Accelerating Photon Mapping for Hardware-Based Ray Tracing

René Kern
TU Clausthal

Felix Brüll
TU Clausthal

Thorsten Grosch
TU Clausthal

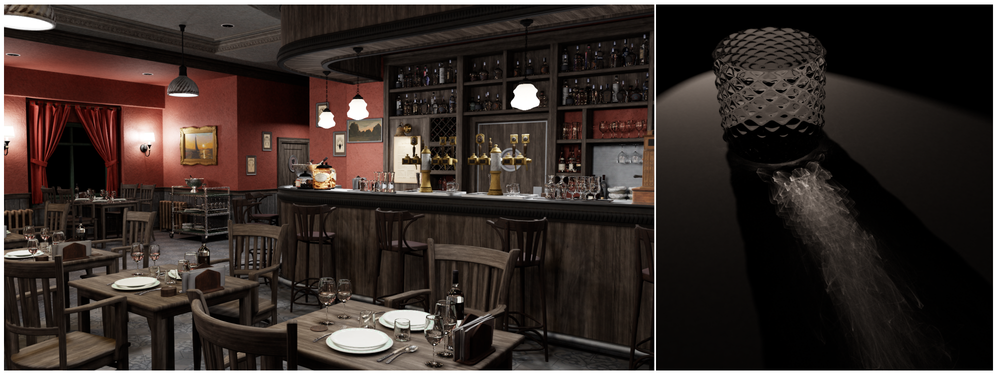

Figure 1. Two scenes rendered in less than a minute with our hardware ray traced photon mapper. Two million photons are distributed per iteration in both scenes. Left: Bistro (30 s, 2304 it, $\sim$13 ms per iteration). Right: Caustic Glass (2 s, 193 it, $\sim$10.4 ms per iteration). Times were captured with an NVIDIA GeForce RTX 2080 SUPER.

Abstract

Photon mapping is a numerical solution to the rendering equation based on tracing random rays. It works by first tracing the paths of the emitted light (photons) and storing them in a data structure to then collect them later from the perspective of the camera. Especially, the collection step is difficult to realize on the GPU due to its sequential nature. We present an implementation of a progressive photon mapper for ray tracing hardware (RTPM) based on a combination of existing techniques. Additionally, we present two small novel techniques that speed up RTPM even further by reducing the number of photons stored and evaluated. We demonstrate that RTPM outperforms existing hash-based photon mappers, especially on large and complex scenes.

1. Introduction

One popular approximate solution to the rendering equation [Kajiya 1986] is photon mapping [Jensen 2001], which is based on ray tracing and Monte Carlo integration. Other popular algorithms to approximate the rendering solution are path tracing [Kajiya 1986], bidirectional path tracing [Lafortune and Willems 1993], metropolis light transport [Veach and Guibas 1997], and vertex connection and merging [Georgiev et al. 2012], which is a combination of photon mapping and bidirectional path tracing. In contrast to these techniques, photon mapping [Jensen 2001] especially excels in rendering patches of light called caustics.

Photon mapping is a two-pass algorithm that is split into photon generation and collection phases. Photons are distributed from light sources and ray-traced through the scene for photon generation. Hits on diffuse surfaces are stored in a photon map. In the collection step, the scene is traced from the point of the camera where all photons within a given radius that hit a diffuse surface are collected.

The improved graphics hardware that now allows for hardware-supported ray tracing has sped up many of those techniques. This can be used directly to speed up the gathering of photons. With fast radius search [Evangelou et al. 2021], it was shown that the collection step can be sped up by exploiting the ray tracing API by storing the photons inside a ray tracing acceleration structure and collecting them via ray tracing. The performance of the new collection method was compared to a k-d tree framework with the result that the hardware ray tracing accelerated collection is faster.

We describe a progressive photon mapper optimized for ray tracing hardware that uses the idea of fast radius search [Evangelou et al. 2021] to collect the photons. In addition, we introduce two novel techniques that speed up the progressive photon mapper even further. One is *Photon Culling*, where photons outside our camera view are not stored, reducing the acceleration structure build time for large and complex scenes, especially when large parts of the scene are not visible from the camera view. The other is *Stochastic Evaluation*, where we only evaluate a small number of photons stochastically to reduce the number of lighting calculations. This speeds up the photon collection process for areas with higher photon densities, which often occur with larger photon radii. We compare our photon mapper against two state-of-the-art GPU hash techniques [Mara et al. 2013; Hachisuka and Jensen 2010] on various scenes.

2. Related Work

Photon mapping [Jensen 2001] uses two passes to approximate global illumination by using photon maps. In the first pass, packets of energy (photons) are emitted from the light sources, which are stored in two photon maps on every diffuse surface hit within the scene. One photon map stores caustic photons, which are reflected or transmitted by specular surfaces before hitting the first diffuse surface. This is equivalent to LS+D paths, where L is the light source, S is a specular hit, and D is a diffuse hit. All other photons are stored in the global photon map. A smaller radius is usually used for the caustic photon map as the density of photons is generally higher. The second pass renders the scene using path tracing and both photon maps. The caustic photon map is directly evaluated with a radiance estimate on the first (diffuse) hit. The global photons are evaluated using final gathering, where the photon map is only evaluated for indirect lighting. A path tracer is used to calculate the direct lighting on the first hit and determine the second (diffuse) hit, where the global photon map is evaluated. The radiance estimate can also be used for the global photons to get a fast preview of the final image. However, photon mapping is biased, as it retains the same search radius for photons for the whole pass, which leads to a smoothing effect. An infinite number of photons with an infinitely small radius would have to be used to get an unbiased result.

Progressive photon mapping [Hachisuka et al. 2008] and stochastic progressive photon mapping [Hachisuka and Jensen 2009] solve that problem by reformulating the photon mapping algorithm to an iterative one. With this, the problem is decoupled from its memory bottleneck, and better results can be achieved as the image converges to a consistent result. Depending on the number of photons that were collected for one pixel, the collection radius is reduced for the upcoming iterations. Knaus and Zwicker [2011] showed that the radius reduction can be done by a fixed sequence instead of depending on the number of photons collected.

With parallel progressive photon mapping, Hachisuka and Jensen [2010] presented a fast GPU implementation for photon mapping by stochastically storing only one photon for one hash cell. This avoids creating and searching through a list of hash entries, which does not work well in parallel.

In 2013, Mara et al. [2013] introduced four photon mapping algorithms that work well on a GPU. Two of them use photons as a bounded geometry and render them into the scene. For the other two, a hash grid and a frustum-based tile grid are used to accelerate the photon collection step.

Caustics using screen-space photon mapping [Kim 2019] and real-time ray traced caustics [Yang and Ouyang 2021] both introduced a real-time caustic rendering technique using hardware-supported ray tracing. The hardware ray tracing API is used to trace the photons while only storing the caustic photons. Both use different screen-space techniques to project the caustic photons to the final image.

Fast radius search [Evangelou et al. 2021] introduced a way to exploit the hardware ray tracing API to accelerate the radius search operation. Instead of searching for all points in a radius around the query, the operation is reversed so that the radius is mapped to each search point instead. A query then collects all search points where the query is inside the radius of a search point. A ray tracing acceleration structure is built with the search point and a small ray from the query is then used to collect all search points.

3. Hardware Ray Tracing Photon Mapper

In this article, we present implementation details for the hardware ray tracing–based progressive photon mapper (RTPM). Section 3.1 outlines the fundamentals needed. Sections 3.2 to 3.6 explore our implementation in detail. In Sections 3.7 and 3.8, we present our two novel speedup techniques for our RTPM: Photon Culling and Stochastic Evaluation.

3.1. Fundamentals

RTPM is based on photon mapping [Jensen 2001], which can approximate the global illumination of a scene and excels at rendering caustics. First, the scene is traced from light sources, with every ray representing a photon. Photons are only stored in photon maps if they hit diffuse surfaces. In the second pass, photons are collected from the perspective of the camera. With a photon map, the exitant radiance for a location $\vec{x}$ on a surface can be estimated with

$$L(\vec{x},\hat{\omega}) \approx \frac{1}{\pi r^2}\sum_{p=1}^{N} f_r(\vec{x},\hat{\omega},\hat{\omega}_p)\Phi_p(\vec{x},\hat{\omega}_p), \tag{1}$$

where N is the number of photons inside the radius r. For every photon in the radius, the flux Φ_p is multiplied with the BRDF f_r, which depends on the point $\vec{x}$, view direction $\hat{\omega}$, and photon direction $\hat{\omega}_p$. The sum of reflected photon flux values is divided by the circular area in which photons are collected.

