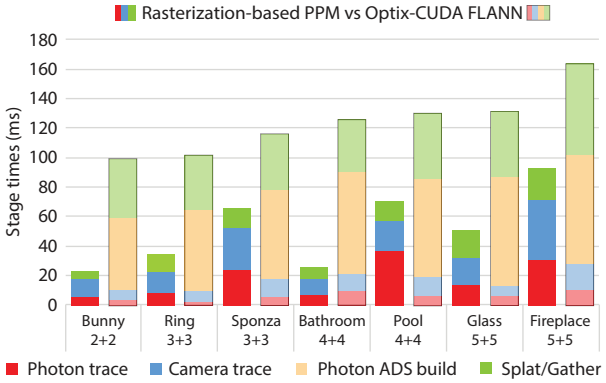

Fig. 11 Timing comparison between our PPM variant and OptiX-CUDA FLANN for the test scenes of Figure 7.

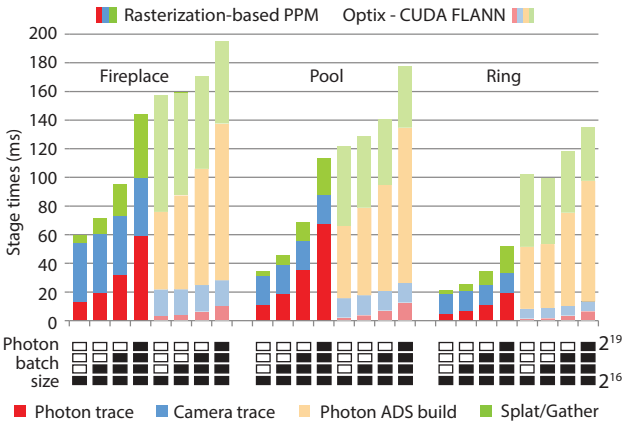

Fig. 12 Impact of photon batch size on PPM cycle timings. Comparison between our method and the OptiX-CUDA FLANN implementation for three scenes from Fig. 7 with different photon distributions.

5.3 Quality

Since our method uses analytical intersection tests and either full photon energy deposition or a statistically compensated one (Russian Roulette), it has no impact on the quality of the converged frames, as illustrated in Figure 14 - bottom row. Furthermore, due to the

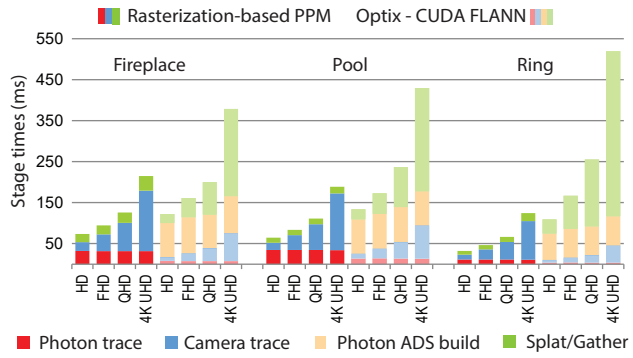

Fig. 13 Impact of frame buffer resolution on PPM cycle timings. Comparison between our method and the OptiX-CUDA FLANN implementation for three scenes from Fig. 7.

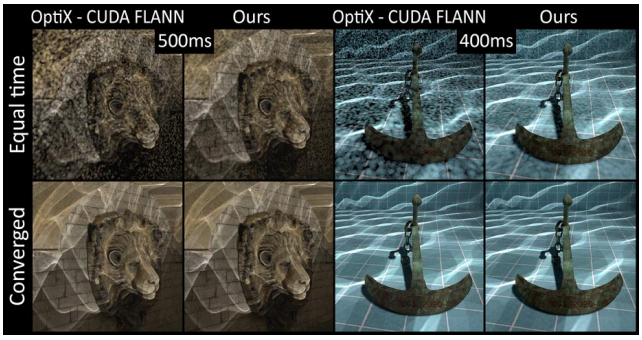

Fig. 14 Quality comparison for equal time (top) and converged results (bottom) between our method and the OptiX-CUDA FLANN implementation for the Sponza (left) and Pool (right) scenes.

faster PPM cycles, our method delivers higher quality for equal rendering time (Fig. 14 - top).

5.4 Dynamic Geometry and Animation

In Figure 15, two animation examples are shown, one where the geometry is procedurally displaced via a compute shader (Pool scene, left) and one, where only the camera and light positions change (Bunny scene, right). When previewing animations, we can set the number of iterations per frame to $N_{ipf} > 1$ so that PPM can perform a few cycles prior to displaying each frame. In these particular examples we use $N_{ipf} = 2$. Since typical construction times for the DIRT ADS fall way below 11ms, as shown in Table 1, the ADS can be completely rebuilt in every frame.

