Extending AAPS for higher quality caustic rendering

Master's thesis

Author:
August P.B. van Casteren (student)

ICA-5905818

University Supervisor / Examiner
Dr. Jacco Bikker, Utrecht University

2nd University examiner:
Dr. Peter Vangorp, Utrecht University

May 15th, 2023

Abstract

Rendering caustics within the real-time realm has recently become possible. While adaptive anisotropic photon scattering is capable of rendering high quality caustics with tight time constraints, the previous understanding of the method was lacking in several aspects. In this article, we report on the performance, features and limitations of AAPS. Two of these problem areas are aliasing and indirect visibility and we propose solutions for them. Aliasing can be significantly reduced at a low cost. Indirect visibility can also be solved in real-time, but the cost of doing so using our method is significantly higher than AAPS on its own.

Utrecht University

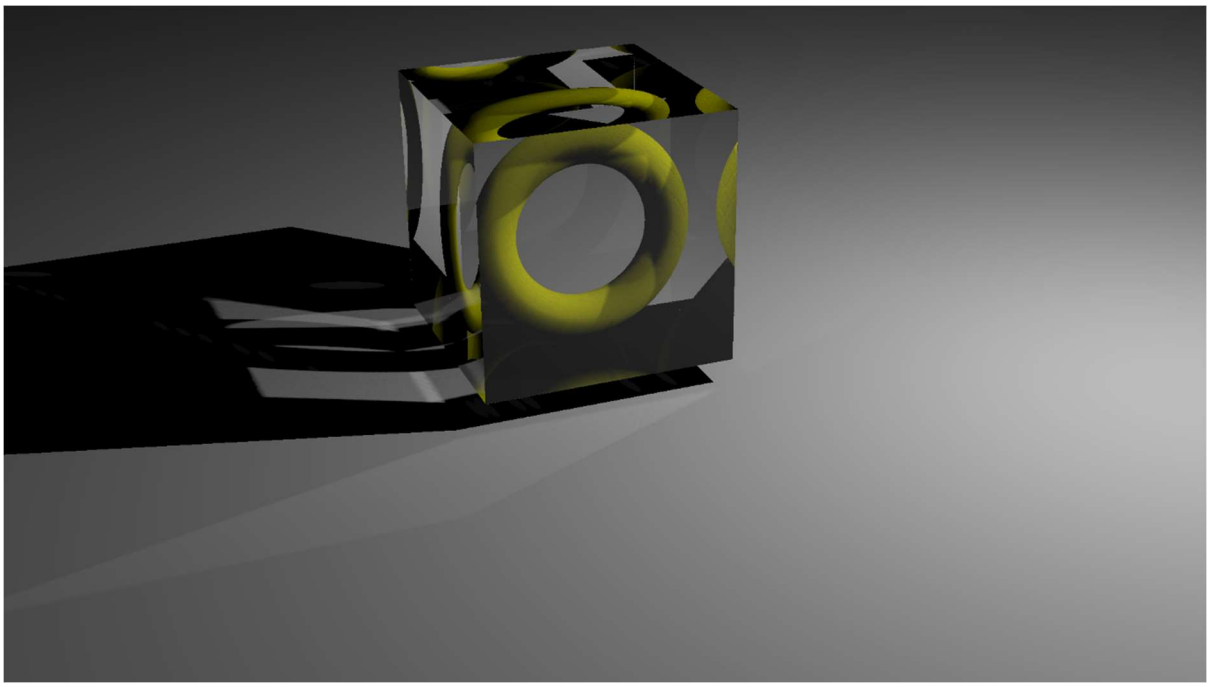


Figure 1 - A yellow torus within a glass cube. This scene exposes difficult to render refraction -> diffusion -> refraction light paths. Using our extension of AAPS, we were able to render it in real-time (this image took <10ms). Lighting is fully dynamic, and guided by AAPS. 180,000 photons were recorded and 1920x1080 primary rays were used.

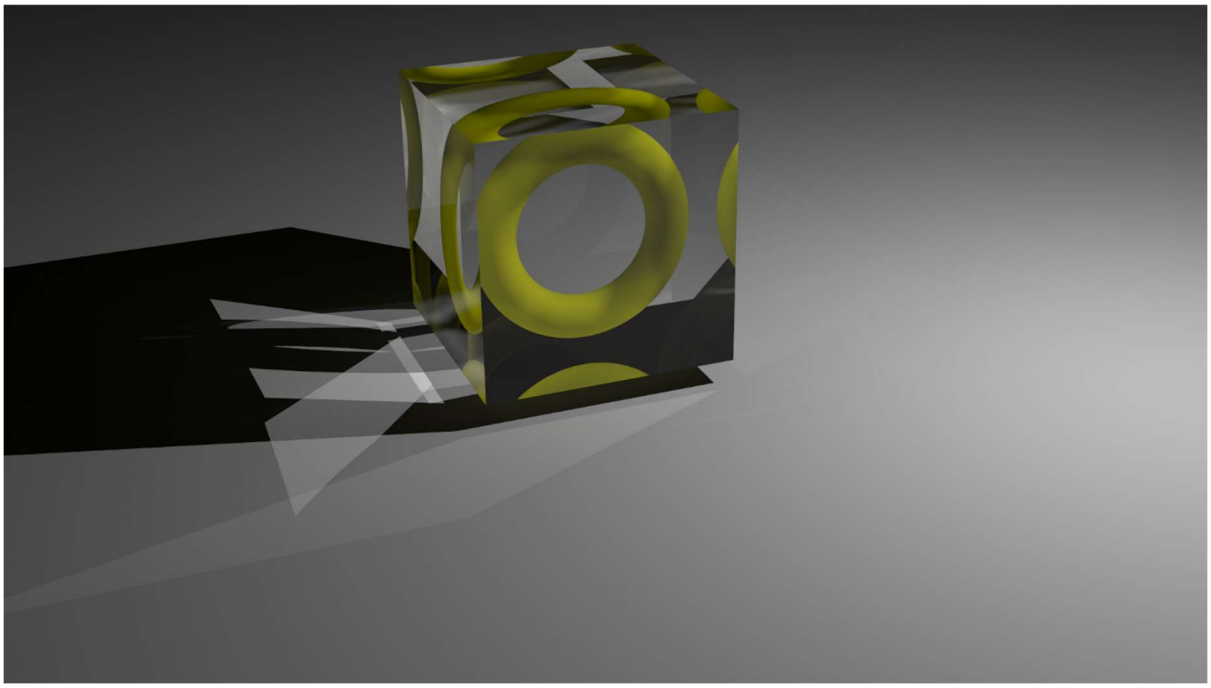


Figure 2 - A yellow torus within a glass cube. Rendered using a reference renderer, at 2500 samples per pixel. Some caustic patterns visible in this image are not visible in Fig. 1. This is due to a recursion cap that was introduced to help maintain high framerates.

Table of Contents

1. Introduction

1.1 What are caustics?

Caustics are an optical effect created when light is concentrated in an area on a surface after interacting with a specular or dielectric object. The exact guideline for when that light on a surface is concentrated enough is ambiguous. For this reason, in this document, we will refer to all light that follows a path containing a reflection or refraction before making at least one bounce on a non-specular, non-dielectric surface, as caustics. An example of a caustic caused by a ring can be seen in Fig. 3.

Rendering caustics has been a goal in real-time settings such as games for a long time. Research publishing real-time caustics dates back as early as the year 1996 [Stam 1996]. Because underwater caustics are among the most prominent cases of caustics in real life, rendering these specifically has received special attention [Shah et al. 2007]. Though early implementations of caustics in games mostly used physically inaccurate tricks designed to fit within existing performance constraints, they clearly indicate an interest in rendering caustics in games.

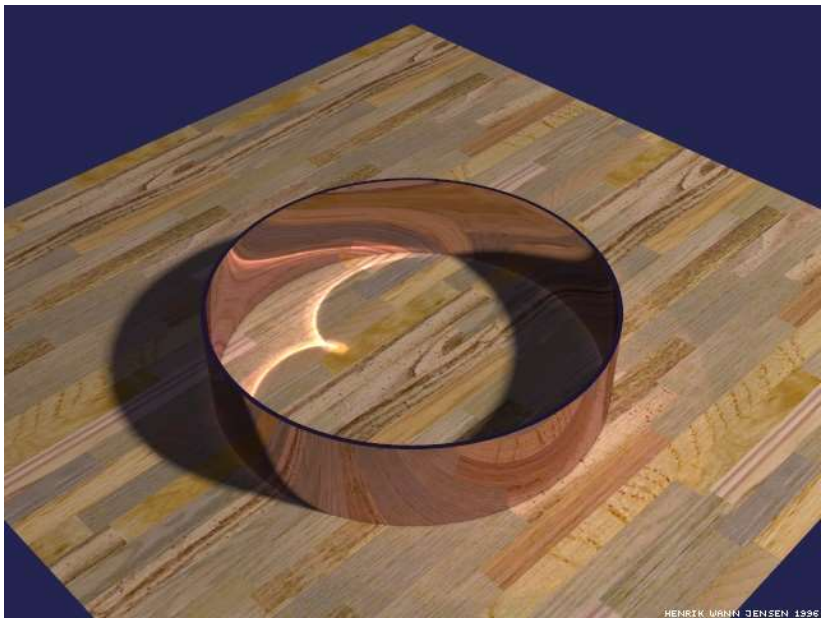


Figure 3 – A caustic caused by a ring, rendered using photon mapping (Image by Henrik Wann Jensen)

1.2 Overview

In this thesis, we thoroughly investigate caustics rendering in real-time settings. We start by explaining what caustics are, and what techniques exist to render them. After that, we outline multiple categories of physically based rendering techniques that use ray tracing and are able to render caustics.

One of these techniques is photon mapping. Photon mapping is the most successful method in terms of rendering caustics. For this reason, we go through all important extensions that have been made to photon mapping over the years. After a detailed explanation of photon mapping itself, and covering the relevant extensions, we explain Adaptive Anisotropic Photon Scattering (AAPS). AAPS is the most effective method of rendering caustics, and for that reason, the main focus of this work.

After introducing the existing techniques, we explain how we implemented AAPS, and what decisions we had to make. We try to be as reproducible as possible, as recreating AAPS from its

original article was an ambiguous task. After this, we explain the scene setup of our experiments, and the measurement criteria we use.

Next, we discuss the results of the experiments, measuring performance and quality, and showing the effect certain parameters have on them. We then dissect the time spent on executing the AAPS algorithm into different categories, on different hardware, and discuss the findings.

After reflecting on the performance of AAPS, we list and explain the shortcomings that we found for the images it produces. One category of shortcomings covers problems resulting from rasterizing small primitives. We also display experiment results that reinforce our explanation of these shortcomings.

In the chapter after that, we discuss several approaches to alleviate the shortcomings. We start with convergence related problems. We first list three approaches that we implemented and experimented with, and then two more speculative approaches, all aimed at improving convergence. We also show that two of the three tested approaches are difficult to argue in favor of, but the third method, which applies MSAA (with invocations) exclusively to small photons, has a more realistic range of use cases.

After discussing solutions for convergence related problems, we introduce a technique we used to render caustics that fall outside of the frustum or are obstructed, but are still indirectly visible, i.e. through a glass object. This is done in real-time, something that was not possible so far. Finally, we explain two more speculative approaches have potential to improve ray tracing performance, that we have not experimented with, but are worth looking into.

Finally, we conclude by looking back at the results and summarizing important findings and takeaways about AAPS. Additions to AAPS are also evaluated on their usefulness here. We also point out three important areas of the algorithm that we did not fully cover and may deserve attention in the future.

1.3 Non-physically based techniques to render caustics

Before graphics cards were fast enough to perform physically based lighting calculations in real-time applications, whenever caustics were rendered, it was through tricks. This is still the case except for cutting edge software running on cutting edge hardware – a rare combination. The tricks used to render caustics were designed specifically to be cheap to compute, and yield aesthetically pleasing results, as these requirements go hand in hand with video games.

Non-physically based approaches have proven useful over the years. One such example implements caustics in Unreal Engine 4 as seen in Fig. 4 [Enchev 2020]. While far from physically correct, these techniques are able to render convincing looking underwater caustics. Another example of such a method is Random Caustics [Stam 1996]. A more common way with a wide range of implementation variants is using decal textures. Unfortunately, caustic hacks are mostly viable for one specific use case: underwater caustics caused by a large body of water. Faking other types of caustics is not as easy to do in a convincing manner. The method of doing so that has seen most use fakes caustics by letting some amount of light through glass, depending on factors such as thickness and angle [Nichols 2019].

A **decal texture** is a texture that is layered on top of another texture. It supports rendering of various surface features such as bullet holes, stains, wall cracks, but also caustics. One example of such an implementation is stacking four caustic decal textures on top of each other that all gradually pan in different directions. When this is combined with a moving water surface that includes refraction or

an approximation thereof, the resulting caustics, while fake, look convincing. Different rendering engines can have their own tweaks to this process, changing parameters or adding steps to the approach, to better fit to their game. Doom Eternal uses decal textures that are generated per frame based on the water surface [Geffroy et al. 2020]. They simulate refraction based on the water's displacement map, projecting it onto scene geometry afterwards.

Figure 4 - Fake water caustics that look very convincing, while being cheap to compute. (Image by Enchev)

1.4 Physically based techniques that support caustics

1.4.1 Path tracing

While path tracing [Kajiya 1986] is technically able to render caustics, caustics rendered with path tracing typically end up being very noisy. An example of this is visible in Fig. 6, which also compares it to other techniques. This is unsurprising: A major reason path tracing is able to render relatively noise-free images without exceedingly high execution time is importance sampling. Next event estimation is a useful technique which significantly lowers variance, potentially by orders of magnitude. Unfortunately, sampling caustics does not benefit from this improvement. This is in contrast with sampling direct light and indirect light that comes from diffuse surfaces, as illustrated in Fig. 5.