As it is not feasible to emit and collect all photons in one iteration on the GPU, our algorithm is based on stochastic progressive photon mapping (SPPM) [Hachisuka and Jensen 2009], which is an extension of progressive photon mapping (PPM) [Hachisuka et al. 2008]. In PPM the emission and collection steps are done iteratively with a shrinking radius, depending on the number of photons collected. The collection points from the second pass of the original photon mapping stay the same throughout the whole algorithm. In SPPM these collection points are traced anew for every iteration. Knaus and Zwicker [2011] showed that a fixed sequence (Equation (2)) can be used for radius reduction instead of reducing the radius for every collection point depending on the number of photons collected:

$$r_{i+1} = r_i\sqrt{(i+\alpha)/(i+1)}. \tag{2}$$

Here, r is the radius, i is the current iteration, and α is a user parameter between 0 and 1. Kaplanyan and Dachsbacher [2013] proved that an α value of $2/3$ is optimal. The radius reduction by Knaus and Zwicker [2011] is used in RTPM because the

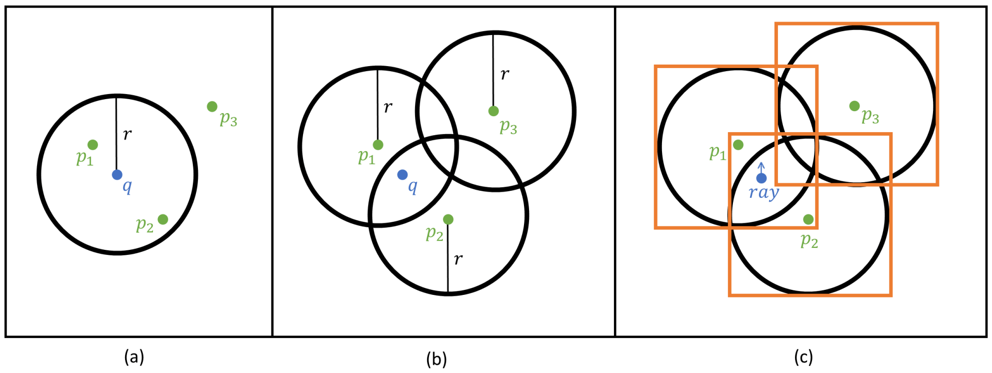

Figure 2. (a) A radius search operation, where the query q searches for all points p in the radius r. (b) The inverse radius search where the radius r is mapped to each point p instead of the query q. All points that have the query within their radius are collected. (c) How the radius inverse search is implemented for a ray tracing operation. The spheres are stored as AABBs and for the search query, an infinitely small ray is used. All spheres that intersect with the ray are collected.

radius for caustic and global photons can be stored globally in contrast to storing the radius for every photon. Additionally, the radius reduction used in PPM [Hachisuka et al. 2008] and SPPM [Hachisuka and Jensen 2009] would not be feasible, which is further explored in Section 3.2.

In RTPM, photons are collected with small rays using the inverse radius search operation [Evangelou et al. 2021]. Figure 2 shows how inverse radius search works by deriving it from the regular radius search. To guarantee that each AABB from Figure 2(c) is only called once, a flag needs to be set in the geometry description during the acceleration structure generation, telling the API not to invoke the any-hit shader multiple times.

3.2. Overview

To optimize the progressive photon mapping algorithm [Hachisuka et al. 2008; Knaus and Zwicker 2011; Hachisuka and Jensen 2009] for ray tracing hardware, we reorganized the algorithm and adjusted some parts to be more fitting for the hardware. A general overview of the algorithm can be seen in Figure 3.

Using the camera viewpoints instead of photons for the search operation is not practical for a GPU implementation. This is mainly because of data race, as multiple photons could want to contribute to the same camera point at the same time. This could be resolved with locks, but the number of blocking threads would hurt performance immensely. By inverting the search operation, it remains lock-free as photons are only read on collection. To still get the benefit of culling photons outside of the camera pixels, we introduced our novel technique Photon Culling, which is presented in Section 3.7.

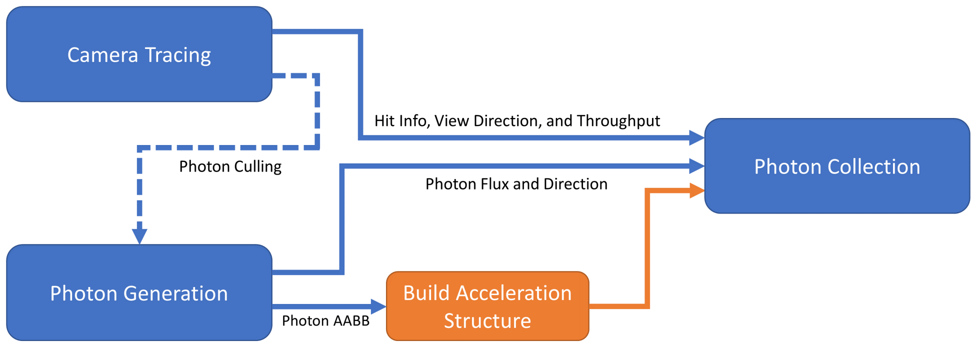
Camera Tracing
Hit Info, View Direction, and Throughput
Photon Collection
Photon Culling
Photon Flux and Direction
Photon Generation
Photon AABB
Build Acceleration Structure

Figure 3. Overview for RTPM. The Camera Tracing pass (Section 3.3) generates information about the first diffuse surface hit. In the Photon Generation pass (Section 3.4) the photons are traced from the light sources through the scene. Upon hitting a diffuse surface, photons are stored in multiple buffers (Section 3.5). The information of both passes is used in the Photon Collection pass (Section 3.6) to approximate the global illumination for the scene. When using our novel technique Photon Culling (Section 3.7), a dependency between the Camera Tracing and Photon Generation passes is created. The camera position is used to create a hash grid, which is used to cull all photons that would not be collected in this iteration.

The radius reduction (Equation (2)) by Knaus and Zwicker [2011] optimizes performance for the inverted search as the radius of the photon is now independent of the hit surfaces. It would still be possible to perform the inverse radius search with the radius reduction from stochastic progressive photon mapping. However, the photon radius would need to be set to the global maximum for that to work.

Other than in stochastic progressive photon mapping, the tracing of camera rays and the collection step are split into two separate passes, resulting in three passes. This increases shader performance because it is not determined when the first diffuse surface is hit. This can lead to many threads waiting before the collection rays can be used. Additionally, this split is needed for the Photon Culling (Section 3.7).

3.3. Camera Tracing

We need information about the first diffuse surface hit from the camera perspective to collect the photons. A modified G-buffer [Hargreaves 2004] or V-buffer [Burns and Hunt 2013] can be used. In our implementation, we used a V-buffer as it has a lower bandwidth than the G-buffer. Suppose that the scene contains highly specular or transparent materials. A standard G- or V-buffer that stops on the first hit is insufficient, as photons are only collected on diffuse surfaces. To get the information needed in the Photon Collection pass, all paths must be traced until they hit a diffuse surface (or reach the recursion limit, in which case they are invalid). To determine if a diffuse surface was hit, on every valid hit point, a new direction is sampled with the

Field	Format	Bytes
V-Buffer	RGBA32Uint	16
View Direction	RGB[A]32F	16
Throughput	RGB[A]32F	16

Table 1. Outputs for the Camera Tracing pass. We store all components with full precision as it is only generated once per frame. The V-buffer needs 16 bytes. For a triangle hit, 8 bytes are header (instanceID, primitiveID) and 8 bytes are barycentric coordinates with 4 bytes each.

```
void rayGeneration(){
    Ray ray = getCameraRay(); //Initialize the ray.
    rayPayload.throughput = float3(1.0,1.0,1.0); //Initialize throughput with 1.
    for(uint i = 0; i< MaxRecursion; i++){
        //Trace the ray through the scene. The hit is handled in closestHit(...) function.
        TraceRay(sceneAS, ray, rayPayload); //sceneAS is the acceleration structure for the scene.
        if(rayPayload.terminate) return; //Terminate on miss or diffuse surface hit.
        ray = rayPayload.ray; //Update ray information for next ray tracing operation.
    }
}
//Gets called for the closest hit from the TraceRay function
void closestHit(RayPayload rayPayload, Attributes hitAttributes){
    //Get the hit attributes from build in functions/variables.
    HitInfo hitInfo = getHitInfo(InstanceIndex(), PrimitiveIndex(), hitAttributes);
    Sample bsdfSample = sampleBSDFNewDirection(hitInfo); //Sample a new direction with BSDF.
    //On diffuse surface hit, write viewDirection, throughput, and hitinfo
    //into the G- or V-buffer at the current pixel.
    if(bsdfSample.isDiffuseReflection()){
        writeHit(DispatchRaysIndex().xy, WorldRayDirection(), rayPayload.throughput, hitInfo);
        rayPayload.terminate = true;
    }
    //Update info for next iteration.
    rayPayload.throughput *= bsdfSample.throughput;
    rayPayload.ray = Ray(hitInfo.getPosition(), bsdfSample.direction);
}
```

Listing 1. HLSL pseudocode for the Camera Tracing pass. The camera path is traced until a diffuse surface is hit. On non-diffuse reflection or refraction, the throughput and view direction are updated. If the ray misses the scene, it is terminated. The cyan-colored functions are built-in HLSL functions.

BSDF. We stop if the sampled direction results from a diffuse reflection. For a specular reflection, we trace the path further. In addition to the hit data of a diffuse surface, the incoming view direction of the surface and throughput are needed because they cannot be reconstructed from camera data alone. This results in a total of 48 bytes needed, as can be seen in Table 1. The pseudocode for the Camera Tracing pass is shown in Listing 1.