In general, the complete rebuild of the ADS does not significantly impact the rendering times, therefore the method is well-suited to dynamic content, with the performance having minimal correlation to the extent or nature of the update.

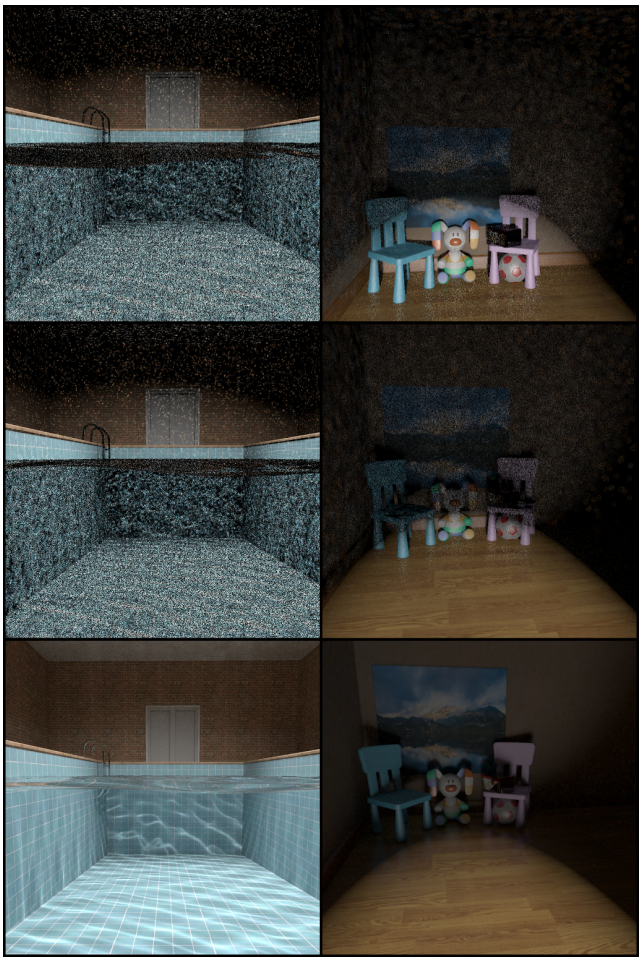

Fig. 15 Animation examples showing caustics from a procedurally animated water surface (left) and specular shadows from a moving spotlight (right). The 1MPixel frames were rendered at 80ms and 48ms respectively, with 2+2 path lengths and $N_{ipf} = 2$. The bottom insets correspond to convergence after 1000 iterations.

6 Conclusions

We have presented an adaptation of the probabilistic progressive photon mapping method for the GPU, by reordering camera and light path computations so that we can take advantage of splatting for all hits on the camera paths, even those that are not visible to the camera frustum. We employ rasterization for both the splatting operations at all camera trace levels and the construction of the ADS for the ray tracing, taking full advantage of the GPUs hardware. We compare our method against a typical general-purpose GPU photon mapping implementation, performing both ray tracing and gathering operations on the latter, and show how the splatting at camera path diffuse events at any possible depth can vastly accelerate the density estimation procedure.

One limitation of our method is the dependence of the performance on the positioning of the cubemap in the scene. Although the projective cell density of the DIRT ADS provides opportunity for performance increase, when placed near a light source or the camera, it can be difficult to tune in certain cases, leading to a potential ray tracing performance degradation of approximately $10-20\%$. For example, in scenes with multiple equivalent light sources power-wise, placing the DIRT ADS over a single emitter, may lead to sub-optimal ray traversal performance for the rest.

Acknowledgements

The Bathroom scene was based on a model from `https://www.cgtrader.com/` and the Fireplace and Sponza Atrium models were downloaded from McGuire's Computer Graphics Archive [15]. The remaining scenes were created by the authors.