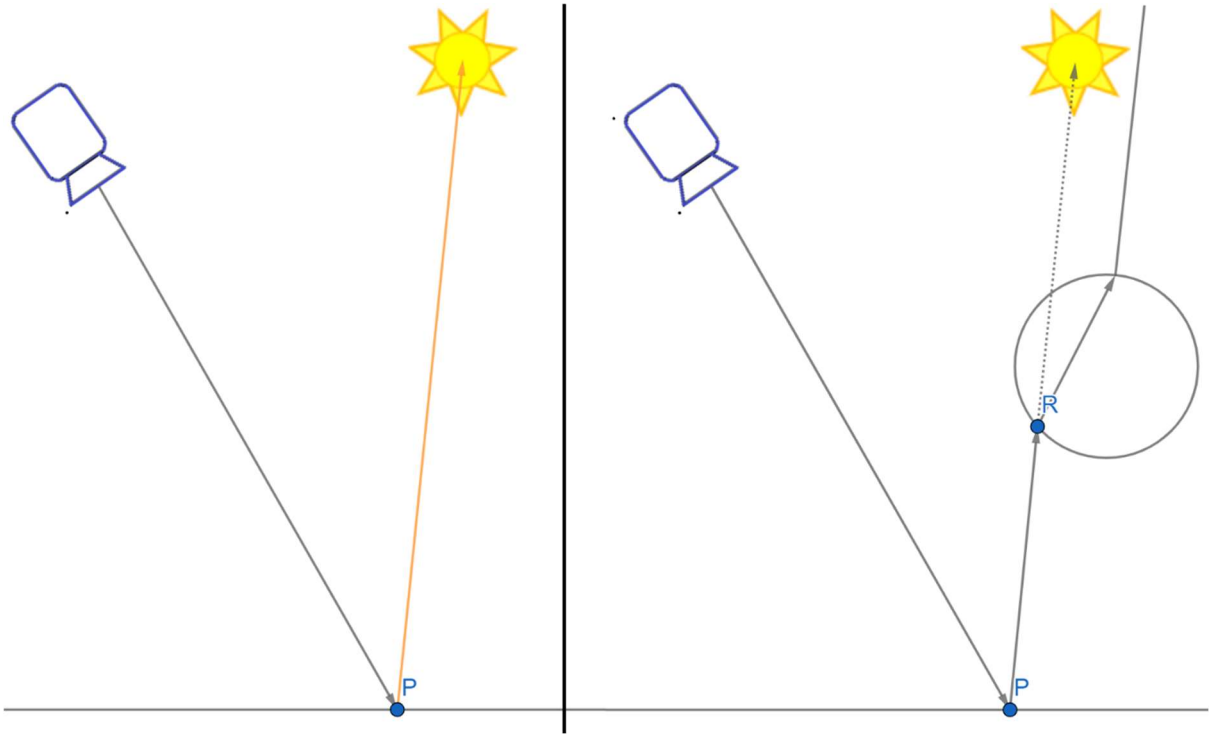


Figure 5 - Where direct light at any point can be sampled using next event estimation, this is not possible for caustic light when doing backward ray tracing (i.e. from the camera).

Instead, in order to render caustics with path tracing, the hemisphere is sampled in a brute force manner. Randomly sampling the hemisphere typically comes with a low probability of hitting a light source, especially if the light source is small or far away. That is why next event estimation is so effective: it finds a path by sampling the light source directly instead of sampling the hemisphere. Unfortunately, next event estimation does not hold up when light arrives at the sampled surface indirectly and the path contains an intermediate vertex, which is the case for caustic light. That is why the number of samples necessary to render caustics using path tracing is prohibitively large. Simultaneously, real-time path tracers only have time to trace a few samples per pixel. As a result, when path tracing is applied to a scene that contains caustics, in practice it leads to completely unrecognizable caustics made up of none other than a few noisy white pixels.

1.4.2 Bidirectional path tracing

Whereas ray tracing [Whitted 1980] takes into account the viewing point, and other algorithms such as the progressive radiosity method [Smits et al. 1992] put emphasis on the light source, bidirectional path tracing [Lafortune and Willems 1993; Veach and Guibas 1995] is described to take into account both the importance of the light source and the viewing point. Considering rendering caustics, bidirectional path tracing is much more effective than regular path tracing as it does part of its random walk starting from the light source. In Fig. 6, the improvement bidirectional path tracing provides over regular path tracing is clear.

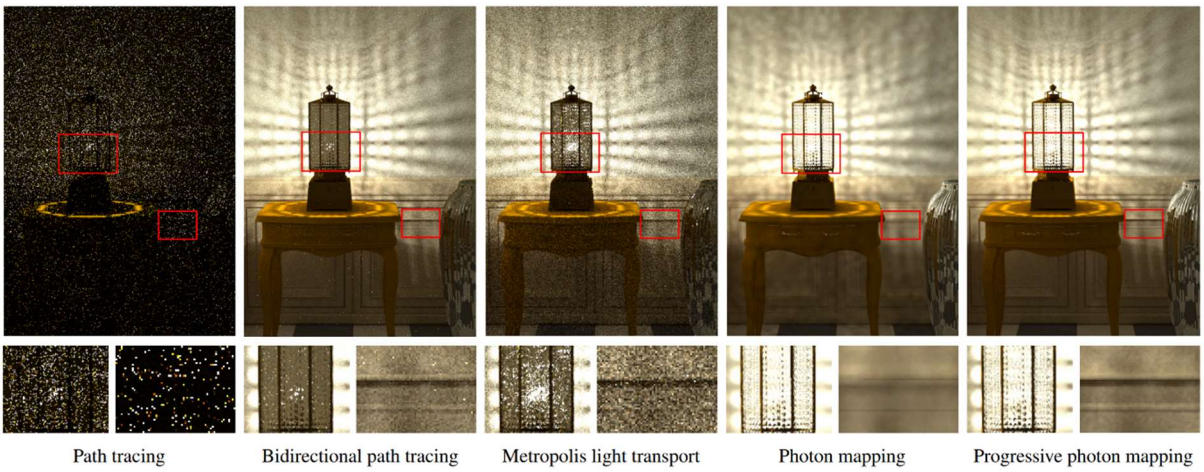


Figure 6 - A comparison of techniques in a scene where nearly all the light is caustic light. The image clearly shows the difficulty in rendering caustic light paths, and how various techniques improve this. Note in particular how noisy images rendered with regular path tracing are. (Image by Veach and Guibas)

Where a regular path tracing algorithm would sample light by strictly going from the eye towards the light source, a bidirectional path tracer samples connections between two paths. One of these paths is a random walk starting at the light source, the other path is a random walk starting at the eye. Shadow rays are traced between combinations of vertices taken from the two different paths. Because one of the random walks starts at the light source, caustic light paths can be rendered without depending on finding the light source by pure chance.

1.4.3 Metropolis light transport

Metropolis light transport [Veach and Guibas 1997] is another contender of the main technique we will investigate (photon mapping) when it comes to rendering caustics. It is a rendering technique that requires more engineering than most because it works off mutations that are implementation dependent. However, it is also distinguished by its ability to render difficult scenes where most of the light transport happens in a small fraction of all possible paths [Pharr et al. 2017].

Metropolis light transport functions by continuously mutating existing paths which have been found to transport some energy, favoring small changes in paths. This reduces the cost per sample to one or two rays in most cases and allows to build mainly on important paths. The mutations that a metropolis light transport renderer features can be hand crafted to target specific difficult to render cases, and more mutations can be invented and added as they are needed. However, while hand-crafted mutations are great at serving specific purposes, the versatility of these mutations and as a result the entire implementation is lacking. This can be problematic when applied in the dynamic scenes real-time applications need to support.

1.4.4 Photon mapping

Photon mapping, which is the main method that we go into, works by tracing photons directly into the scene from the light source(s), and storing them in a photon map. Photon maps containing more photons produce higher quality images. After this map has been created, the shading phase begins. At the start of the shading phase, rays are traced from the camera until they reach diffuse points. These diffuse points can then be shaded using the previously created photon map. Photon mapping is the technique most often used for rendering caustics and its approach lends itself well for that purpose.

1.4.5 Vertex Connection and Merging (VCM)

Vertex Connection and Merging [Georgiev et al. 2012] is a technique that combines bidirectional path tracing and photon mapping together into one method. Its creators reasoned that bidirectional path tracing and photon mapping both struggle to render different, specific, types of lighting. A combination of the two would then cover weaknesses of one technique with the other, for both techniques. Resulting from this is one of the most complete GI solutions out there, covering a broad range of light transport paths with acceptable convergence. It is unknown what the performance of VCM is like on modern hardware, though multiple importance sampling calculations were stated to be taxing [Georgiev 2012]. Georgiev's findings reinforce the idea of applying photon mapping for suitable light transport paths and seeking other methods (in his case bidirectional path tracing) to estimate incoming light from other types of paths. This idea of using a method only where it excels works well for real-time rendering of caustics, as we will discover throughout this article.

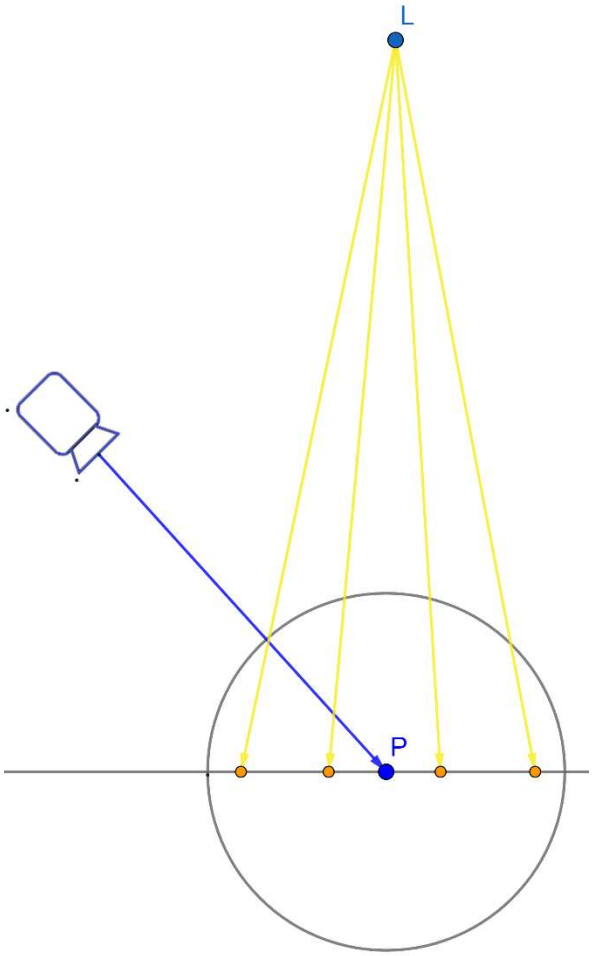


Figure 7 - Photons are gathered around point P within a radius.

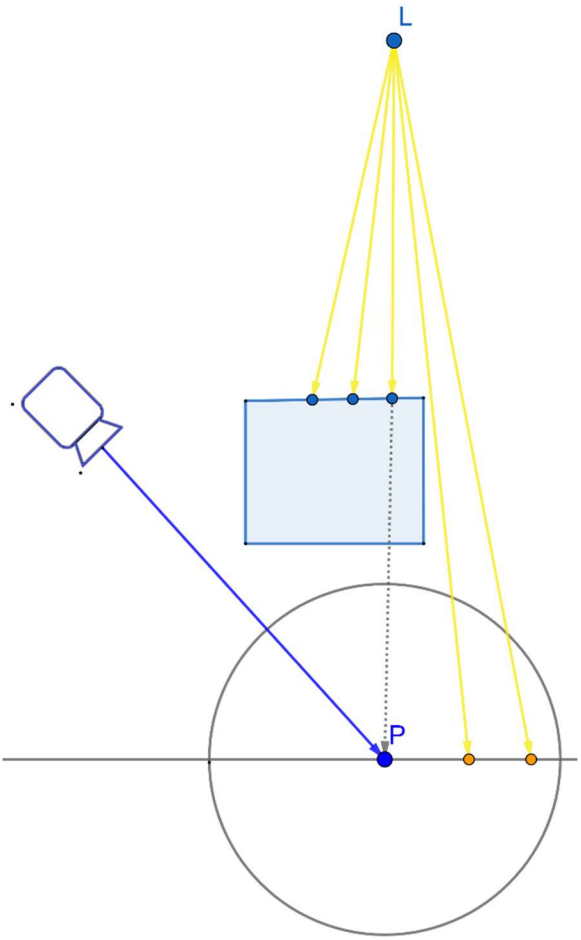


Figure 8 - Photons are gathered around point P. The light source is obstructed, and the point should receive no direct light, but it receives it anyway.

2. Existing photon mapping work

2.1 Photon mapping by Jensen

Photon mapping as originally described [Jensen 1996] is a technique to render images with realistic lighting. Fig. 11 displays an example of an image rendered by photon mapping [Zhou et al. 2008]. Photon mapping is physically based and converges to the ground truth while including a wide range of lighting effects. These lighting effects include for example direct lighting, indirect lighting, any type of materials or BSDF, caustics, color bleeding.

Like most ray tracing techniques, photon mapping has been too slow to be applied in real-time contexts for most of graphics history. On the other hand, its support for a wide range of lighting effects made it [Jensen 2001], and still makes it [Estevez and Kulla 2020], suitable for cinematic rendering. Due to continuous performance improvements of graphics hardware and rendering algorithms, some forms of photon mapping are now also seeing real-time applicability. This includes implementations that do fully dynamic photon tracing.

2.1.1 What photon mapping solves

The rendering equation, first described in 1986 [Kajiya 1986], was reformulated to create photon mapping [Jensen 1996]. Both equations seek to answer the question of how bright a point on a surface is, as seen from a certain direction. Before that question can be asked though, a problem every single rendering algorithm needs to solve is the visibility problem. Before we can render a scene with a camera, we need to know which pixels contain which geometry. This is also known as the visibility problem. Ray tracing and rasterization can both be used to solve the visibility problem. This leads to important data for every single pixel that is required to be able to do shading operations. This data usually includes which primitive occupies the pixel and the location and normal on that primitive that correspond to that pixel.