3.4. Photon Generation

The photon tracing pass is similar to the original photon mapping [Jensen 2001]. A number N of photons are generated from light sources and are traced through the scene. Photons should only be stored if they hit diffuse surfaces. On specular or transparent surfaces, most photons would contribute nothing to the exitant radiance because specular and transparent surfaces only have small reflection lobes. The probability of photons coming from the direction of the small lobes is very small (and zero for perfectly specular materials). Therefore, it is better to store photons only for diffuse surfaces and trace the specular photon paths.

After storing a photon, Russian roulette is used before tracing the next bounce to terminate photon paths with a low contribution beforehand. The average throughput of the photon is used as the probability for Russian roulette.

Because the photon map uses the same radius for the whole scene, caustics can appear blurred if the radius is too large. To get sharp caustics, the photon map can be split into two: a caustic photon map for the LS+D path photons and a global photon map for all other photon paths that end on diffuse surfaces. A smaller radius should be used for the caustic map because caustic photons often land close to each other. Both photon maps use a different initial radius that is reduced after every iteration separately with Equation (2).

3.5. Storing Photons

On a diffuse surface hit, a photon is stored in either the caustic or global photon map, depending on the surface that was hit beforehand. Caustic photons are always stored, whereas a rejection step is used for global photons to reduce the number of photons stored. In the rejection step, a photon is rejected with a user-defined probability. Photons that pass the test are weighted up by the rejection probability. Even when not passing the test, the photon path is traced further. With the rejection step, we can trace more photons to get more caustic photons while retaining similar acceleration structure building and collect times by not storing all global photons. We used a rejection probability of 70% for all of our tests.

For every photon, information about the position, radiant flux, and photon direction is required for collection and shading. The structure of a photon can be seen in Table 2. The radiant flux and direction are both stored individually in a texture. A 16-bit-per-channel texture has enough accuracy for this information. The position is stored in an AABB buffer to build the acceleration structure photon map. This is necessary to use an inverse radius search [Evangelou et al. 2021]. Additionally, the face normal can be stored to handle problems on corners and thin geometry. The face normal can be converted from the Cartesian coordinates system to the spherical coordinate system to be stored in the alpha channel of the flux and direction textures. An atomic counter is used to determine the photon index.

Field	Format	Bytes
Position	AABB	24
Flux	RGB[A]16F	8
Direction	RGB[A]16F	8
Face Normal	RG16F	0 [4]

Table 2. Structure of a photon. Forty bytes are needed in total. Every field has its own buffer. Flux and direction are stored as a texture while the position is stored as a buffer. The face normal can be encoded to two channels and stored in the alpha channels of the direction and flux.

As GPU buffers cannot be dynamically resized, a maximum size of the photon buffers needs to be set beforehand. If the buffer is too small, photons will be lost. A large buffer will cost more memory and will increase acceleration structure build time, because empty AABBs take building time. Using the number of photons from the Photon Generation pass is not a good option due to the cost of GPU-to-CPU synchronization, as the building size for the acceleration structure needs to be set on the CPU. Our results show that the number of stored photons stays around the same. To reduce the impact on acceleration structure time, we take the number of photons from the last iteration, plus a small percentile offset, as the input size for acceleration structure building. The first iteration will use the maximum buffer size as the input size for the acceleration structure, however the same method for approximating the acceleration structure building size can be used to approximate a good maximum buffer size for the current camera perspective in the scene.

The acceleration structure needs to be rebuilt for every iteration with the photon position AABB buffer as input. Updating after every iteration instead of rebuilding leads to worse performance because photons at the same index will move a lot. Each photon map will be built as a bottom-level acceleration structure (BLAS) with a different instance mask for separate collecting. Both BLASes will be used to build the top-level acceleration structure (TLAS).

3.6. Collect Photons

A pseudocode version of the Photon Collection pass can be seen in Listing 2. The collection step is based on an inverse radius search [Evangelou et al. 2021].

For every pixel, an infinitely small ray is shot to collect all the photons. As for photon mapping [Jensen 2001], the radiance estimate (Equation (1)) is used to approximate the radiance for the pixel. To compute the sum, the any-hit shader of the ray tracing API is used. The any-hit shader is called for any successful intersection with a photon (see Figure 2). Additionally, information about the geometry, material, and view vector for the pixel is needed, which are obtained from the Camera Tracing

```
float3 rayGenerationShader(){
    //Initialize the rayPayload.
    rayPayload.vBufferHitInfo = VBuffer[DispatchRayIndex()];
    rayPayload.radiance = float3(0.0);
    //Collect caustic photons with the Photon Acceleration structure.
    TraceRay(photonAS, causticInstanceMask, surfacePosition , viewDirection , rayPayload, tMax=0.001);
    float3 L = rayPayload.radiance / (PI ∗ causticRadius^2);
    //Collect global photons with the Photon Acceleration structure.
    rayPayload.radiance = float3(0.0); //Reset radiance in rayPayload.
    TraceRay(photonAS, globalInstanceMask, surfacePosition , viewDirection , rayPayload, tMax=0.001);
    L += rayPayload.radiance / (PI ∗ globalRadius^2);
    return L;
}

//Is called for any AABB intersection in TraceRay()
void intersectionShader(){
    //Get the AABB from the buffer. InstanceIndex() → photonMap ID. PrimitiveIndex → photon ID.
    AABB aabb = photonAABB[InstanceIndex()][PrimitiveIndex()];
    if(hitSphere(aabb, RayOrigin())) //Check if the origin is inside the sphere.
        ReportHit(RayTCurrent());
}

//Is called for any successful intersection
void anyHitShader(inout RayPayload rayPayload){
    //InstanceIndex() → photonMap ID. PrimitiveIndex → photon ID.
    Photon p = photonBuffer[InstanceIndex()][PrimitiveIndex()];
    BSDF bsdf = getBSDF(rayPayload.vBufferHitInfo);
    //View Direction is stored in WorldRayDirection().
    float3 f_r = bsdf.eval(WorldRayDirection(), p.direction);
    rayPayload.radiance += f_r ∗ p.flux; //Radiance estimate
}
```

Listing 2. Pseudocode for the Photon Collection pass. For both TraceRay() functions, an infinitely small distance is used, e.g., (tMin $\rightarrow$ 0, tMax $\rightarrow$ 0.001). The closestHit and miss shaders can be skipped or left empty. The cyan-colored functions are built-in HLSL functions, and the purple-colored variables are buffers.

pass (see Figure 3). The required shading information, in the form of a V-buffer and view vector, is stored in the ray tracing payload for fast access from the any-hit shader.

To separate the photon map into global and caustic photon maps, the ray tracing step can be repeated. It is faster to separate both photon maps because it is more consistent, as there is no case distinction for the radius and photon map buffer. The instance mask of the API is a fast way to separate both photon maps. The instance index, which is available in the any-hit shader, can be used to differentiate between the caustic and global photon maps.

The intersection shader always needs to report a valid hit to call the any-hit shader, which is a distance between the maximum (TMax) and minimum (TMin) ray distances. The closest-hit shader can be skipped with a flag, and the miss shaders can be left empty, as both are not needed for the collection step.

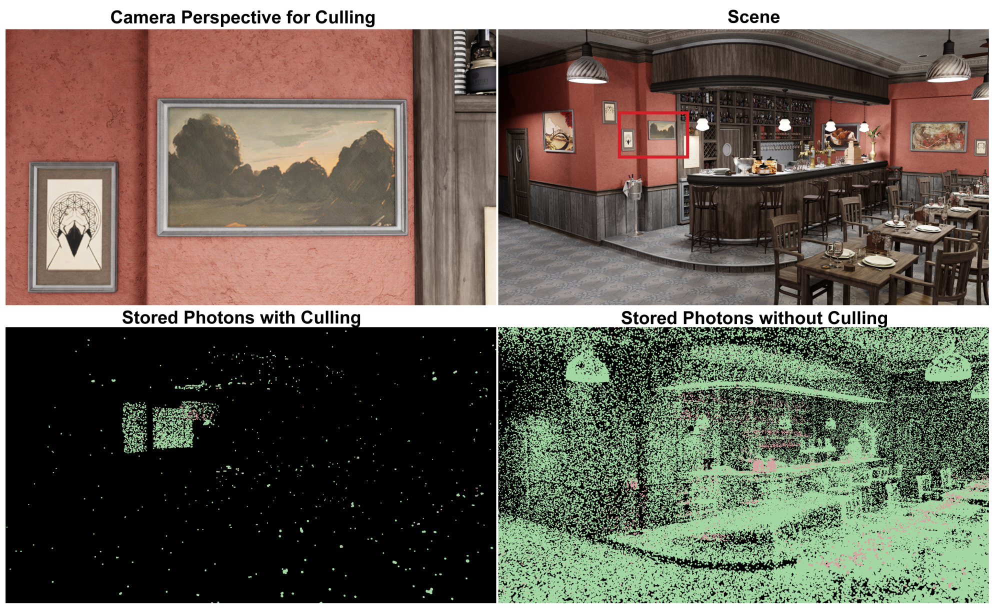
Camera Perspective for Culling
Scene
Stored Photons with Culling
Stored Photons without Culling

Figure 4. Photon culling in the Bistro scene. At the top are the camera perspective used (left) and an overview of the scene (right). On the bottom are visualizations of the stored photons for the camera perspective with culling (left) and without culling (right). Photon sizes were adjusted for the photon visualizations. The global photons are colored green, and the caustic photons are colored pink.