References

1. Frisvad, J.R., Schjøth, L., Erleben, K., Sporring, J.: Photon Differential Splatting for Rendering Caustics. Computer Graphics Forum **33**(6), 252–263 (2014)
2. Guntury, S., Narayanan, P.J.: Raytracing Dynamic Scenes on the GPU Using Grids. IEEE Transactions on Visualization and Computer Graphics **18**(1), 5–16 (2012)
3. Hachisuka, T., Jensen, H.W.: Stochastic Progressive Photon Mapping. ACM Trans. Graph. **28**(5), 1—-8 (2009)
4. Hachisuka, T., Ogaki, S., Jensen, H.W.: Progressive Photon Mapping. ACM Trans. Graph. **27**(5) (2008)
5. Hu, W., Huang, Y., Zhang, F., Yuan, G., Li, W.: Ray tracing via GPU rasterization. The Visual Computer **30**(6-8), 697–706 (2014)
6. Igehy, H.: Tracing Ray Differentials. In: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '99, pp. 179—186. ACM Press/Addison-Wesley Publishing Co., USA (1999)
7. Jensen, H.W.: Global Illumination Using Photon Maps. In: Proceedings of the Eurographics Workshop on Rendering Techniques '96, pp. 21—-30. Springer-Verlag, Berlin, Heidelberg (1996)
8. Jensen, H.W.: Realistic Image Synthesis Using Photon Mapping. A. K. Peters, Ltd., USA (2001)
9. Kalojanov, J., Billeter, M., Slusallek, P.: Two-Level Grids for Ray Tracing on GPUs. Computer Graphics Forum **30**(2), 307–314 (2011)
10. Knaus, C., Zwicker, M.: Progressive Photon Mapping: A Probabilistic Approach. ACM Trans. Graph. **30**(3) (2011)
11. Li, S., Simons, L., Pakaravoor, J.B., Abbasinejad, F., Owens, J.D., Amenta, N.: kANN on the GPU with Shifted Sorting. In: Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics. The Eurographics Association (2012)
12. Ma, V.C.H., McCool, M.D.: Low Latency Photon Mapping Using Block Hashing. In: SIGGRAPH/Eurographics Workshop on Graphics Hardware. The Eurographics Association (2002)
13. Mara, M., Luebke, D., 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, I3D '13, pp. 71—78. Association for Computing Machinery, New York, NY, USA (2013)
14. Mara, M., McGuire, M., Nowrouzezahrai, D., Luebke, D.: Deep G-Buffers for Stable Global Illumination Approximation. In: Proceedings of High Performance Graphics, HPG '16, pp. 87—-98. Eurographics Association (2016)
15. McGuire, M.: Computer graphics archive (2017). URL `https://casual-effects.com/data`
16. McGuire, M., Luebke, D.: Hardware-Accelerated Global Illumination by Image Space Photon Mapping. In: Proceedings of the Conference on High Performance Graphics 2009, HPG '09, pp. 77–89. Association for Computing Machinery, New York, NY, USA (2009)
17. McGuire, M., Mara, M.: Efficient GPU Screen-Space Ray Tracing. Journal of Computer Graphics Techniques (JCGT) **3**(4), 73–85 (2014)
18. Moreau, P., Sintorn, E., Kämpe, V., Assarsson, U., Doggett, M.: Photon Splatting Using a View-Sample Cluster Hierarchy. In: Proceedings of High Performance Graphics, HPG '16, pp. 75—-85. Eurographics Association, Goslar, DEU (2016)
19. Muja, M., Lowe, D.G.: Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In: VISAPP (1), pp. 331–340. INSTICC Press (2009)
20. Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., Stich, M.: OptiX: A General Purpose Ray Tracing Engine. ACM Trans. Graph. **29**(4), 66:1–66:13 (2010)
21. Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., Hanrahan, P.: Photon Mapping on Programmable Graphics Hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '03, pp. 41–50. Eurographics Association, Goslar, DEU (2003)
22. Sousa, T., Kasyan, N., Schulz, N.: Secrets of CryEngine 3 Graphics Technology. In: ACM SIGGRAPH Talks (2011)
23. Stürzlinger, W., Bastos, R.: Interactive Rendering of Globally Illuminated Glossy Scenes. In: Rendering Techniques '97, pp. 93–102. Springer Vienna, Vienna (1997)
24. Uludag, Y.: Hi-Z screen-space cone-traced reflections. In: GPU Pro 5, pp. 149–192. CRC Press (2014)
25. Vardis, K., Vasilakis, A.A., Papaioannou, G.: A Multi-view and Multilayer Approach for Interactive Ray Tracing. In: Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '16, pp. 171—-178. Association for Computing Machinery, New York, NY, USA (2016)
26. Vardis, K., Vasilakis, A.A., Papaioannou, G.: DIRT: Deferred Image-based Ray Tracing. In: Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics. The Eurographics Association (2016)
27. Vinkler, M., Havran, V., Bittner, J.: Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing. Computer Graphics Forum **35**(8), 68–79 (2016)
28. Weiss, M., Grosch, T.: Stochastic Progressive Photon Mapping for Dynamic Scenes. Computer Graphics Forum **31**(2pt3), 719–726 (2012)
29. Yao, C., Wang, B., Chan, B., Yong, J., Paul, J.C.: Multi-Image Based Photon Tracing for Interactive Global Illumination of Dynamic Scenes. Computer Graphics Forum **29**(4), 1315–1324 (2010)