Solving shading can be done using ray tracing. Real-time software comes with a fixed budget for ray tracing each frame. Resulting from this, a developer can choose between noisy path-traced images that require denoising, or a biased photon mapping image. With limited ray rates, this makes photon mapping attractive.

When the visibility problem has been solved, it also leaves us with the same question that we can ask for every single pixel on this screen: what color does it have? In other words: how much light is going from this point on this piece of geometry towards the direction it is observed from? This is what the rendering equation solves:

$$L_S(x, \Psi_r) = L_e(x, \Psi_r) + \int_{\Omega} f_r(x, \Psi_i; \Psi_r) L_i(x, \Psi_i) cos\Theta_i d\omega_i$$

Where L_s is surface radiance, L_e is surface emission, and the value of the integral, which we name L_r, is reflected light. Ψ_r is the reflected direction, Ψ_i is the incoming direction, and x is the point that is being shaded. Path tracing only counts light incoming at the exact point that is being shaded. Photon mapping, on the other hand, looks at an area around a point and all photons within that area. This is represented by the following equation for reflected light, and the approximation of it.

$$L_r(x, \Psi_r) = \int_{\Omega} f_r(x, \Psi_r, \Psi_i) \frac{d^2\phi_i(x, \Psi_i)}{dA\, d\omega_i} d\omega_i \approx \sum_{p=1}^{N} f_r(x, \Psi_r, \Psi_{i,p}) \frac{\Delta\phi_p(x, \psi_{i,p})}{\pi r^2}$$

Where ϕ_i represents incoming flux, $\Delta\phi_p$ represents flux arriving from photon *p*, and in the approximation ΔA is approximated with πr^2. The Helmholtz reciprocity principle states that incoming and outgoing light can be considered as reversals of each other, so having Ψ_r and Ψ_i switched equates to the same.

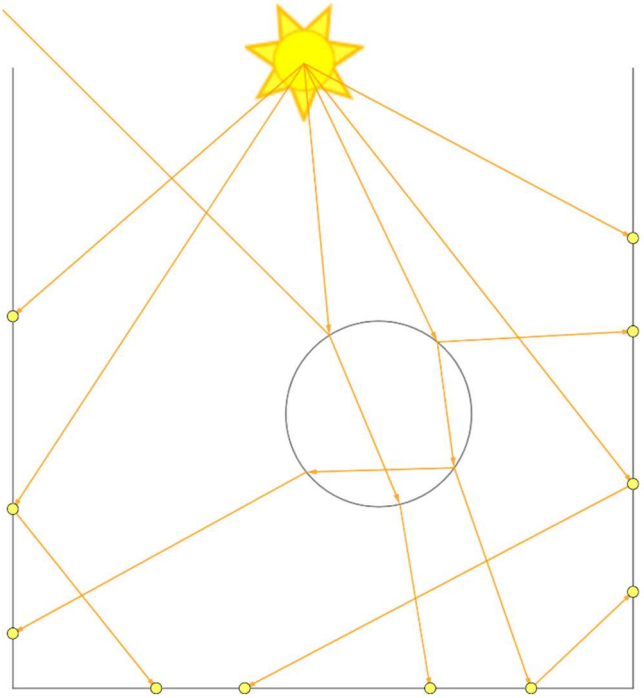


Figure 9 - Photon generation. A light source emits photons in a scene with a glass ball. Only events at diffuse objects are recorded, marked with yellow points.

2.1.2 How photon mapping works

Photon mapping starts by simulating light transport into the scene. It does so by simulating random individual photons being emitted uniformly from a light source in Monte-Carlo fashion. This is done in a physically correct way, and photons are all stored for use in the second part whenever they encounter a suitable surface, such as a diffuse surface. Photons can refract, reflect, or make diffuse bounces, just like real photons.

These photons allow us to then estimate lighting at any point in the scene. To do so, a radius and a number of photons used for shading each point, *N*, are first picked. Fig. 7 shows how this radius works in practice, gathering nearby photons to collect information about incoming lighting at a point. After collecting the photons, the density of photons within the specified radius at that point can be calculated. If more than *N* photons were found within the radius, only the nearest *N* photons are retained for the radiance estimate and as a result the radius for that estimate is reduced.

In photon mapping, photons that do not fall exactly on the shaded point still contribute to the lighting estimate of that point, which is biased, as illustrated in Fig. 8. However, as the radius approaches zero, and photon counts approach infinity, ground truth is approached. The fact that photons have a virtual size in world space is opposite from path tracing, where photons are infinitely small. This size means that photons project their energy onto nearby surface points, which is what makes the technique biased. Photons are usually traced using normal ray tracing techniques, but photon generation, storage, and lookup techniques vary largely per implementation.

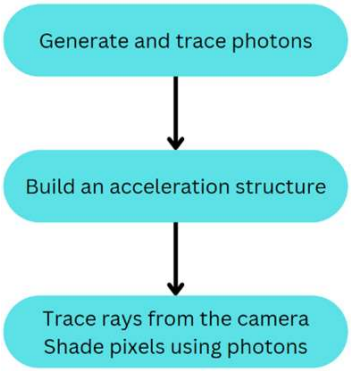


Figure 10 - The Photon Mapping Pipeline

Figure 11 - An example of a caustic pattern rendered using photon mapping using real-time construction of kd-trees. (Image by Zhou et. al)

2.1.3 Jensen's algorithm in practice

Jensen's algorithm for photon mapping is made up of three major steps:

- Generate, trace, and record photons
- Build an acceleration structure containing all the photons
- Trace rays from the camera, and shade them using the photons in the acceleration structure.

Generate and trace photons

Every light source emits a myriad of photons, the exact number of which depends on the desired quality of the image. Any type of light source is supported by photon mapping, as long as the photons are distributed uniformly and randomly over the possible light source area and emission directions. For a square area light for example, a random point on the square combined with a random direction in the hemisphere suffice, as long as both are uniformly generated.

The photon array keeps track of all photons. Any time a photon encounters a material that is not fully reflective or dielectric, its properties are stored into the photon array. Aside from storing its current state, its simulation is continued by reflecting it with a probability distribution defined by the BSDF of the primitive that was hit. Sampling according to the BSDF's distribution can be seen as a form of importance sampling as it prioritizes light paths that are more likely to occur and thus more important. However, in reality it is doing none other than following the principles of physics by doing correctly distributed reflections that follow real life distributions.

Before diffuse reflections are performed with a photon, Russian roulette is performed. Without Russian roulette, photons would risk bouncing indefinitely, depending on the scene. Russian roulette gives the photon a non-zero chance to survive at every random bounce, dividing its contribution by its chance to survive if it does survive so that the expected value remains unchanged. Unimportant photons are usually given low survival odds, to avoid wasting time computing something that will have relatively low effect on the end image. The same thing is usually done in backward, Kajiya style, path tracers, by incorporating it into the implementation of Russian roulette.

Shading

To estimate the irradiance from caustic light at a pixel, Jensen performs a *k* Nearest Neighbor, in short *k*-NN, query. While *k*-NN shading can be used with other photon mapping implementations, different methods to do shading also exist. The radius within which photons are collected is entirely dependent on the scale of the scene. Larger scenes with larger scales in terms of vertex coordinates would also require larger radii. For the example renders shown when the technique was introduced, radiance estimates used photon counts around the range of 50-500 in a *k*-NN query.

For *k*-NN queries, a *k*d-tree was used. The *k*d-tree is traversed, and any relevant photons are added to the max heap, maintaining only the closest *N* photons, if the max heap is full. After the traversal is complete, the photons in the max heap are used to estimate irradiance. *k*-NN queries were stated to produce better images than their more basic counterpart of collecting all photons within the specified radius.

It was later shown that the order of steps in photon mapping can also be reversed with a technique called Progressive Photon Mapping [Hachisuka et al. 2008]. Instead of tracing photons from the light source first, this technique starts its photon mapping pipeline by tracing rays from the camera, and storing the world positions found for the pixels. This is then followed up by tracing photons and shading.

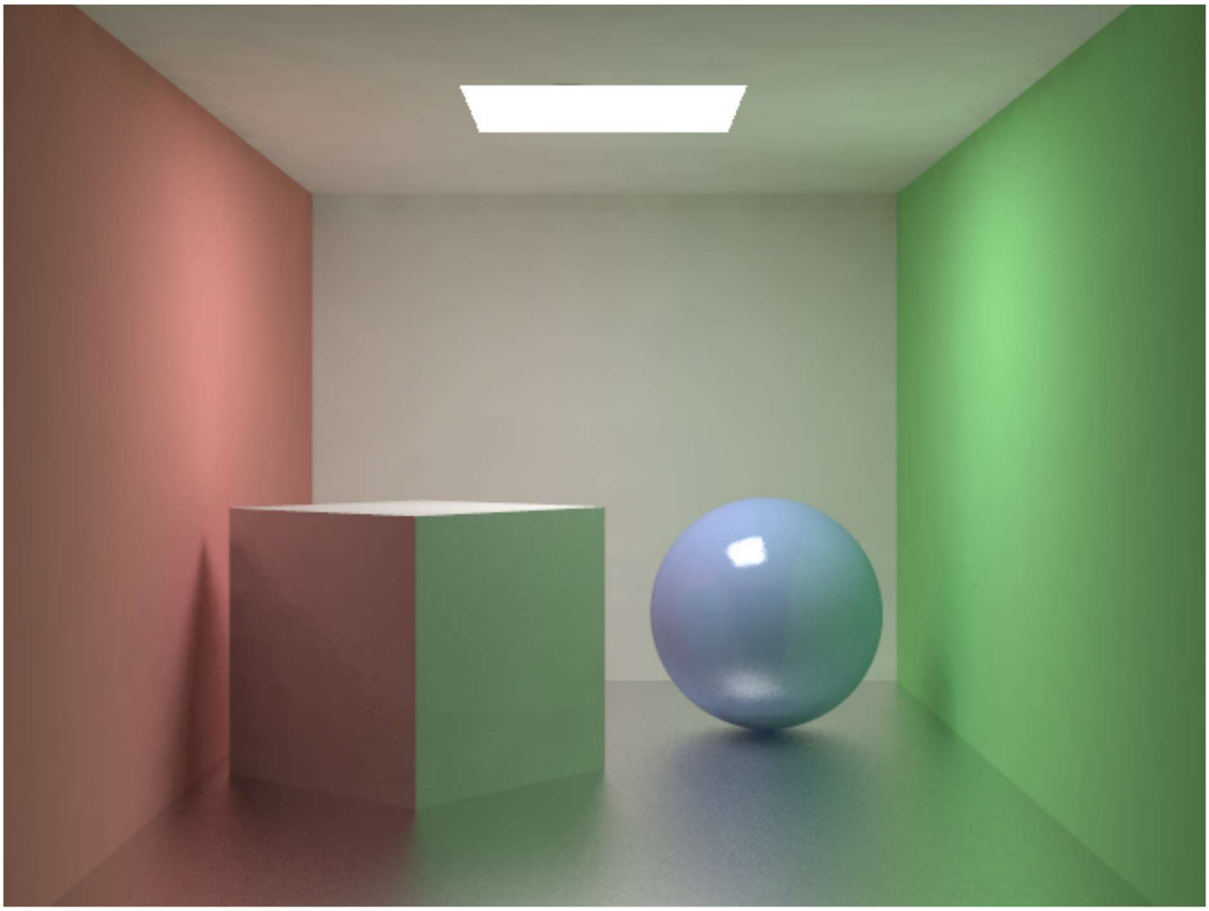


Figure 12 - Cornell box with glossy objects, rendered using photon mapping (Image by Henrik Wann Jensen)

2.2 Photon mapping extensions

The original work on photon mapping led to many new adaptations, which benefit from the extensive range of lighting effects that photon mapping provides. Within photon mapping research, most contributions fall into one of three categories, narrowly representing the steps described in the previous section:

- Photon generation
- Photon storage
- Shading

In order to provide a structured overview, contributions are laid out on by these three categories. Because acceleration structures can vary greatly, and even be omitted altogether, photon storage is the most flexible step, meaning there is likely the most room for improvement. While improvements in acceleration structure do not directly affect image quality, they allow for faster computation times. This in turn allows more photons to be used in the same amount of time, improving the overall image.

2.2.1 Photon generation

When photon mapping was first created, photon mapping and ray tracing in general were too slow to use for real-time applications. Logically, realism was given more weight with photon mapping implementations and performance took the back seat. As GPUs got faster over the years, new improvements were introduced, including techniques that do only part of the photon mapping pipeline in real-time. In recent years, with hardware ray tracing pushing forward the possibilities

within physically based real-time rendering, the real-time generation of photons is also easily within reach. As a consequence, we have even seen variants of photon mapping executing the entire pipeline including the generation of photons in real-time. Improving the relevance of traced photons is still vital, as it yields multi-faceted improvements. Having fewer but more relevant photons means less time needs to be spent throughout all steps: tracing, (optionally) building an acceleration structure, and shading.

Uniformly generating photons

In Jensen's original book on photon mapping, a uniform distribution of emitted photons was used for light sources. It was also found that stratification of samples, limiting randomness to a smaller range of values and equally distributing the photons over these ranges, significantly improved image quality. This was attributed to a more even distribution, leading to reduced variance. This is in line with our experiments.

Pre-generating photons

Mara et. al [Mara et al. 2013] opted to decouple the generation of photons from the shading steps, only doing the photon generation and storage building a single time. The aim was to support real-time shading using a photon map. The advantage of this method is that photon generation and acceleration structure building no longer affect framerate, as they are part of preprocessing. Skipping the acceleration structure building is not possible when it stores photons in a view-dependent manner. Later on, we will discuss one such technique for which the storage system is built around the perspective of the camera. The downside of pre-generating photons is that doing this will cause lighting to be completely static, as photons are only traced once.