On thin geometry, photons may be collected from the other side of an object (see Figure 9(b)). This can happen on corners because photons that land on the other side of the corner can be valid for both sides. The effect is most noticeable if the invisible side is directly illuminated while the visible side is only indirectly illuminated. With a shrinking radius, this effect will disappear over time as fewer photons from the corner wall are collected, but extreme brightness differences can take a long time. The optionally stored photon face normal can reject photons from other surfaces. The photon face normal is compared against the face normal from the camera hit. If they deviate too much, the photon is rejected. This step is performed in the any-hit shader before the BSDF is evaluated.

3.7. Photon Culling

Photon Culling is our first novel addition, where we want to exclude all non-visible photons using the first diffuse surface hit from the camera. Acceleration structure build times are reduced by Photon Culling, especially in larger and more complex scenes where many photons are not visible to the camera (Figure 4). Naive frustum culling cannot be used because we collect on the first diffuse surface hit and cannot guarantee that all of these hit points are within the camera frustum. Similar to

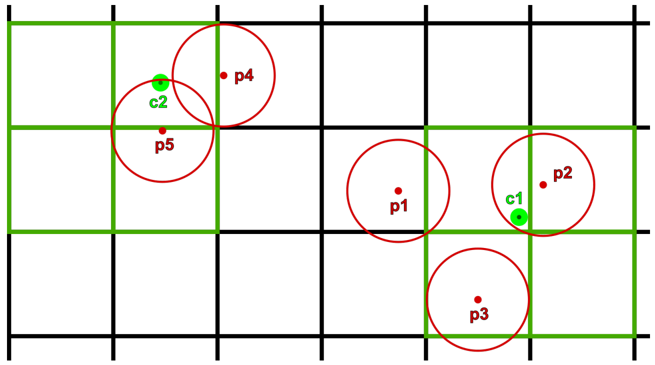

Figure 5. An example of Photon Culling in 2D space. The green-bordered cells are grid cells marked by the positions where the photons should be collected (c1, c2). We need to mark only the nearest four grid cells because a cell has the same size as a photon. A photon is only stored if the corresponding grid cell is marked. Therefore, only p2, p3, and p5 are stored in the photon map. Only p2 and p5 will be collected in the end; p3 is in a marked grid cell but is too far away from c1 to be collected. In the 3D case, we need to mark four additional cells to cover the z-axis.

SPPM [Hachisuka and Jensen 2009], we can use a hash grid to store where the first diffuse surface hit took place. In contrast to SPPM, we do not need the complete hit information because we only want to know if a photon should be stored.

Therefore, we can use a binary mask and mark the hash cells corresponding to the first diffuse surface hits from the camera perspective. In the Photon Generation pass, we can then calculate the hash cell corresponding to the photon's position to determine if the photon should be stored in the photon map.

We implemented the hash map as an 8-bit single-channel unsigned integer texture. The information about the first diffuse hit from the camera's perspective is taken from the Camera Tracing pass. The position, taken from the V-buffer of that pass, is used to mark the corresponding hash cells in a compute shader. The information of one hash grid cell cannot be used alone to determine if a photon should be stored because hash grid cells are fixed spatially. Because we want to keep the number of texture lookups in the Photon Generation pass as small as possible, we need to mark multiple grid cells. We also want to keep the number of marked cells as small as possible because a texture write is used for each marked cell. Therefore, we use the diameter (radius $\times$ 2) from the global photon map as the cell size because it guarantees that we only need to mark the nearest eight cells to guarantee that we collect all photons. A 2D example is shown in Figure 5 where the nearest four grid cells are marked. To determine if a photon should be stored in the Photon Generation pass, we look up the hash cell corresponding to the photon position. If the cell is marked, we store that photon.

Because multiple 3D positions can correspond to the same 1D hash (hash collision), photons may be stored in the photon map, even though they are not visible.

```
//Write into the culling mask. In the compute shader the function is called once per screen pixel.
//The cullingHashMap should be cleared to 0 before calling the compute shader.
void writeCullingMask(float3 position, float photonRadius){
    int3 cells[8] = posToNeighboringCells(position, 2 * photonRadius); //Get all 8 neighboring hash grid cells.
    for(uint i=0; i< 8; i++){ //Loop over all cells.
        uint hashIdx = calcHash(cells[i]) & (GRID_SIZE − 1)}; //Get 1D hash from 3D cell
        cullingHashMap[hashIdx] = 1; //Mark the cell.
    }
}

//Called in the PhotonGeneration pass to check if a photon should be stored
bool checkCullingMask(Photon p, float photonRadius){
    int3 photonCell = posToCell(p.position, photonRadius);
    uint hashIdx = calcHash(cells[i]) & (GRID_SIZE − 1);
    return cullingHashMap[hashIdx] == 1;
}
```

Listing 3. Pseudocode for the Photon Culling technique. The writeCullingMask() function is called once for every screen pixel in a compute shader to mark the cells in the binary hash grid mask. The checkCullingMask() function is called in the Photon Generation pass before storing a photon. We only use the first x-bits from the hash because otherwise the hash grid buffer would be too big (2^{32} elements for a 32-bit hash). GRID_SIZE is always 2^x, so the upper bits can be omitted with an AND operation, and the hash is still valid.

The detection of hash collisions would take more time than it would take to build the acceleration structure with all the false-positive photons. Therefore, this problem is not treated further. The false-positive photons are the source of most of the scattered photons outside of the camera view seen in Figure 4, with some of them being real reflections.

The pseudocode for Photon Culling can be seen in Listing 3. The hash function by Wang [2007] was used in our implementation.

3.8. Stochastic Evaluation

Stochastic Evaluation is our second novel technique to speed up RTPM in areas with high photon density. We want to reduce the impact of the high photon density area by only evaluating the BSDF for a small number of photons stochastically. However, we still need to collect all photons via ray tracing to correctly weigh the exitant radiance in the end.

The pseudocode for Stochastic Evaluation is in Listing 4. We set a fixed number of photons that are evaluated at the end of the collection process. Additionally, we need a counter to track the total number of photons collected for this pixel. The photon list and the counter are stored in the ray tracing payload because we need access in the any-hit shader to store the photon indices in that list. The any-hit shader is only used to store a photon into the list after it passes the sphere intersection test in the

```
 1  struct RayPayload{
 2      uint photonList[LIST_SIZE];
 3      uint n = 0; //Numbers of photons collected
 4      SampleGenerator rng; //Random number generator state
 5  }
 6
 7  float3 rayGenerationShader(){
 8      //Collect photons with ray tracing and fill the photon list in payload. Only one photon map is used here.
 9      TraceRay(photonAS, ... , rayPayload);
10      //Calculate the radiance estimate for the photon list.
11      float3 L = float3(0.0);
12      for(uint i=0; i<LIST_SIZE; i++)
13          L += bsdf * photonBuffer[rayPayload.photonList[i]].flux;
14      return L / (PI * photonRadius^2);
15  }
16
17  //Is called for any successful sphere intersection
18  void anyHitShader(inout RayPayload rayPayload){
19      uint insertIndex = rayTracingPayload.n;
20      rayPayload.n++;
21      //When the list is full, get a random index between 0 and n (reservoir sampling).
22      if(insertIndex >= LIST_SIZE)
23          insertIndex = random(rayPayload.rng) * rayPayload.n //random() returns a float between 0 and 1.
24      if(insertIndex < LIST_SIZE)
25          rayPayload.photonList[insertIndex] = PrimitiveIndex(); //Photon ID
26  }
```

Listing 4. Pseudocode for Stochastic Evaluation with one photon map. To use a caustic and a global photon map, lines 9–13 need to be repeated. Similar to Listing 2, the respective InstanceMask and photon radius must be used for the TraceRay() function and radiance estimate.

intersection shader. We want to keep the list as small as possible because a too-big ray tracing payload can harm performance due to register rearrangement. For all of our tests, we used a list size of 3.

When the photon list is full, multiple strategies could be used. Similar to parallel progressive photon mapping [Hachisuka and Jensen 2010], where only the last photon is stored in the hash map, we could only store the last photons in the list. Depending on how the ray tracing API handles the intersections with the spheres, this could lead to bias. To guarantee that we remain bias-free, we choose reservoir sampling [Vitter 1985] to handle the case where the list is full. On a full list, a photon is inserted stochastically with the probability of $\frac{\text{listSize}}{n}$, with listSize being the fixed size of the photon list in the ray payload and n the current number of photons collected. Because the list is full, $n >$ listSize is guaranteed.

After collecting all the photons, they need to be evaluated with the radiance estimate. For that, we iterate through the photon list. At the evaluation, every photon is additionally weighted with $\frac{n}{\min(n,\text{listSize})}$, which is the inverse insert probability for $n >$ listSize photons collected and else 1.

Stochastic Evaluation replaces the Photon Collection pass, but uses the same inputs and outputs and is therefore easily interchangeable.

Thin geometry that is illuminated by both sides can be a problem for Stochastic Evaluation because photons from both sides of the wall can end up in the photon list as they are only evaluated after. The photons from the invisible side will contribute nothing to the radiance, which leads to visible noise on lower iteration counts because we have less valid samples (see Figure 9(b)). The photon face normal can be used to reject the photons from the other side of the thin geometry before they are stored in the list. For Stochastic Evaluation, we implemented the face normal rejection in the intersection shader. This leads to two additional texture read operations to get the encoded face normal (see the photon structure in Listing 2). The face normal from the camera hit point can be stored in the ray direction, which is not needed in any other way.

The performance of Stochastic Evaluation depends on the scene and the initial radius. When the photon list is not filled, Stochastic Evaluation performs slightly worse than the full Photon Collection from Section 3.6. This is due to the additional overhead of storing the photon hits in the payload before evaluating and thread divergence when the list is full to different degrees. However, the shaders can be swapped after a certain number of iterations to allow optimal performance. The performance of Stochastic Evaluation is further explored in Section 4.5.

4. Performance Analysis

We implemented our RTPM in the Falcor framework [Kallweit et al. 2022] with available source code (see Supplemental Materials). Hash grids or k-d trees are often chosen as acceleration structures for photon mappers. We chose to compare our RTPM against state-of-the-art hash-based photon mappers because in our experience hash-based photon mappers work better on the GPU than k-d tree-based photon mappers. Additionally, a comparison against k-d trees was already done in the fast radius search paper [Evangelou et al. 2021], where the ray tracing accelerated collection performed better than collection with a k-d tree. We chose a hash grid algorithm (Hash-PPM) based on Mara et al. [2013] and parallel progressive photon mapping (Stoch-PPM) [Hachisuka and Jensen 2010] to compete against our RTPM. For a fair comparison, these two algorithms were also implemented in Falcor, with source code available in our GitHub repository. In Section 4.1 we give implementation details for the hash progressive photon mappers. Sections 4.2 to 4.4 cover the quality and performance comparisons between our RTPM and both hash progressive photon mappers. In Section 4.5 we present a performance analysis of Photon Culling and Stochastic Evaluation.

4.1. Hash Photon Mappers Implementation

The implementations of the other techniques are as close as possible to our RTPM. As such they only differentiate in the storing of photons and the collection step, which means that the Camera Tracing pass and most of the Photon Generation pass stay the same. The format of a photon, which can be seen in Table 2, changes in the position field as we now can use an RGBA32F texture instead of an AABB buffer. This lowers the byte count to 32 bytes, as an RGBA32F texture needs 16 bytes compared to the 24 for the AABB.

For Stoch-PPM, we create a hash buffer of fixed size, which stores the number of photons and three textures of the same size, corresponding to the photon format. The size of a hash cell corresponds to the radius of a photon. In parallel progressive photon mapping [Hachisuka and Jensen 2010], the last photon is always inserted. This can lead to bias, as Davidovič et al. [2014] showed. They also proposed a rectified version based on reservoir sampling [Vitter 1985], which we used for our implementation. For photons that should be stored, the counter for the hash cell gets increased by one, and the photon is inserted with the probability of $\frac{1}{n}$, where n is the current number of photons for this hash cell. In the collection step, the hash cell of the hit point for the current pixel is determined. A sphere intersection test is performed on the hash cell and its neighbors. All neighbor cells must be examined to guarantee that all photons are collected. This results in a total of 27 lookups. If a photon passes the sphere intersection test, it is weighted with the number n of photons for the hash cell and added to the radiance for the pixel.

For Hash-PPM [Mara et al. 2013], we use a fixed-size hash grid buffer with a list of photon indices for each cell. Additionally, each cell stores the number of photons per cell. The size of the photon indices list is a user parameter, as the buffer is fixed. On a full list, the photon is inserted with reservoir sampling [Vitter 1985], as we did for Stochastic Evaluation (Section 3.8). As each hash can only contain one cell, quadratic probing is used to resolve conflicts. For the collection, the hash cell corresponding to the hit and all neighbor cells are tested for photons inside the search radius. If the photon passes the sphere intersection test, the BSDF is calculated and the photon is weighted with $\frac{n}{\min(n,\text{maxPerCell})}$, with n being the number of photons per cell and maxPerCell being the fixed maximum number of candidates for a cell. The stochastic collection from Mara et al. [2013], with a Bernoulli random variable as the index offset, is also implemented as an option and was used for all tests.

Both algorithms use the integer hash function by Wang [2007], which was also used by Mara et al. [2013].

We used a buffer with the size of 2^x elements for both hash techniques, with x being the number of hash bits used. Using fewer bits from the hash will result in more collisions but better cache performance. We used the first 20 bits from the hash for Hash-PPM throughout all of our test scenarios. For our system, this was the best trade-off performance-wise between the cache performance and the additional time needed to resolve the hash collisions. For Stoch-PPM, we used 16 to 18 bits of the hash, depending on the number of photons and the scene's size. We reached this number in Stoch-PPM by comparing the structural similarity index measure (SSIM) [Wang et al. 2004] to a reference image after a constant amount of time and choosing the hash size with the best result.

4.2. Test Scenes

Since the efficiency of photon mapping depends on the given scene, multiple scenes were used to evaluate the quality and performance of our photon mapper. The scenes range from smaller scenes with a single light source to huge scenes with hundreds of different light sources. We dispatched a different count of photons for each scene, depending on scene size and number of light sources for the scene. The caustic and global initial radius was chosen to be a good fit for our photon mapper, as large photon radii result in a huge performance drop, which is further explored in Section 4.3. Quality and performance were evaluated by comparing the runtime and resulting image quality. To measure image quality, SSIM [Wang et al. 2004] was used to compare the three photon mappers with a reference image. We used the following test scenes:

- **Bistro (Interior):** The bistro is a complex scene that contains a high number of transmissive objects for caustics as well as a high number of emissive triangles to dispatch photons [Amazon Lumberyard 2017]. Perspective used for performance tests: Figures 1 and 7.
- **Veach-Bidir:** A commonly used scene that shows caustics generated by indirect light [Bitterli 2016]. Perspective used for performance tests: Figure 9.
- **Living Room:** A medium-sized scene with well-visible caustics and indirect light. A single big sphere with many emissive triangles is used to illuminate the scene [Bitterli 2016; Wig42 2014]. Perspective used for performance tests: Figure 7.
- **Caustic Water:** A smaller scene with a very high number of caustic and global photons in a tight space [Bitterli 2016]. Perspective used for performance tests: Figure 7.
- **Caustic Glass:** A small scene that contains a high number of caustic photons that are distributed over a bigger surface. From pbrt-v3 scenes [Pharr et al. 2016]. Perspective used for performance tests: Figure 1.
- **Bistro Full:** The interior and exterior of the bistro were combined to create a big scene with a high number of lights [Amazon Lumberyard 2017]. To get good results, a high number of photons is needed per iteration. Perspective used for performance tests: Figure 9.

Scene	Number of Triangles	Number of Emissive Triangles	Dispatched Photons	Stored Photons*
Bistro	1308k	3576	2 mil	15k / 243k [29k / 654k]
Veach-Bidir	11k	4	1 mil	10k / 328k [10k / 434k]
Living Room	787k	960	2 mil	40k / 610k [66k / 1305k]
Caustic Water	103k	0**	1 mil	[735k / 167k]
Caustic Glass	88k	0**	1 mil	[68k / 110k]
Bistro Full	4140k	24k	4 mil	14k / 139k [40k / 1212k]

* Format is "caustic / global." Numbers in brackets are stored photons without culling.

** Both scenes use one analytic light.

Table 3. Additional information on the complexity and dispatched photons for the scenes used for the performance comparison (Figures 7 and 8). Note that the number of photons stored is significantly lower than the photons dispatched due to the rejection step for global photons (see Section 3.5).

All results were rendered with a resolution of 1920×1080 on an NVIDIA RTX 2080 SUPER. Additional information for each scene is in Table 3. We used different hash grid resolutions for the hash-based photon mappers. For Hash-PPM, we used a hash grid with 2^{20} cells, and for Stoch-PPM, a hash grid with 2^{16} to 2^{18} cells, depending on scene size and the number of photons dispatched. For all of our ray tracing operations, we used a maximum path length of 10. As mentioned, we do not store 70% of the global photons (rejection step) to allow for more caustic photons. The hash grid mask used for Photon Culling used 2^{22} cells.

4.3. Performance

In Figure 6(b) we observed the SSIM [Wang et al. 2004] over time for the three photon mappers in the Living Room scene. The graph shows that RTPM approaches the quality of the reference image the fastest, while Stoch-PPM approaches it the slowest. The same pattern can be observed in the Bistro and Caustic Water scenes in Figure 7 and the rest of the scenes.

RTPM and Hash-PPM have the same quality per iteration. Both have a similar SSIM after the same number of iterations. Figure 8 shows that RTPM has a $2\times$ or more speedup per iteration on average compared to Hash-PPM. This is supported by Hash-PPM having a slower increasing SSIM, as seen in Figure 6(b), and worse image quality after the same time, as seen in Figure 7.