Adaptive photon distribution

Rather than sending out photons uniformly from the light source, some researchers opted to create algorithms to send more photons to directions where they are likely to have more impact [Yang and Ouyang 2021; Wyman and Nichols 2009; Estevez and Kulla 2020]. Yang and Ouyang's Adaptive Anisotropic Photon Scattering uses a new workflow for photon mapping which contains multiple steps throughout the rendering pipeline that combine to generate a task buffer for the next frame. This task buffer describes a way of distributing the generation of photons. It does so by maintaining a 2D texture and a quadtree that are updated every frame.

All light sources are contained together in that 2D texture, in a tiled manner. The quadtree defines a division of work, defining for every thread / task ID which light source and information it needs to find out which direction it should sample. To obtain this information, a thread repeatedly uses their task ID to choose which of the four options it should go deeper into. The described implementation of the task buffer is claimed to be a better alternative to uniformly generating photons, needing fewer photons, especially when combined with anisotropic footprints. This method is described in its entirety in section 4.

Rasterization based

In 2009, image space photon mapping was introduced [McGuire and Luebke 2009]. It proposed to do the first bounce of photons through rasterization. In doing so, it achieved major speedups. It has later been applied mostly in high-performance photon mapping approaches, amongst which an adaptation named screen-space photon mapping [Kim 2019]. This approach does not extend to area light sources, though. Rasterization also causes a fixed resolution, meaning if the light source is close to one object and far from another, the far object will receive fewer photons, and as a result less detail in lighting. In addition, these performance gains may not extrapolate well into the future. This is because ray tracing is rapidly improving in performance and is already faster than rasterization for

large triangle counts. Geometry of game scenes also gets more and more detailed as hardware improvements are realized.

Anisotropic splats

Anisotropic shapes enhance photons by projecting photons in an elliptical area, rather than a disc. It works particularly well in a photon splatting pipeline, which splats photons onto surfaces by turning them into flat ellipses. Before anisotropic footprints can be implemented, a way to calculate their shape in an informed manner is required. For this, photon differentials (explained in section 2.2.4) have been used in practice. Anisotropic photons require little extra engineering or computation to splat, compared to isotropic photons. Elliptical shapes often better represent photons, because the difference in compression of photon distribution can be quite different between the ellipse's two different axes. A clear and frequently occurring case is when a diffuse bounce is made at a low angle of incidence, somewhere on a photon's path. For these reasons, anisotropic photons improve final image quality [Yang and Ouyang 2021]. Anisotropic photon splatting is described in full detail in section 4.

2.2.2 Photon storage

The reason photon storage matters and we do not just use an array is *performance*. Without a good storage type for photons, an obscene amount of photon pixel combinations would need to be checked to produce an image. The most basic data structure, an array, exemplifies this: doing a range query for a single pixel on it takes *O(N)* time, *N* being orders of magnitude too large, as it is equal to the total number of photons. It is only natural then, that storages are mainly judged by performance measurements.

A good photon storage data structure has two key properties: It allows for fast range queries (i.e. return all points in a radius around a point) and fast build (i.e. inserting/updating is efficient). Additionally, low memory usage is desirable. This is similar but not quite the same problem as is typically faced in ray tracing, and as such data structures and heuristics which are typically optimal for ray tracing like BVHs and the surface area heuristic might not be optimal for photon mapping.

Pre-generating photons does not allow for dynamic lighting. In order to get fully dynamic lighting, the new storage has to be built or updated with the new photons every single frame, meaning build performance is just as important as shading performance. In addition, to support dynamic lighting, building the storage needs to be possible on massively parallel graphics hardware. As performance constraints are tight, it is important to maximize it by providing workflows which fit within the SIMT architecture.

kd-tree

Jensen proposed to build a balanced *k*d-tree containing all the photons. A *k*d-tree is a space-partitioning data structure which can organize geometry, including points, in 3D space. They make it possible to quickly find photons at or near a specific location, without having to do a brute force search through an entire array. Zhou et. al found that it was possible to build *k*d-tree reasonably fast using graphics hardware in 2008 [Zhou et al. 2008]. With the rapid advancements made in GPU technology over the last 15 years, building a *k*d-tree is believed to be fast enough for real-time use on state-of-the-art hardware. Wald also covered parallel-focused and fast *k*d-tree construction [Wald 2022].

Because reversing photon mapping is also an option, instead of building a *k*d-tree containing photons, building a *k*d-tree containing pixels instead is also an option. It was also found that using a

voxel volume heuristic to build a *k*d-tree instead of building a balanced *k*d-tree improved the performance of photon map queries by a factor of between 1.3 and 5.8 [Wald et al. 2004].

Bounding volume hierarchy

Similar to a *k*d-tree, a BVH could also be used. There has not been much research on using a BVH for photon mapping, but their prevalence in state-of-the-art ray tracing has gone hand in hand with a plethora of support and knowledge. This includes both BVH building and traversal. Acceleration structures in DirectX12 are typically implemented as BVHs under the hood, and include functionality to build them directly on the graphics processor, at real-time rates.

In addition, most modern graphics processors support hardware acceleration structure traversal. Together, these building blocks may be altered into a highly experimental acceleration structure that contains photons. To traverse this acceleration structure, very short rays can be used. We have implemented this and performed brief experiments with the implementation, as can be read in section 5.2.

Block Hashing

Ma and McCool originally introduced a technique called Block Hashing [Ma and McCool 2002], pointing to latency problems for *k*d-tree based *k*-NN due to the serial dependencies in memory accesses. In Block Hashing, hash functions that take constant time are used to categorize photons by position. As is described in section 3.3.2, this technique does introduce some error, though in practice, it was found that this error is hardly noticeable.

There may also be problems in terms of constructing these cells on a frame per frame basis. While hashing itself and finding a block lends itself well to GPU architecture, issues may occur if multiple photons are being entered into the same cell at the same time. In addition, memory requirements of individual cells will vary per frame, as the number of photons differs per cell per frame.

Increasing the memory footprint of all cells might work but would be undesirable. Appending a pointer to an additional cell would also work albeit not ideal for performance. Despite these engineering challenges, block hashing may prove to be worth it due to its algorithmic efficiency and simplicity.

Later, an adaptation of block hashing was made, dubbed a HashGrid [Mara et al. 2013]. In the introducing paper, the importance of low collision count with the used hash function was stressed. This is because threads that experience no collisions are idly waiting until every other thread is completely done evaluating the hash function. A novel approach also exists which can compress information and uses neural networks to disambiguate hash collisions, all while being parallelizable on the GPU [Müller et al. 2022].

Tiled photon storage

The Tiled approach was found to be one of the best overall options for real-time shading using a photon map due to its exceptional performance and quality [Mara et al. 2013]. In short, it divides the camera space up into frustra, and puts copies of each photon into all relevant frustra, if any. This allows for easy access to relevant photons when doing shading.

While the tiled approach has some great advantages, it does not extend to cases where camera rays are transformed outside of the frustrum, e.g., when looking at a mirror that displays objects behind the camera. Constructing the tiled storage is very similar to the block hashing storage and can thus be expected to perform similarly in construction. The main difference is instead of hashing photons,

they are structurally stored based on which frustum tile they fall into, both are inexpensive constant time steps.

Splat array

While it is technically possible to use a plain array to contain the photons for regular photon mapping, it would be prohibitively slow as the number of operations required to do brute force photon mapping is orders of magnitude too large. The only known technique that is able to do photon mapping with just an array, while doing it within a reasonable amount of time, is splatting.

Splatting turns photons into an elliptical shape by splatting them onto a surface. The ellipse's size is calculated from the photon's differentials. For isotropic approaches, this ellipse is simply a disk. This ellipse is used to directly rasterize photons onto a render target with the same size as the screen. Because splatting does not need a special storage system, a lot of time is saved that would otherwise be spent building and querying a large storage system containing many photons.

Reducing photon footprint

In addition to storage solutions, Mara et al. also looked into how to reduce the size of a photon in memory, while still containing the photon's position, power, direction, and probability. They found they could reduce photon's size to 18 bytes, or 12 bytes when storing photons in a Hash-Grid or Tiled representation, without observable loss in quality.

While obviously being good for lowering memory usage, lowering the memory footprint of a single footprint also brings with it potential performance improvements. Better cache and register utilization, and fewer memory accesses leading to less latency for some types of photon storages, would result. In practice, the memory footprint of photon mapping implementations is small. A forward looking estimate based on our measurements including the generation of as many as 5 million photons would still only equate to 90MB worth of pure photons, reduced to 60MB with this compression scheme. Storage overhead from potential acceleration structures would also remain unaffected, resulting in less impact caused by compression.

2.2.4 Photon Shading

At the very basis, what the different variants of shading methods have in common is that they all need to match photons with pixels. While the storage type that is used is an important aspect of shading, some methods also differentiate themselves by innovating on which photons are and which photons are not used when calculating a point's irradiance estimate.

All shading methods are embarrassingly parallel, either on a per-pixel or a per-photon basis. There is never a dependency between threads. Most methods also inherently come with high occupancy in a SIMT context. However, with some acceleration structures such as a *k*d-tree and BVH, some more engineering is required to achieve high occupancy rates during traversal.

k-NN

A *k*-NN query or *k* Nearest Neighbor query was originally used to estimate irradiance for the points. When using a *k*d-tree, the algorithmic complexity of shading a point was shown to be *O(log N + k)*. In practice, fewer than *k* steps are required for darker parts, because fewer than *k* photons exist within the radius. Despite this, the lack of recent work with *k*d-trees makes it difficult to argue in favor of their use.

Lowering the radius improves performance, because if the radius is smaller, many pixels require the inclusion of fewer photons, especially in dark areas. Lowering the photon count also improves performance for the same reason, but mostly in brighter areas. However, both also lead to noisier

output as both naturally lead to higher variance due to fewer photons being included in the estimate.

Approximate k-NN

Because of how Ma and McCool's block hashing works, some photons are falsely omitted when gathering for *k*-NN. Because of this inaccuracy, they named their shading approximate *k*-NN They stated image quality was virtually the same with their A*k*-NN implementation in comparison with *k*-NN. This statement poses the question whether approximations can be applied with other photon storage systems, too, without reducing image quality. In their block hashing algorithm, photons are stored in blocks. To find which block a position belongs to, the position is hashed.

Splatting

We define splatting as the act of transforming a photon from a point into a 2D shape (an ellipse, in our case). Photon splatting is a technique which benefits from hardware acceleration by using the rasterization pipeline to render photons [Stürzlinger and Bastos 1997]. As shown in recent work, when combined with a fast generation step, this allows for a complete rendering solution using photon mapping that is easily within real-time limits [Yang and Ouyang 2021]. Splatting has been combined with rasterization in the past, one downside of this is that rasterization does not work on its own if the light is refracted or reflected before reaching the camera.

Splatting makes use of additive blending informed by the geometry buffer blend photons onto a render target. It does so by rendering the light caused by an individual photon directly onto the screen, using the surface of the affected rasterized triangle as a surface to splat the photon onto. Because the render target has the same resolution and projection parameters as the regular render target, rasterizing splats is only supported inside the viewport.

Splatting is similar to older photon mapping methods that stored lighting for each triangle into a lightmap texture that belongs to the triangle. These older methods have the advantage that the shading step later is as simple as a texture lookup. They have largely been forgotten because the fixed resolution of these textures brought too many limitations, but extensions to combat this may be possible. On the contrary, because photon splatting enables the rasterizer to render photons at the screen's resolution, it does not suffer from these same limitations.

Cascaded caustics map

Another technique with real-time applicability is called ray guided water caustics [Yang and Ouyang 2021]. Similar to the splatting approach, it has a way of mitigating the fixed resolution problem that occurs when rendering to a texture. However, instead of solving it by rendering to the screen using the triangles as intermediates, it renders to several caustic map cascades. Caustic map cascades work off the same principle as shadow cascades: there are several maps which caustics are rendered to, each of different sizes in the world. The caustic maps are ordered in ascending order, the smallest one being used for the caustics that are nearest to the camera.

While this technique is very fast and at the same time capable of providing sharp and physically realistic images of caustics, it falls short on a few occasions. For example, it does not extend to scenes containing any other caustics than those caused by a single body of water. As a first step, the algorithm does a rasterization step starting at the light source, effectively mapping the water surface onto the light source. Then, it uses a grid to figure out which photons would have landed where. This approach does not work for area light sources, as they cannot be rasterized to. While the predetermined resolution works well for a large relatively flat seafloor or bottom of a pool, artifacts

stemming from the distribution of this resolution may become visible when less favorably shaped objects are placed underneath the water.

Adaptive radius

In 2013, a new technique called Adaptive Progressive Photon Mapping [Kaplanyan and Dachsbacher 2013] was introduced which utilized a locally adaptive radius for photon gathering. Like progressive photon mapping, it starts by creating a map of the pixels and then does iterations of generating photons. Between iterations, the technique finds parts of the image where variance is high based on previous iterations and adaptively increases the size of the gather radius. This effectively does an intelligent and optimal tradeoff by lowering variance in favor of bias where it is necessary.

Photon differentials

In 2007, Photon differentials [Schjøth et al. 2007] were first introduced, applying ray differentials [Igehy 1999] to photon mapping. It was found that regular photon mapping leads to blurring of important features, notably caustics, caused by photon mapping's bias that results from the collection radius when shading. When used to intelligently change the radius of photons, photon differentials proved to significantly improve image quality, as seen in Fig. 13.

For isotropic photons, photon differentials are obtained by taking the partial derivatives of the individual parameters of the photon and combining these derivatives. For anisotropic photons, the partial derivatives are applied to the scaling of the axes of the ellipse. In practice, obtaining the derivatives is done by performing small perturbations of the parameters and tracing these slightly altered paths along with the original photon. While adaptive progressive photon mapping and photon differentials vary greatly in terms of approach, they both reaffirm the same core idea: intelligently changing the impact radius of photons is worth doing.