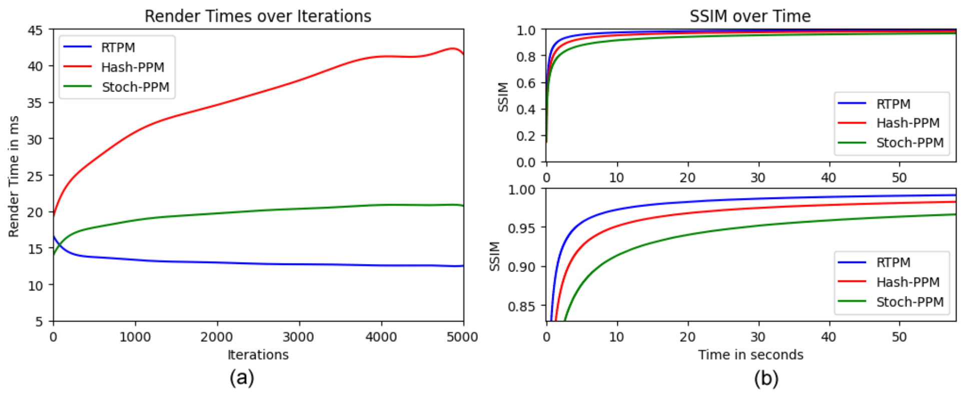

Figure 6. Both graphs are from the Living Room scene. (a) Render times per iteration for 5000 iterations. Render times increase with smaller radii for Hash-PPM and Stoch-PPM while decreasing for RTPM. (b) Changes in SSIM over time. RTPM approaches the reference image the fastest, while Stoch-PPM approaches it the slowest.

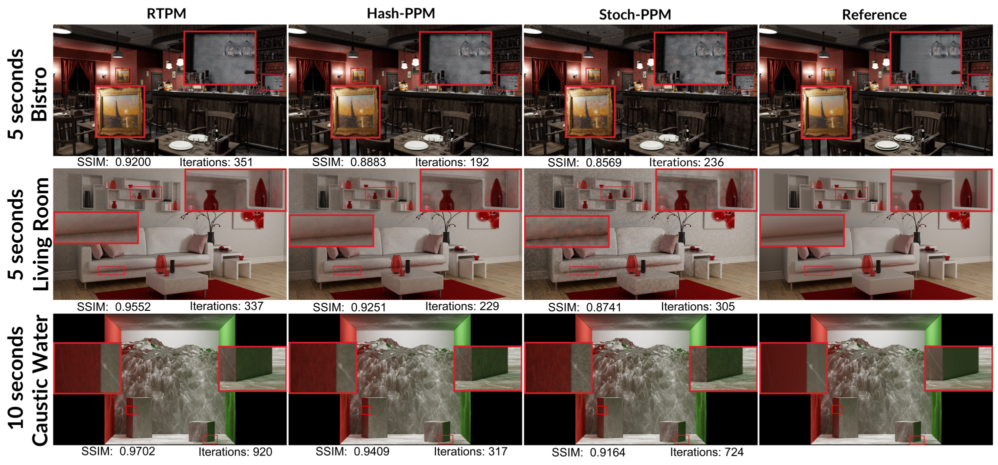
RTPM
Hash-PPM
Stoch-PPM
Reference
5 seconds Bistro
SSIM: 0.9200 Iterations: 351
SSIM: 0.8883 Iterations: 192
SSIM: 0.8569 Iterations: 236
5 seconds Living Room
SSIM: 0.9552 Iterations: 337
SSIM: 0.9251 Iterations: 229
SSIM: 0.8741 Iterations: 305
10 seconds Caustic Water
SSIM: 0.9702 Iterations: 920
SSIM: 0.9409 Iterations: 317
SSIM: 0.9164 Iterations: 724

Figure 7. A quality comparison of the three photon mapping techniques. Images for the same scene were rendered at the same time for all three techniques. The images for the Bistro scene were rendered in 5 s, while Living Room and Caustic Water were rendered in 10 s. The reference image was created with our RTPM using 100,000 iterations.

RTPM is faster than Stoch-PPM on all scenes. The gap between both photon mappers is the smallest on the smaller scenes (Veach-Bidir, Caustic Water, and Caustic Glass), as seen in Figure 8. It is possible to achieve better iteration times for Stoch-PPM by lowering the hash resolution. This, however, worsens the quality over time for all scenes. The quality per iteration is lower than RTPM and Hash-PPM. This is supported by the SSIM in Figure 6(b) and the quality comparison in Figure 7. Stoch-PPM has faster render times than Hash-PPM but produces worse results in the same time frame.

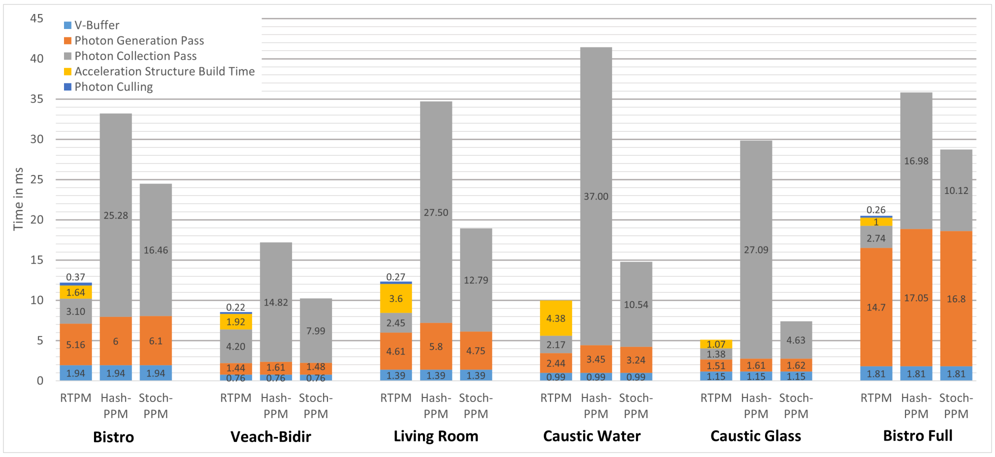

Figure 8. Average iteration times in ms. We used 5000 iterations per scene. Information about dispatched photons, stored photons, and scene size is in Table 3.

When comparing the render times for the individual render passes, it can be observed that RTPM is the fastest for all of the Photon Generation and Photon Collection passes. Hash-PPM is the slowest on the Photon Generation pass because hash collisions need to be handled in addition to inserting photons into the photon buffers. For Stoch-PPM, handling of hash collisions is not needed, which is why it is faster than Hash-PPM for most Photon Generation passes. RTPM is the fastest for most of the Photon Generation passes because the photons only need to be inserted into an array. This has a better cache performance than insertion into a hash grid. For the Photon Collection pass the results show that the RTPM with the inverse radius search [Evangelou et al. 2021] is much faster than Hash-PPM and Stoch-PPM. For both, multiple hash cells need to be searched. Hash-PPM is further slowed down by hash collisions which can occur on each cell. In addition to the Photon Generation and Photon Collection passes, RTPM needs to build the acceleration structure which takes between 1 to 4 ms, depending on the count of photons for the scene. The stored photons for each scene used are in Table 3. The culling pass takes between 0.2 ms and 0.4 ms and mostly affects the acceleration structure build time; the impact will be elaborated in Section 4.5.

The radius reduction impacts the RTPM and both hash techniques differently. The impact that a decreasing radius has on the render time for the three techniques is depicted in Figure 6(a). The increase and decrease in time are mainly from the Photon Collection pass. The times for the other passes remain almost constant with a shrinking radius. For RTPM, the render times get better with a shrinking radius, while the render time increases for both hash techniques. That is because the runtime for the RTPM is dependent on the number of hit photons for one pixel. With a lower radius, the number of hit photons per pixel is reduced, reducing the time for

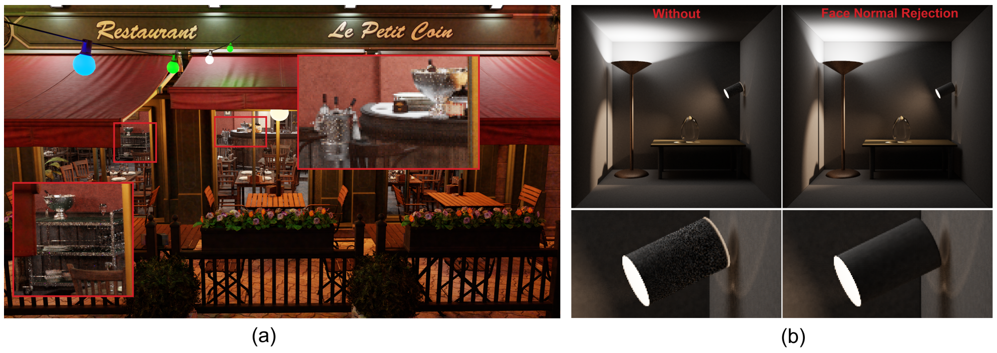

Figure 9. Both images were rendered with 2000 iterations. (a) Bistro Full scene with noticeable noise on metallic objects. (b) Veach-Bidir scene with errors introduced by the thin wall of the lamp. The bright halo on the left side is present in both collection versions. The noise on the lamp is only present for Stochastic Evaluation. Face normal rejection fixes both the halo and the noise.

the Photon Collection because the any-hit shader is called less. The hash grid will increase in resolution for both hash techniques as the hash cells shrink. This leads to more hash collisions and worse cache performance, as neighboring pixels now have different hash cells. In turn, this means that a too-big radius hurts runtime for the RTPM, though it positively impacts runtime for the hash techniques. We have chosen our initial radius so that the RTPM does not need many iterations to reach optimal performance, which corresponds to a good initial radius for image quality.

4.4. Quality

Figures 6(b) and 7 show that the image quality of RTPM is close to the reference after a few seconds. Well-illuminated objects converge in seconds, whereas objects only illuminated by indirect light take longer to converge as only a small number of photons can be collected per iteration. The number of light sources also has a significant impact on image convergence. Scenes with a smaller number of light sources converge faster as most photons are concentrated in one spot.

Objects with rougher metallic materials take longer to converge, as seen in Figure 9(a). This emerges from the modified V-buffer because most rough metallic surfaces scatter the paths over a bigger lobe, which leads to noise. The resulting noise is identical to path tracing noise and could be reduced with the use of a denoiser.

Thin geometry in combination with corners can lead to problems. Photons could be collected from the other side of the corner through an object. This is depicted in Figure 9(b). The bright photons from the inside of the lamp are collected on the outside, resulting in a bright halo. A small halo stays even after over 100k iterations due to the difference in luminosity. The problem can be fixed by using the additional normal rejection. Depending on the scene, this can lead to a small performance gain or loss.

Scene	Without Both	Photon Culling	Speedup	Stoch. Collect.	Speedup	Both	Speedup
Bistro	15.43 ms [30.86 s]	14.21 ms [28.41 s]	1.09	14.14 ms [28.29 s]	1.09	12.86 ms [25.72 s]	1.20
Veach-Bidir	13.75 ms [27.49 s]	13.18 ms [26.36 s]	1.04	10.64 ms [21.27 s]	1.29	10.14 ms [20.28 s]	1.35
Living Room	17.24 ms [34.48 s]	15.01 ms [30.02 s]	1.15	15.76 ms [31.51 s]	1.09	13.51 ms [27.01 s]	1.27
Caustic Water	10.86 ms [21.72 s]	11.14 ms [22.28 s]	0.97	10.90 ms [21.80 s]	1.00	11.16 ms [22.32 s]	0.97
Caustic Glass	5.81 ms [11.61 s]	6.20 ms [12.40 s]	0.94	6.04 ms [12.07 s]	0.96	6.40 ms [12.80 s]	0.90
Bistro Full	26.96 ms [53.91 s]	21.25 ms [42.50 s]	1.27	26.12 ms [52.23 s]	1.03	20.91 ms [41.81 s]	1.29

Table 4. Runtime comparison of both optimizations for RTPM. Both are compared individually and together against the runtime without both optimizations. Time is shown as the average render time over 2000 iterations. The total render time is shown in brackets below.

4.5. Stochastic Evaluation and Photon Culling

Photon Culling is recommended for all scenes where large regions of the scene are not visible from the perspective of the camera. The time savings in acceleration build times make up for the extra cost of generating the culling hash for all bigger scenes. This is also supported by Table 4, where the culling provides a 10% to 20% speedup on bigger scenes. In smaller scenes like Caustic Water or Caustic Glass, Photon Culling is slower than without as there is only a small number of photons that get culled. The acceleration time benefit does in this case not outweigh the around 0.25–0.4 ms culling hash build time (see Figure 8). The Veach-Bidir scene shows that Photon Culling can also be faster for smaller scenes if enough photons are obscured. For this scene, many photons are inside the two lamps and so are not visible to the camera (see Figure 9(b) for the scene).

The performance of Photon Culling is also dependent on the size and resolution of the hash grid. The resolution is dependent on the diameter of the photons. Similar to the hash-based photon mappers, the performance will drop with smaller photon radii. Because we use a low-bandwidth mask, the performance drop is minimal, even for a bigger mask. The size of the hash grid influences the write performance and the number of false positives. If the size is too big, the performance for marking the cells will drop drastically, while acceleration structure build time decreases. If the size is too small, marking the cells will be fast but instead will store many false-positive photons due to the increasing number of collisions. In the end, this is a balancing act. For our hardware, a hash grid size around 2^{22} performed the best for all scenes. However, the performance for hash grid sizes of 2^{21} to 2^{25} was similar and could be better depending on the scene or viewport resolution. A higher resolution of the viewport means that more hash grid cells are marked, and therefore there are more false-positive photons. The size of the hash grid can be increased accordingly to counteract this.

The number of photons stored in the scene can also impact the performance of Photon Culling. Generally, the more photons that can be rejected, the better Photon Culling performs. So, lowering the percentage of rejected photons in the rejection step or increasing the number of photons dispatched positively affects the performance difference for Photon Culling. For example, we can take the Bistro scene with the viewport from Figure 7 and compare the total render time for 3000 iterations with and without the rejection step for Photon Culling enabled and disabled. For 70% rejected global photons, we need 40.9 s without Photon Culling and 36.8 s with it (speedup of 1.11). Without the rejection step, we needed 68.14 s without and 51.0 s with Photon Culling (speedup of 1.34). We observed the same for other scenes. Additionally, it can be worthwhile to increase the size of the hash grid because the time gained for culling more false-positive photons can make up for the additional mask write time.

Stochastic Evaluation works well if there is a high photon density with many photons overlapping. As Table 4 shows, it is faster or as fast as Full Evaluation with one exception. It performs worst on the Caustic Water and Caustic Glass scenes. For both scenes, a single analytic light is used. A smaller initial radius can be chosen for scenes with analytic light, which leads to worse performance for Stochastic Evaluation. Render times with Full and Stochastic Evaluation for different radii can be seen in Figure 10. Full Evaluation is faster at a radius of 0.024 or smaller for the Bistro scene because fewer photons overlap (Figure 10(a)). That means that Stochastic Evaluation performs worse if the photon list (in our test, the list size was 3) is not filled for most pixels. With the radius reduction by Knaus and Zwicker [2011], the radius is not reduced linearly as in Figure 10(a) but exponentially. Depending on the initial radius, the point where Stochastic Evaluation is slower than Full Evaluation is reached either very quickly or after a long time. In Figure 10(b) we compare the performance of both photon evaluation techniques in the Bistro scene. For the Bistro scene, the second case occurred. Full Evaluation is only faster after around 11,000 iterations, with both techniques having the same render time for the range between 8500 to 11,000. At that point, the collection strategy could be changed from Stochastic Evaluation to Full Evaluation. For the scenes where Stochastic Evaluation does not perform well (Caustic Water, Caustic Glass, and Bistro Full), we recommend a swap at 100 to 500 iterations. At early iterations Stochastic Evaluation is very likely better than Full Evaluation, as Figure 10(a) shows. The swap point needs to be approximated for scenes where Stochastic Evaluation is faster after 1000 iterations. A swap point at around 10,000 iterations provided good results in our tests. Overall, Stochastic Evaluation allows for a lower render time penalty when using higher radii.

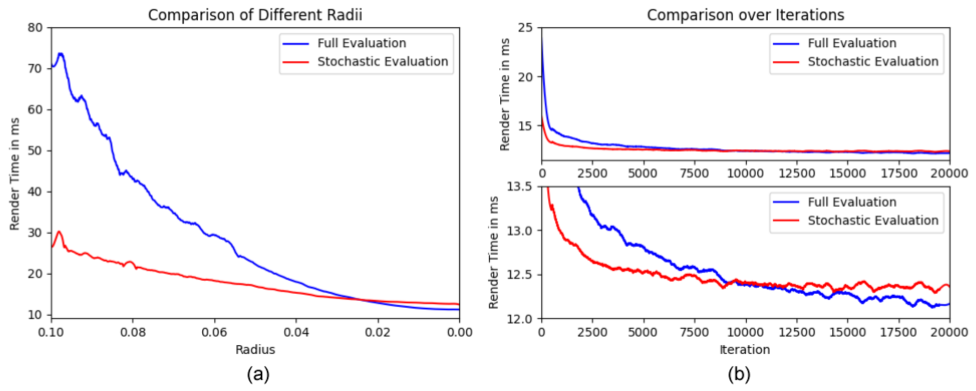

Figure 10. Comparison of Stochastic Evaluation and Full Evaluation in the Bistro scene. (a) Render times for different global radii. Culling and caustic photon collection were disabled. (b) Render times over iterations. The radius for the performance tests (Figures 7 and 8) were used. Global radius starts at 0.07 and caustic radius at 0.012. The radius is reduced with the formula by Knaus and Zwicker [2011] (Equation (2)). This leads to a fast reduction of the radius at the start that quickly decreases.

This can lead to better results in the same render time for most scenes if used at the correct point.

Figure 9(b) shows an error that introduces extra noise that can only happen for the Stochastic Evaluation. This can happen on thin surfaces where photons hit both sides of the surface. During collection, only photons from the other side of the surface may be collected, which in turn are shaded black due to the BSDF. This is only really a problem if there are more photons on the invisible side of the surface, as it is more likely that they are collected instead of the photons from the correct side. Enabling the photon rejection via the photon face normal fixes this problem completely. This introduces a slight additional cost but lowers the noise in affected areas significantly.