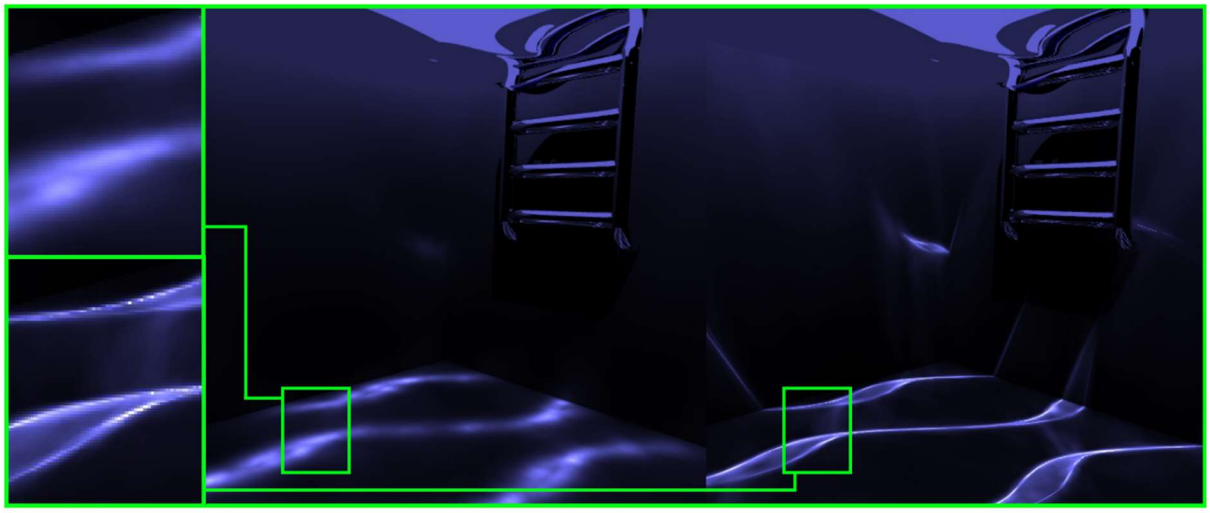


Figure 13 - An underwater image rendered without (a) and with (b) photon differentials. Photon differentials allow for sharper caustic features, allowing for overall better image quality. (Image by Schjøth et. al)

2.3 Adaptive Anisotropic Photon Scattering (AAPS)

We consider adaptive anisotropic photon scattering to be the most effective method in a real-time setting. One of the reasons is its tested performance far exceeds any other techniques. It also supports many types of scenarios in which caustics are cast, including light patterns caused by stained glass windows. Fig. 14 shows an example image that displays what kind of quality caustics the method is capable of rendering in a real-time setting. Overall, it provides major algorithmic improvements that allow for caustic rendering that is so fast it makes real-time use a possibility while leaving most of the frame's budget for the rest of the rendering pipeline. For these reasons we want to give a more detailed overview of how the method works before we continue.

Before going into the step by step process of running AAPS, a few definitions that are important to know are:

- Light atlas coordinates. Light atlas coordinates are simply the pixel coordinates in the ray density texture. Multiple lights can be present in the ray density texture, and its resolution can be adjusted to match the desired amount of detail.
- Ray density texture. The ray density at a light atlas coordinate defines how many photons are sent into the corresponding range of directions. For a directional light, rather than a range of directions, it defines a range of origins.
- Projected area texture. This texture keeps track of the average projected area photons emitted at a light atlas coordinate have. A higher projected area leads to a higher ray density, this is controlled by the target area. When the projected area is larger than the target area, the number of samples gets increased (through the ray density texture). When it is smaller, the number of samples gets decreased.
- Variance texture. This texture keeps track of the average variance photons emitted at a light atlas coordinate have. A higher variance leads to a higher ray density. The effect of variance on ray density is multiplied by variance gain.

While AAPS is based on photon mapping, in a way, it is the most different from regular photon mapping out of the photon mapping based technique that exist. Where regular photon mapping follows the three steps of: photon tracing, building a photon storage, and photon shading, the AAPS workflow consists of four different steps that we will explain in more detail:

1. Emit photons. The number of photons sent is increased in directions where they are likely to contribute to visible caustics. Trace and record the photons. Additional information about photons that is required for later use is also recorded.
2. Rasterize photon splats onto the caustics render target. The authors refer to the process of rendering photons through the rasterization of splats created from photons as scattering.
3. Apply the caustics render target to the scene.
4. Compute the next photon distribution, by combining the previous one with the newly obtained data.

Figure 14 - A scene with many glass objects that cast caustics. The scene was rendered in real-time, and caustics were rendered using AAPS. (Image by Yang and Ouyang)

2.3.1 Emitting, tracing, and recording photons

To define the adaptive sampling distribution of photons that exit a light source, every light source has a 2D texture. Each pixel in this texture corresponds to a range of directions, if the light source is a point light. The exact way these directions paired with texture coordinates is unspecified, but we go into specifics of what requirements we maintained in section 3.1.2. If a directional light is used instead, each pixel corresponds to a 2D range of light origin points. The value in a pixel of this texture defines the ray density within that pixel. The actual number of rays that will be sent in the direction range that corresponds to that pixel is equal to the nearest square number less than the value. Rounding it down to a square value allows for an easy distribution of samples in a grid pattern. The textures for all the light sources are all combined in one larger tiled texture.

Before this texture can be used to guide rays towards the correct parameters, it needs to be converted to a quadtree that can be traversed by a thread. This is done by executing a number of iterations of a compute shader each building the next layer on top of the previous layer. Building the quadtree comes with a low computational cost. For a thread to traverse this quadtree given a task ID, it can loop through the mipmap levels, choosing the applicable subtree out of the four at every level. It repeats this until it reaches the end at which point it will have the necessary information to generate the ray. Fig. 15 shows an example of this traversal process.

After the photons have been generated, the tracing of photons follows an approach akin to path tracing, potentially with a high number of maximum bounces. During the tracing itself of photons, in order to maximize effectiveness of the work done, the authors propose to exclude unimportant or overly costly photons. For example, photons with a footprint that is too large can be resized or excluded, and photons with energy levels below a threshold can be discarded. The authors do not specify exact thresholds or criteria to define which photons to cull, or which factors determine their culling, but there is a large degree of freedom here.

Whenever a photon intersects an opaque surface, it is recorded in the photon buffer and its footprint area is added to target area. Because the algorithm is described to only be used for caustic

rendering, only photons that contribute to caustic light are recorded. Other types of light paths are excluded from AAPS.

Finally, in order to change the photon's footprint and additionally allow for anisotropic footprints, photon differentials (described in section 2.2.4) are also tracked during the entire tracing process of a photon. Depending on whether the light source is a point light or a directional light, the parameters for a photon are either two positional parameters, or two directional parameters.

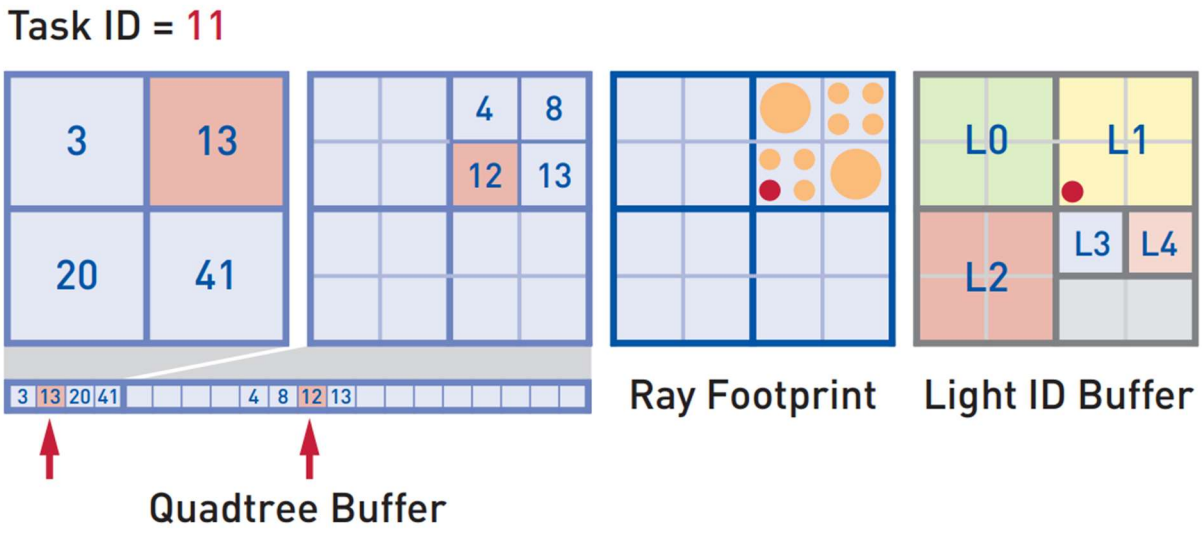


Figure 15 - An example of how a thread uses a task ID to traverse a quadtree and find the correct set of ray parameters to use. On the first level, shown on the left, the top-right node is traversed because the Task ID is larger than 3 and not larger than 13. On the next level, the bottom-left node is traversed because the Task ID is larger than 8 and not larger than 12. After finding the correct leaf, a position on the final grid is chosen based on the Task ID, as seen in the 3rd image. (Image by Yang and Ouyang)

2.3.2 Rasterization

After the first step is complete, there is a buffer which contains every photon and all the necessary information to rasterize it. Fig. 16 shows an example of how a splatted photon can be rasterized to render it. The information needed to splat a photon includes the photon's intensity, incident angle, and the differentials. Photons are then constructed as two triangles (akin to a quad) which is done based on photon differential information. Surface attributes are retrieved from the G-buffer, similar to deferred shading.

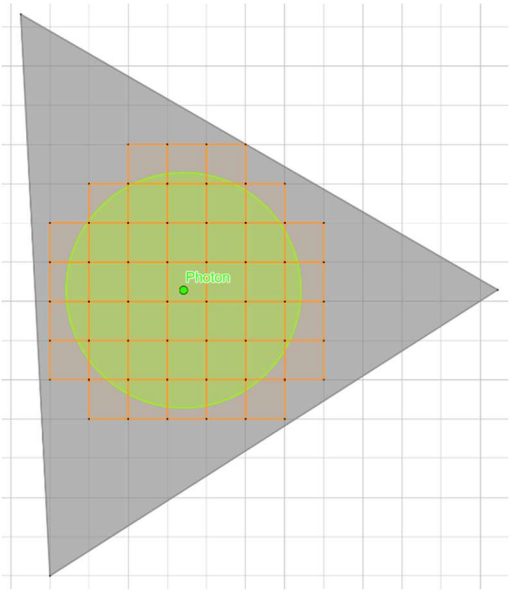


Figure 16 - A photon has been splatted onto a triangle's surface. After this, it is rasterized to the screen. All highlighted pixels are affected by light from the photon, in this example.

These triangles are then rendered to the caustics buffer, which is a render target specifically for caustics. This is done with additive blending. After all photons have been rasterized, variance gets recorded by comparing the caustics value of the center pixel of a splat (the pixel that contains the photon's position) to the caustics value at the pixel it would have ended up on in the previous frame. To get the new pixel position, the newest projection matrix is used to transform the photon's position. To get the previous pixel position, the projection matrix from the previous frame is

used to transform the photon's position. Then, the difference from last frame is computed for the photon and added to the corresponding value in the light source's variance texture.

2.3.3 Apply caustics buffer to the scene

Applying the caustics that were directly visible to the screen is trivial, as each pixel on the caustics buffer directly maps to a pixel on the screen, and simple addition suffices. Rendering caustics that are indirectly visible is also possible, as long as they are directly visible somewhere else on the screen. Image space photon mapping [McGuire and Luebke 2009] does not originally support indirectly visible caustics. However, in 2009 Wyman stated that an older existing technique [Wyman 2005] with simple modifications would suffice to achieve deferred shading for refraction [Wyman and Nichols 2009].

2.3.4 Generating next frame's task buffers

At the end of the frame, a new ray density guide value is computed for every single pixel in every light source's guide texture. The formula as described by the authors is as follows

$$D' = D\frac{A}{A_t} + vg$$

In the article, the parameters are described with the following: "*D'* is the suggested ray density, *D* is the previous ray density, *A* is the average screen-space projected area, A_t is the target projected area, *v* is the caustics variance, and *g* is the variance gain". A_t and *g* can be altered to influence photon size and temporal flickering, respectively. The rate of change of *D'* is kept in check by blending with neighboring pixels, to avoid local instability. We do not have enough information to know how problematic temporal stability is, but it may be possible that maintaining both temporal stability and adaptability is in some cases impossible to do while maintaining sufficient quality. Configurable temporal and spatial weights are used to control the balancing, according to the formula:

$$D_{\text{final}} = w_t D' + (1 - w_t)\frac{\sum_i w_i D_i}{\sum_i w_i}$$

2.3.5 Soft caustics

When describing AAPS, the authors also showed how to include area light source parameters into photon differentials. This could further improve image quality when area light sources are used as it allows for soft caustics as photon footprints are directly dependent on photon differentials. Unfortunately, the adaptive sampling they introduced in their method has not been extended to area light sources yet, so while soft caustics can be included in the full AAPS pipeline, the area light sources that are required to cast them are not fully supported.

3. Implementing Adaptive Anisotropic Photon Scattering

Most of our experiments were designed to learn AAPS's characteristics, and how to improve AAPS. AAPS is significantly faster than any other existing method at rendering caustics. We believe this is mainly due to three factors of improvement it achieves. The first factor is that the task buffer significantly reduces the number of unnecessary photons, which saves costs for both tracing and shading them. The secondary factor is that no acceleration structure needs to be built or traversed to shade points based on these traced photons. Rasterization is a direct way of finding all pixels that are affected by a photon. The third and final factor is the anisotropic footprints that are based on photon differentials. It yet again improves the effectiveness of photons, by adjusting their size and shape to better fit the distribution of light based on its differentials. The difference this makes is particularly noticeable in the cases with high anisotropy. This happens when the distance between photons along one axis is small, but the distance among the other axis is large. Disk shaped photons would be too small to bridge the gap between photons along the axis with large differences. Making them larger to compensate for this would result in oversized photons, leading to a blurry image.

AAPS was implemented using DirectX12 with DirectX Raytracing (DXR). The implementation was not optimized on a low-level, focusing mostly on algorithms. For reproducibility purposes, culling was also not applied in our implementation, unless specified. More detail on culling in section 3.3.5. DXR allows the use of fast ray tracing which benefits from ray tracing hardware. Ray tracing using DXR requires acceleration structures, which have to be built using DXR, too. They support triangles or procedural primitives with an axis aligned bounding box around them, but we only used triangles. These acceleration structures are typically implemented as a BVH under the hood.

The process of splatting and rasterizing photons is referred to as scattering. We purposefully make the distinction between the splatting and rasterization components, as that distinction helps clarify our proposal to replace rasterization with ray tracing from section 5.2.

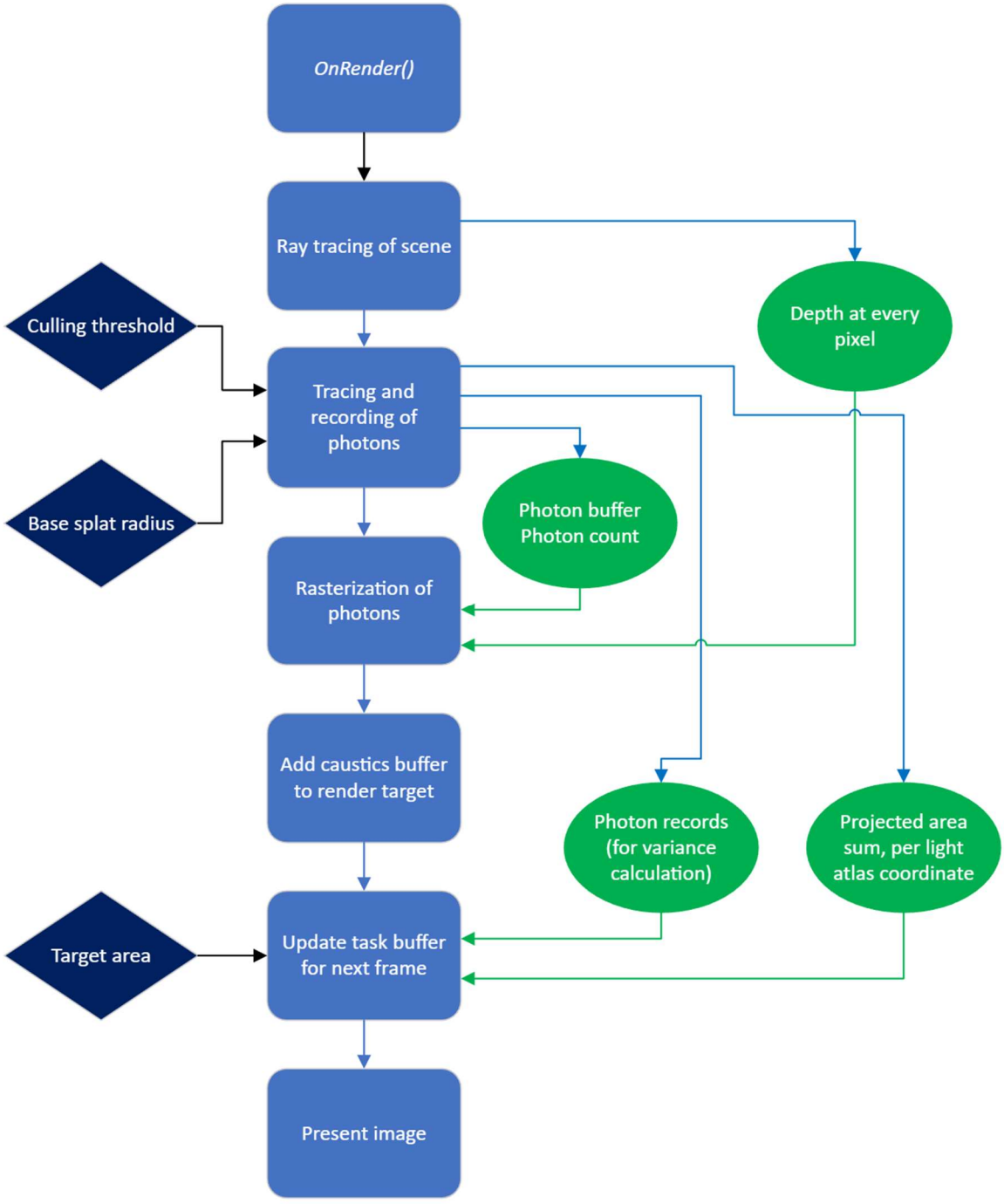


Figure 17 - The AAPS pipeline in our implementation. Rectangles (rounded) denote steps,dDiamonds denote parameters, ellipses denote information that is passed from one step to the other.

3.1 AAPS pipeline

For AAPS to work properly, we had to design a fitting pipeline and work out its components.

3.1.1 Ray tracing of scene

The first thing our renderer does every frame is ray trace the scene, and figure out which triangle occupies which pixel. In our case this is done with ray tracing. Depth information for each pixel is also recorded. Additionally, direct lighting is evaluated using a single shadow ray. The cost of recording depth information is negligible, assuming the scene is ray traced or rasterized irrespective of whether AAPS is used or not. Additionally, calculating direct lighting is also not a part of the AAPS

algorithm. For these reasons, we have excluded this first ray tracing step in its entirety from all performance measurements.

3.1.2 Generating, tracing, and recording of photons

In the next step, a number of ray generation shaders at least as large as the number of photons to generate is dispatched. Every shader instance uses its ray index to traverse the density texture quadtree and find out which leaf it belongs to. This leaf is a value in the density texture, and it represents a region on the sphere of directions around the light source. A leaf of the quadtree always contains a square number. This square number defines the number of samples sent into the leaf's region, and these samples are distributed in a square grid shape through the leaf's parameter space.

To convert a 2-dimensional task buffer to a range of directions, the 2D space is mapped to a sphere using a function that takes two parameters and returns a normalized direction, equally distributing the inputs. It is important that the distribution is equal. For example, using the two parameters as longitude and latitude leads to a higher concentration of samples along the poles, which leads to bias. Photon differential rays are also generated using the mapping function, and this helps tracks how much distance there is between samples, and thus how big photons should be made.

After the generation shader has dispatched its photon, it is traced against the scene, and its differential rays are updated along with it. If its first intersection with the scene is not a dielectric object, it is discarded. Otherwise, the photon is kept alive, and interactions are handled until it intersects with a diffuse surface, at which point it is recorded or discarded. How/if photons are split up when interacting with a dielectric primitive is explained in section 3.3.1.

Before checking whether a photon is accepted or not, its footprint is calculated using the photon differential rays. To determine whether or not a photon is accepted, two things are verified:

- The photon must be located in the camera's frustum. This is easy to check. The photons have already been converted to triangles, so regular triangle frustum culling will work.
- The photon's area size as seen from the camera must not be larger than the culling threshold. The area can simply be calculated using the photon's footprint, and the distance of the photon to the screen is used to convert it to the correct scale.

When photons are accepted, their anisotropy is limited to a maximum value by reshaping them if their anisotropy is too high. Then, the necessary information is recorded to three different buffers:

- The projected area buffer, which keeps track of the total projected area of all the photons emitted with specific light atlas coordinates.
- A rasterization buffer, which redefines the photons as triangles to be rasterized. This also contains the brightness information.
- A photon records buffer, in which the data necessary for computing the variance is stored for each photon.

3.1.3 Rasterization of the photons

As all of the important information has already been recorded in previous steps, rasterization itself is not very complicated. In our implementation, the ellipses are represented as two triangles, similar to quads. The vertex shader treats the incoming vertices as any other and simply projects them from world coordinates to the screen, before passing the parameters on. The barycentric coordinates of the quad are assigned to the triangle vertices and can also simply be passed on and interpolated.

The pixel shader verifies that its inputs are within the ellipse, and matches the depth from the G-buffer at that pixel. If either of these are not the case, it returns a black value. Otherwise, it returns the photon's brightness. Before the rasterization process can start, the photon count is kept on the GPU and passed into an argument buffer using a shader, so that it does not need to be transported to the CPU.

3.1.4 Adding the caustics buffer to the render target

Before presenting the image, the caustics buffer has to be additively blended to the render target. Assuming the render target does not contain caustic light, and the caustics buffer only contains caustic light, this operation has an extremely low cost as it simply involves adding two textures together. For MSAA implementations, which we will get into later, the caustics buffer is first resolved from a multisampled caustics buffer to a unisampled caustics buffer, before the unisampled render target is added to the final render target.

3.1.5 Updating the task buffers

In order to ensure a good distribution of photons in areas where they are likely to end up visible on the screen as caustics, the task buffer gets updated every frame. The cost of this is minor but it is vital to the algorithm, as it allows the rendered images to have a high quality while keeping the number of unnecessary rays to a minimum, resulting in performance suitable for real-time games or other applications, while leaving most of the frame time for other rendering schemes. Fig. 18 displays an action shot of the most important task buffers. Updating the task buffer is done as a sequence of compute shaders. The first thing we do is updating the variance texture.

Calculating the variance

To get a density texture value that improves quality in areas with more action, the variance of every pixel in the density texture is also recorded. If a difference value for a photon cannot be found for any reason, we use a value of 0 instead. To be able to calculate variance, when a photon is recorded during ray tracing, three of its properties are also put into a separate array. These properties are necessary for calculating the variance and they include:

- The photon's world position.
  This is used to find the center pixel of the photon and reproject it to its position in the previous frame. The difference between the values of these pixels is then used to calculate the difference which will contribute to the final variance.
- The photon's light atlas coordinates.
  This is used to trace back which variance value the difference value should be added to.
- The fraction of energy that the photon has.
  This is used to easily calculate an average of the variance. The fractions of all photons within a density texture pixel sum up to one.

Updating the density texture

After the variance values have been calculated, the density texture is updated. First, a suggested density value is calculated. This uses the projected area value and variance value to calculate a new density value. This density value is blended with its previous value and the previous value of its four neighbors. In order to avoid areas permanently stopping to get sampled, we set the minimum value for the suggested density value at 4 samples. Because this is combined with our relatively high resolution of the density texture, the overhead of our implementation is on the high side.

Building the quadtree

Finally, to make the ray generation of the next frame feasible, the ray density texture is converted to a quadtree. The cost of this is low. In our implementation, this is achieved by executing multiple compute shaders, each computing the next level of the quadtree up towards the root node. Our quadtree implementation differs slightly from the explanation in section 2.3.1, and contains the sum of all leaf nodes below it as its highest value.

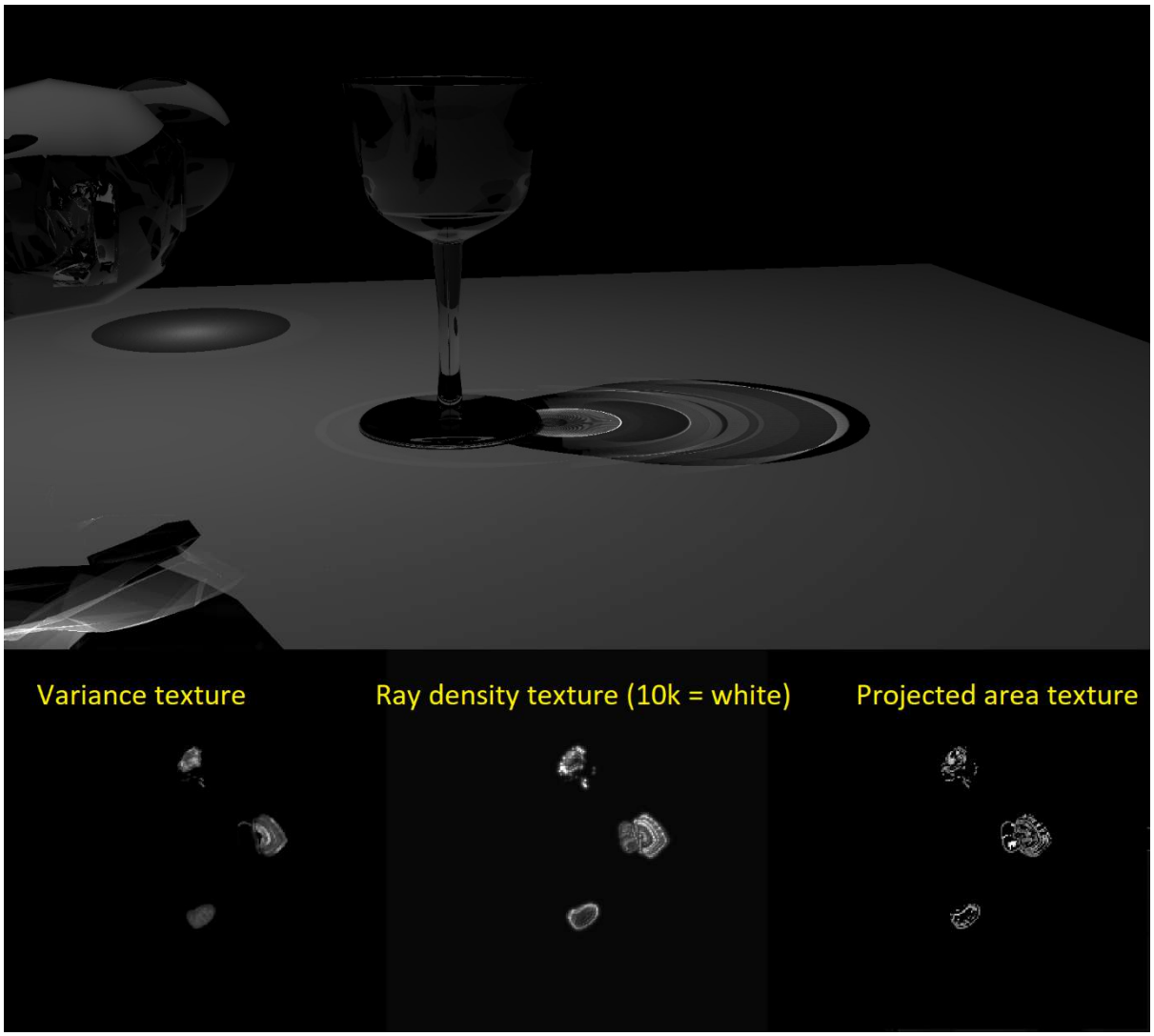


Figure 18 - A visualization of the variance texture, ray density texture, and projected are texture in action. The camera was moving slowly to the left, when the screenshot was taken.