5. Conclusions and Future Work

We presented an implementation for a progressive photon mapper that takes advantage of the current ray tracing hardware. In addition, we introduced two smaller novel techniques that improve ray tracing–based progressive photon mappers further. With this, we demonstrated that ray tracing–based collection outperforms state-of-the-art hash-based GPU photon mapping algorithms.

The basis of this ray tracing–based photon mapper could be used for further improvements. The number of photons stored and built per iteration could be lowered further as acceleration structure building time can be very time-consuming for RTPM. With Photon Culling, we presented a fast way to reduce the number of photons stored per iteration, but we think that this can be reduced even further.

We can also imagine that Photon Culling can be combined with other existing techniques to gain performance boosts. Vertex connection and merging [Georgiev et al. 2012] could be one of them because photon mapping is not needed for all pixels, and a sizable number of photons could be culled. It could also be used for caustic-only photon mapping techniques similar to Yang and Ouyang [2021] or other global illumination algorithms similar to photon mapping that use spatial queries.

The generation time could also be lowered significantly as, currently, we distribute photons from all light sources of the scene, which results in many photon paths that do not contribute to the image. Photon guiding techniques similar to Grittmann et al. [2018] could be used to reduce the number of unused photon paths to speed up the photon distribution and raise the quality for each iteration as more photons would land within the camera frustum. Additionally, this could work well in combination with Photon Culling because the culling mask could be used as a feedback buffer.

The global photon map could also be evaluated via final gathering to produce images better suited for denoising because this will produce a noise similar to path tracing. The photon map noise is hard to denoise due to its blurry nature. An additional camera ray would be necessary to evaluate the global photon map at the second diffuse surface hit as well as the evaluation of direct lighting at the first diffuse surface hit.

As the rendering time for each iteration is within the real-time constraint of 16.6 ms, it could be possible to use photon mapping for real-time global illumination with caustics. Direct lighting could be calculated with other methods while the photon mapper is used for caustics and indirect illumination only. Some kind of temporal filtering would have to be used because one iteration is not enough to produce good results.

Index of Supplemental Materials

The full source code can be found at
https://github.com/SirKero/RTProgressivePhotonMapper.
A publication snapshot is available at
https://jcgt.org/published/0012/01/01/RTProgressivePhotonMapper.zip.

References

Amazon Lumberyard, 2017. Amazon Lumberyard bistro. Open Research Content Archive (ORCA), July. URL: `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`. 17

Bitterli, B., 2016. Rendering resources. URL: `https://benedikt-bitterli.me/resources/`. 17

Burns, C. A., and Hunt, W. A. 2013. The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT) 2*, 2, 55–69. URL: `http://jcgt.org/published/0002/02/04/`. 6

Davidovič, T., Křivánek, J., Hašan, M., and Slusallek, P. 2014. Progressive light transport simulation on the GPU: Survey and improvements. *ACM Transactions on Graphics 33*, 3, 29:1–29:19. URL: `https://doi.org/10.1145/2602144`. 16

Evangelou, I., Papaioannou, G., Vardis, K., and Vasilakis, A. A. 2021. Fast radius search exploiting ray tracing frameworks. *Journal of Computer Graphics Techniques (JCGT) 10*, 1, 25–48. URL: `http://jcgt.org/published/0010/01/02/`. 2, 3, 5, 8, 9, 15, 20

Georgiev, I., Křivánek, J., Davidovič, T., and Slusallek, P. 2012. Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics 31*, 6 (November), 192:1–192:10. URL: `http://doi.acm.org/10.1145/2366145.2366211`. 2, 25

Grittmann, P., Pérard-Gayot, A., Slusallek, P., and Křivánek, J. 2018. Efficient caustic rendering with lightweight photon mapping. *Computer Graphics Forum 37*, 4, 133–142. EGSR '18. URL: `https://doi.org/10.1111/cgf.13481`. 25

Hachisuka, T., and Jensen, H. W. 2009. Stochastic progressive photon mapping. In *ACM SIGGRAPH Asia 2009 Papers*, Association for Computing Machinery, New York, SIGGRAPH Asia '09, 141:1–141:8. URL: `https://doi.org/10.1145/1661412.1618487`. 3, 4, 5, 12

Hachisuka, T., and Jensen, H. W. 2010. Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches*, Association for Computing Machinery, New York, SA '10, 54:1. URL: `https://doi.org/10.1145/1899950.1900004`. 2, 3, 14, 15, 16

Hachisuka, T., Ogaki, S., and Jensen, H. W. 2008. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 Papers*, Association for Computing Machinery, New York, SIGGRAPH Asia '08, 130:1–130:8. URL: `https://doi.org/10.1145/1457515.1409083`. 3, 4, 5

Hargreaves, S., 2004. Deferred shading. Presented at Game Developers Conference, March. URL: `https://my.eng.utah.edu/~cs5600/slides/Wk%209%20D3DTutorial_DeferredShading.pdf`. 6

Jensen, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Ltd., Natick, MA. URL: `http://graphics.ucsd.edu/~henrik/papers/book/`. 2, 4, 8, 9

Kajiya, J. T. 1986. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, SIGGRAPH '86, 143–150. URL: `https://doi.org/10.1145/15922.15902`. 2

KALLWEIT, S., CLARBERG, P., KOLB, C., DAVIDOVIČ, T., YAO, K.-H., FOLEY, T., HE, Y., WU, L., CHEN, L., AKENINE-MÖLLER, T., WYMAN, C., CRASSIN, C., AND BENTY, N., 2022. The Falcor rendering framework, 3. URL: `https://github.com/NVIDIAGameWorks/Falcor`. 15

KAPLANYAN, A. S., AND DACHSBACHER, C. 2013. Adaptive progressive photon mapping. *ACM Transactions on Graphics 32*, 2 (April), 16:1–16:13. URL: `https://doi.org/10.1145/2451236.2451242`. 4

KIM, H. 2019. Caustics using screen-space photon mapping. In *Ray Tracing Gems*, E. Haines and T. Akenine-Möller, Eds. Apress, Berkeley, CA, 543–555. URL: `https://doi.org/10.1007/978-1-4842-4427-2_30`. 3

KNAUS, C., AND ZWICKER, M. 2011. Progressive photon mapping: A probabilistic approach. *ACM Transactions on Graphics 30*, 3 (May), 25:1–25:13. URL: `https://doi.org/10.1145/1966394.1966404`. 3, 4, 5, 6, 23, 24

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1993. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, Association for Computing Machinery, Portuguese ACM Chapter, Lisbon, 145–153. URL: `https://graphics.cs.kuleuven.be/publications/BDPT/index.html`. 2

MARA, M., LUEBKE, D., AND MCGUIRE, M. 2013. Toward practical real-time photon mapping: Efficient GPU density estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Association for Computing Machinery, New York, I3D '13, 71–78. URL: `https://doi.org/10.1145/2448196.2448207`. 2, 3, 15, 16

PHARR, M., JAKOB, W., AND HUMPHREYS, G., 2016. Scenes for pbrt-v3. URL: `https://pbrt.org/scenes-v3`. 17

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, SIGGRAPH '97, 65–76. URL: `https://doi.org/10.1145/258734.258775`. 2

VITTER, J. S. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software 11*, 1 (March), 37–57. URL: `https://doi.org/10.1145/3147.3165`. 14, 16

WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4, 600–612. URL: `https://doi.org/10.1109/TIP.2003.819861`. 17, 18

WANG, T., 2007. Integer hash function, March. URL: `http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm`. 13, 16

WIG42, 2014. The modern living room. Blend Swap, September 14. URL: `http://www.blendswap.com/blends/view/75692`. 17

YANG, X., AND OUYANG, Y. 2021. Real-time ray traced caustics. In *Ray Tracing Gems II*, A. Marrs, P. Shirley, and I. Wald, Eds. Apress, Berkeley, CA, 469–497. URL: `https://doi.org/10.1007/978-1-4842-7185-8_30`. 3, 25

Author Contact Information

René Kern
Clausthal University of Technology
Department of Informatics
Julius-Albert-Str. 4
Clausthal-Zellerfeld, 38678 Germany
rene.kern@tu-clausthal.de

Felix Brüll
Clausthal University of Technology
Department of Informatics
Julius-Albert-Str. 4
Clausthal-Zellerfeld, 38678 Germany
felix.bruell@tu-clausthal.de

Thorsten Grosch
Clausthal University of Technology
Department of Informatics
Julius-Albert-Str. 4
Clausthal-Zellerfeld, 38678 Germany
thorsten.grosch@tu-clausthal.de

René Kern, Felix Brüll, and Thorsten Grosch, Accelerating Photon Mapping for Hardware-Based Ray Tracing, *Journal of Computer Graphics Techniques (JCGT)*, vol. 12, no. 1, 1–28, 2023
http://jcgt.org/published/0012/01/01/

Received:	2022-07-29		
Recommended:	2022-11-30	Corresponding Editor:	Eric Haines
Published:	2023-01-26	Editor-in-Chief:	Marc Olano