3.2 AAPS parameters

3.2.1 Brightness

Determining a photon's brightness can be split into multiple steps. The total energy the light source puts out is divided equally over every pixel in the density texture. The energy attributed to every pixel is equivalent because they all cover a range of the same size. So to find the amount of energy emanating from an individual partition, we simply divide the total energy of the light source by the number of partitions.

As controlled by the task buffer, the number of photons sent in a partition can vary significantly. However, to maintain the same light levels, the energy emitted in a partition should always remain the same. Getting the energy of a singular photon is simple similar to obtaining the energy within a

partition. To obtain the energy contained by a single photon, we take the energy emitted within a partition and divide it by the number of photons we sampled the partition with.

After that, the energy of a photon has been determined. However, it still needs to be converted from energy to brightness. We account for the size of the photon while maintaining the same energy level, by dividing the energy by the area of the splat to obtain its final brightness.

3.2.2 Initial parameters

Density texture resolution

The density texture resolution has a significant influence on the performance and quality of AAPS. Its only hard constraint is the necessity to be able to build a quadtree from its values. It is technically possible to allow for any shape of density texture, as nodes of the quadtree can be adjusted to contain fewer than four values. For simplicity, we chose to only support square shaped density textures, with a power of two as width and height.

In all of our experiments, we used a density texture of 256x256. This is higher than the resolution the original authors used in their experiments, which was 128x128. Because our density texture has four times as many values, the overhead incurred from updating it is larger. The overhead from tracing photons in inactive areas is also higher, as the values are always kept above 1, but it is still very manageable.

Clamping density texture values

The density texture has a minimum value to avoid getting stuck in an infinite cycle where areas of the light source, or even the entire light source, are never sampled. We set this minimum to 4, to ensure a high level of quality where caustics that should get rendered are not missed. This means that the base number of photons emitted is *256*256*4*, or 262,144. We also experimented with an upper limit for the density texture value. In some cases, this helped with temporal stability and keeping the density texture under control. For all of our experiments to collect data, we opted not to use a maximum value though.

3.3 Eliminating ambiguity

Implementing AAPS is a task that is open for interpretation in multiple ways. This section, we eliminate as much ambiguity as possible in terms of how we implemented our version of AAPS.

3.3.1 Photon splitting

Whenever light hits a dielectric object such as glass or water, part of the incoming light is always reflected. In most cases, some of the light is also refracted. The most realistic looking thing to do would be to have our photons match this by splitting them up into two appropriately bright photons whenever they interact with a dielectric object. In practice, this is unfeasible because the number of photons quickly gets out of hand and severely impacts performance.

We deal with this by only splitting up a photon a maximum of one time. After a photon has been split, its two parts are marked as already having split, and will no longer split up. Instead of splitting up, a photon will repeat its previous interaction. In other words, it will refract if its previous interaction was a refraction and reflect if its previous interaction was a reflection.

We found this approach to be a good compromise between high performance and high image quality. We also considered always splitting up photons, and discarding photons whose energy became lower than a certain threshold. In the end, we opted against this option due to temporal stability concerns. When objects or rays slightly move, and photon paths are slightly altered, paths

that were just under the threshold could end up just above it, or the other way around. This might lead to unwanted flickering.

3.3.2 Photon differentials

In order to get two photon differentials, we create two rays with very slight perturbations in the two input values for the direction generation. These two rays are updated as if they intersect the same triangle as the main ray. Evidently, the incoming direction will have been slightly altered which changes the direction of refraction or reflection. Due to different values in the normal interpolation, the normal is also slightly different. Even if the triangle hit by the main ray would be missed by the differential rays, we update them as if they would have hit the triangle, to get a more consistent value resulting in a reasonably sized photon splat.

3.3.3 Elliptical shape

While we want to render photons as ellipses, the rasterization pipeline works with them as pairs of triangles. In order to convert a pair of triangles to an ellipse, we add barycentric coordinates of the quad they make up (ranging from -1 to 1) to the vertex attributes. In the pixel shader, when the sum of the squares of the individual barycentric coordinates is larger than 1, the position on the splat is outside of the ellipse, and should be discarded.

3.3.4 Depth testing

In AAPS, photons are rendered through rasterization of so called splats. When a splat is obstructed by other geometry, from the camera's perspective, it should not be visible. For improved accuracy, we evaluate this on a per-pixel basis in the pixel shader. We also allow for a margin of error of one percent in the depth. This accounts for floating point inaccuracies and also allows splats on surfaces that are not perfectly flat to still be visible.

When rasterizing, to obtain the depth of a point on a splat, we take the distance from the camera's position to the interpolated world coordinates. Getting the scene depth for comparison at every pixel is both simple and uncostly. When rendering using ray tracing, record it with the first hit of the primary ray. When rendering using rasterization, get the depth information from the Z-buffer.

3.3.5 Culling

While the chapter that originally introduces AAPS highlights the importance of culling photons, it mentions it in relation to reducing the cost of the photon tracing stage. At first, we experimented with no culling at all. It quickly became clear that this was detrimental for rasterization performance. We took a pragmatic approach and plotted both root mean square error (RMSE), FLIP (an error metric discussed in section 4.2.1) and frame time in milliseconds against a maximum splat size. Photons exceeding that side would get culled. The results are further discussed in section 4.3. We chose our parameter value based on this graph, prioritizing quality over performance.

Brief experimentation led us to decide on using an approximation for the solid angle as measured from the light source: area divided by the square distance. Using just the world size of a splat would not have been versatile enough to work for scenes of different scales. To ensure highly sampled regions were not under-culled and regions with few samples were not over-culled, the solid angle was calculated before scaling the photon's size based on the density texture resolution and sample count.

Photons whose solid angle was bigger than the threshold size would end up getting culled. In the end we found a value for the culling threshold which was good for performance, RMSE and FLIP at the same time, as it significantly improved performance without negatively affecting image quality. The culling threshold had little to no effect in the Pool scene, though.

The reason this culling was necessary is that a small fraction of photon splats would end up with a very large size. These splats would barely contribute to the image but require obscene amounts of pixel shader executions to render. In some cases, the majority of the time spent on rendering was spent just on the pixel shader executions.

In order to explain why these large photon splats might show up in the first place, we recall that the size of photon splats is directly dependent on the photon differentials. Due to the way photon differentials work, the final differential is magnified by every interaction the photon made on the way to its destination. As a result, chaining several interactions may lead to severe discrepancies between a photon's ray direction and its differential ray directions. Due to the Pool scene's topology, it does not feature photons that change medium more than once, which is likely why it does not benefit from maximum splat size culling.

4. Evaluation

4.1 Scenes

To evaluate the effectiveness of AAPS, we created three scenes that feature caustics:

- The Pool scene (discussed in section 4.1.1)
- The Square scene (discussed in section 4.1.2)
- The Cornell scene (discussed in section 4.1.3)

As some experiments required a large amount of caustics to display significant and stable results, we have excluded the Square and Cornell scene from some experiments. Each of the test scenes was created using Blender. To maximize the usefulness of the task buffer, light sources were placed close to the geometry in all scenes. For simplicity, each of our scenes contained a single point light source, and no additional light sources.

The simplicity of the scenes used for evaluation should be taken into account in particular when evaluating the recorded ray tracing performance. In terms of geometry placement all three scenes are very simple, lending itself well for BVH traversal. In the Pool scene, more than 99% of triangles are part of a single water surface.

The water surface in the Pool scene is nearly flat, and made up exclusively out of quads, which means if the acceleration structure has been built well, the y axis is almost irrelevant. As a result, the number of acceleration structure nodes that is unnecessarily expanded is likely to be smaller than usual. The reasoning for this is as such: intersecting a nearly flat box which neatly fits one or more quads is almost identical to detecting whether a triangle was hit, so the odds that a flat box is intersected but the quad inside of it is not, is small.

A different way to look at it is, acceleration structure nodes are hardly unnecessarily expanded is that there is almost no chance for a ray to intersect multiple triangles, as they are never stacked. The splatting and rasterization of the photons itself is not be affected by scene detail: the number of photons and shader executions does not depend on the geometry of the scene. Instead, the number of photons depends on the task buffer, and the size of the splats depends on the differentials.

Every dielectric object in each of the scenes has an index of refraction of 1.5, which closely matches real life glass. While water has an index of refraction of 1.33, we still used 1.5 for simplicity. This may slightly affect the way the images look. The reference images match this value in all scenes.

In addition to the three main test scenes, we added a fourth scene specifically to display the capabilities of our indirect caustics extension. This scene contains a yellow torus inside of a glass cube and is known for its difficult to render light paths. This scene was not used for our main experiments, and just used to exemplify the capabilities of our method.

4.1.1 Pool

For one of our test scenes, we created a pool with a water surface. The light source was placed above the water, and the camera was placed under water. The reason the camera was placed underneath the water surface is because we wanted to observe refractive caustics, and AAPS would not be able to render them without the camera being placed underwater. More on this limitation can be read in section 4.5.2.

The scene consists of a water surface which was created by using blender tools to deform a plane, and a pool which “contains” the water, made up of very simple geometry. The water surface accounts for nearly all of the 781,276 triangles within the scene.

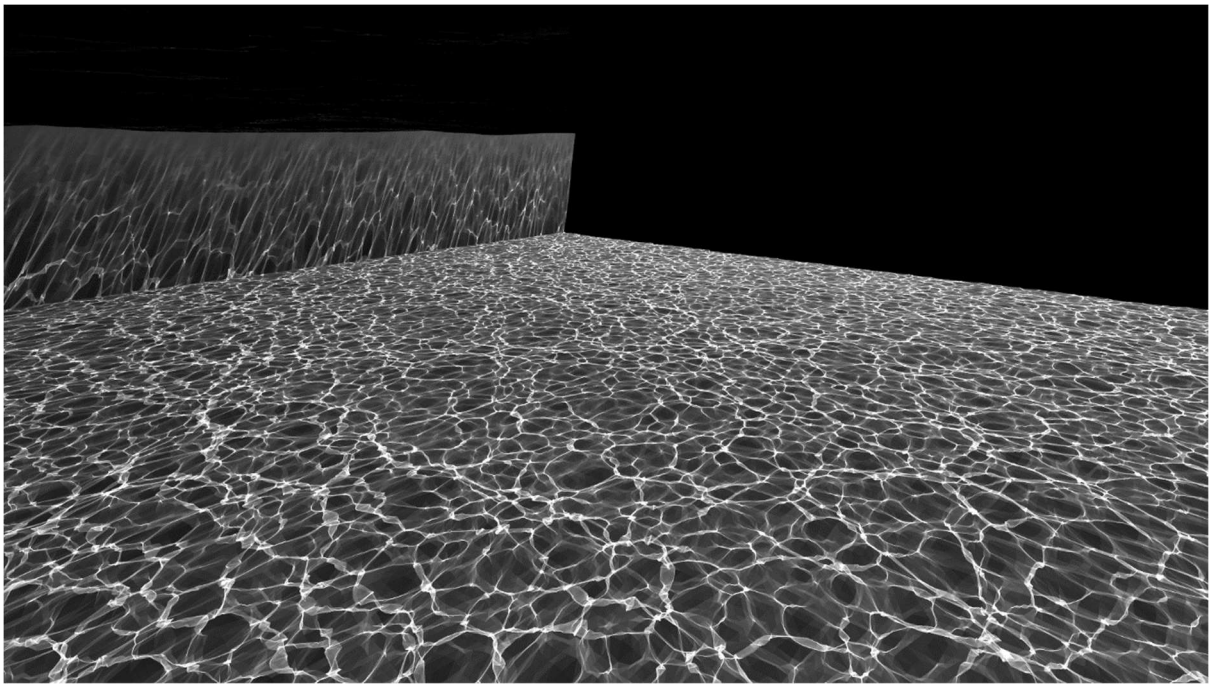


Figure 19 - The reference image of the Pool scene.

4.1.2 Square

The Square scene contains a few different objects all casting caustics on a square floor that is flying in the void. It contains 42,668 triangles in total. Of those triangles, the wine glass accounts for 21,218 of them. The monkey head, also known as Suzanne, is made up of 968 triangles. The glass sphere is an icosphere and is made up of 20,480 triangles. Unsurprisingly, the square floor that the scene was named after consists of just 2 triangles.

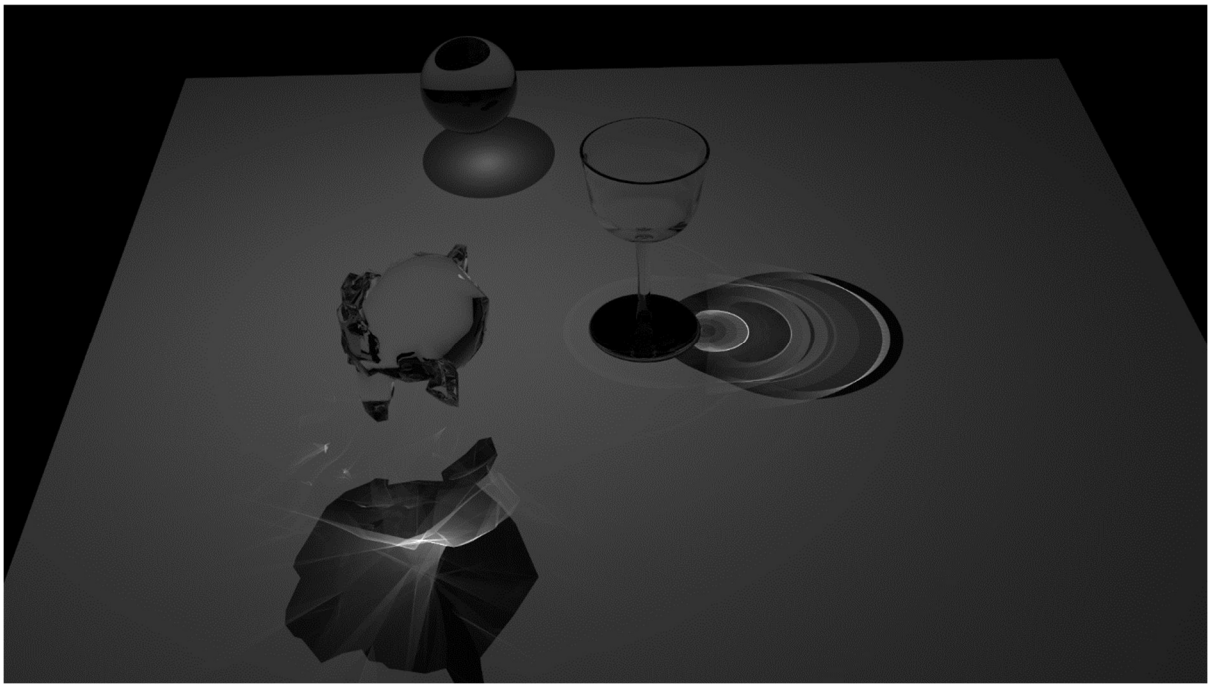


Figure 20 - The reference image of the Square scene.

4.1.3 Cornell

While the “Cornell” scene was inspired by the Cornell Box, it is a unique version with different object sizes and placements. As our implemented renderer does not support colors, the Cornell scene is a black and white scene. The outer box also only has 5 faces, with an opening on the side of the camera. Since neither our reference images nor our own renderer support indirect light, this does not matter, though. The sphere it contains is an icosphere, made up of 5,120 triangles. The other faces are all quads, made up of just two triangles.

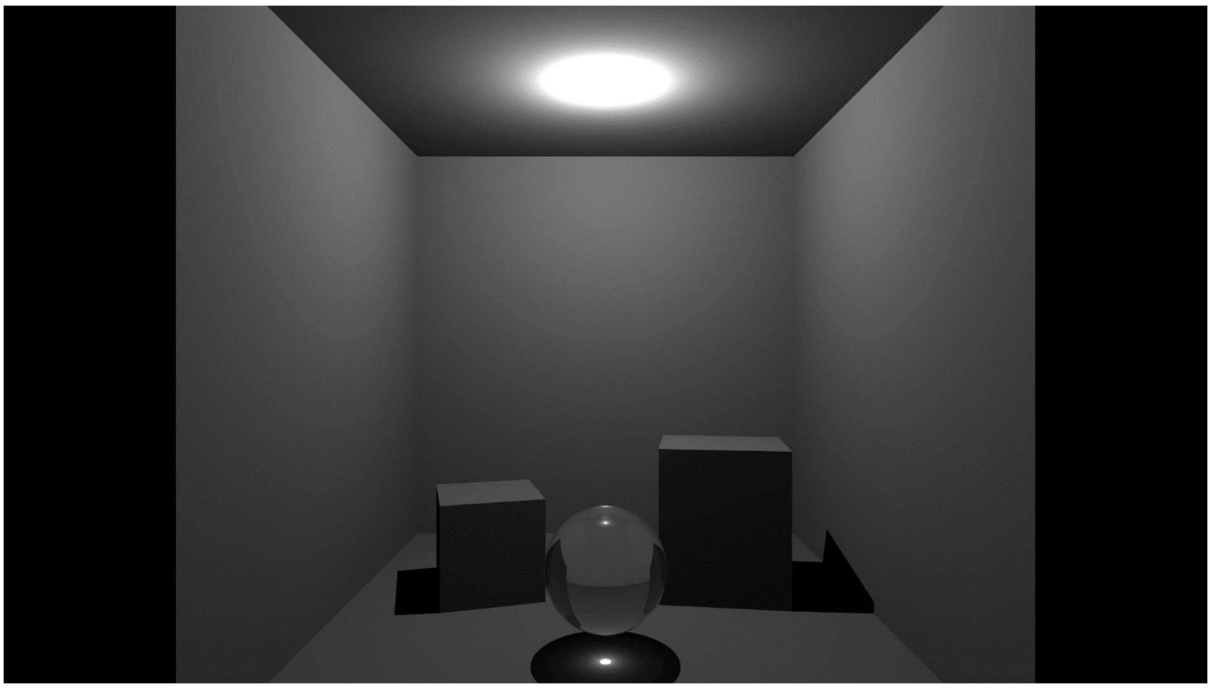


Figure 21 - The reference image of the Cornell scene.

4.2 Quality evaluation

Evaluation and comparison of images was a critical requirement to drawing conclusions on the quality of rendered images. To do this, we employed two image comparison techniques to compare the rendered images to a reference image. Reference images were produced using thousands of samples, and include the same types of light paths as our renderer. More can be read about this in section 4.2.3. Aside from evaluation techniques, visual inspection was also important, to verify results and to spot problems that are not easily visible from the error metrics. One such problem was fireflies. Fireflies are more likely to show up as more samples are taken and the splat size decreases. These have a significant negative influence on perceived image quality and we believe their impact does not proportionally affect FLIP or RMSE value. Another problem we identified with visual inspection was the overall lower light levels in renderings that used MSAA.

4.2.1 FLIP

FLIP [Andersson et al. 2020] is an image difference evaluation technique that is designed and proven to agree with errors perceived by humans. It scores significantly better than similar techniques, which is why we chose to employ it. Our main use for FLIP is to measure the quality of the produced images, between parameter values and anti-aliasing choices. For our results, we used the mean FLIP value. Note that a lower FLIP value is desirable. Identical images result in a FLIP value of 0. Using FLIP to compare a completely black image to a completely white image results in a FLIP value of 0.97.

4.2.2 RMSE calculation

Aside from visual inspection, one method we used to measure image quality was to calculate the root mean square error, based off reference images. Root mean square error is a simple metric which compares two lists of values. As our images are in black and white, we did not have to account for multiple colors when calculating the error of a pixel. To calculate the square error of a pixel, we scale the intensity of both values to a value in the range of 0-1, subtract one from the other, and square the result. The RMSE value is then obtained by taking the average of these and taking the square root of that. We mainly used RMSE to measure convergence as it is a basic error metric that clearly shows basic convergence patterns without doing much processing on the data.

4.2.3 Reference images

In order to measure the quality of the images produced by our implementation, we produced reference images to compare our images to. We used the LuxCoreRender plugin for Blender to create our reference images. The reference images for the three main scenes were rendered with at least forty thousand light samples per pixel. As our own renderer only features direct light and caustics, we also disabled indirect lighting when creating the reference images.

In order to match our image as closely to the reference image as possible, all images were saved either without compression, or with lossless compression. A gamma correction of 2.2 was applied for the reference images, as well as our renderer's images. In addition, we calibrated our renderer's light calculations (how much light corresponds to which pixel value) to match LuxCoreRender's by creating a scene with a point light source far above a flat plane and matching the intensity between both renderers.

4.3 Measuring parameter impact

Balancing parameters for AAPS is not a trivial task. In order to get a better understanding of AAPS, and how parameter choices influence the results, we have done a multitude of measurements with different parameter settings, on different scenes. This was done in a sequential order, one parameter after another. After a parameter was experimented with, the data was collected, and graphs were made, we looked at the results and chose a good value based on these results. This value was then applied and used for further experiments.

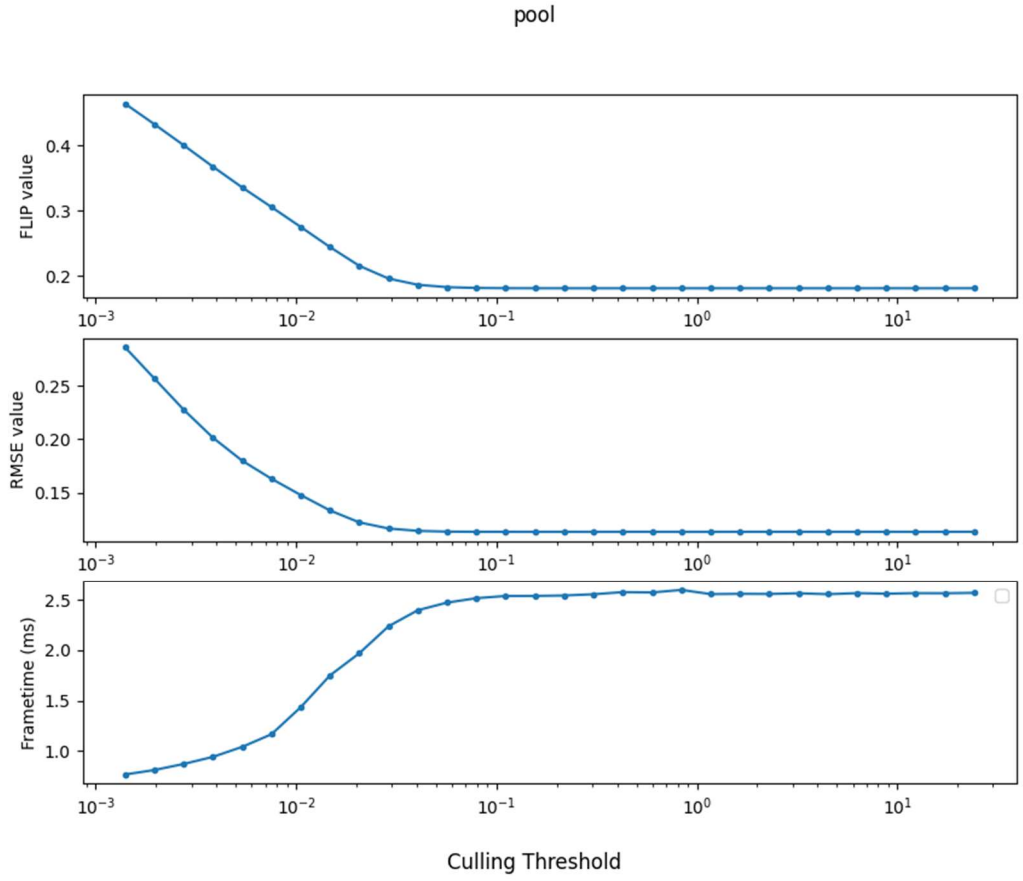
Our experiments with setting the **culling threshold** showed that culling large photon splats is definitely worth it. Fig. 22 shows that in two of our test-scenes, it is possible to achieve significant gains in performance with negligible change in image quality. Visual inspection also confirmed this.

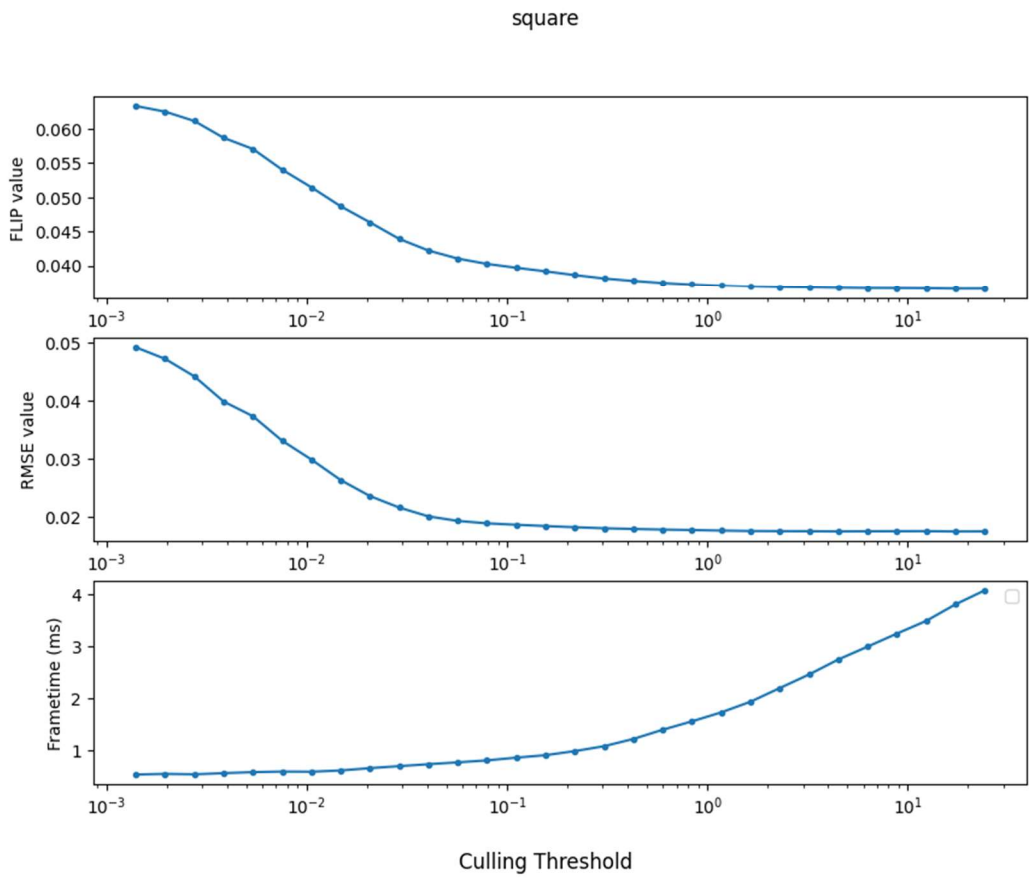
Note that the execution time graph in the Pool scene differs from the other two graphs: It seems to be shifted to the left. We think this is because in the Pool scene, the splat size is much smaller on average than in the other two scenes. This in turn happens because the caustics in the Pool scene are exclusively made up of single-event caustics, while the other two scenes contain mostly caustic light resulting from multiple interactions with a dielectric.

Splats recorded after more interactions have a much higher chance of becoming quite large. Each interaction drives more space between a photon's primary ray and its two differential rays, including their position and direction. As a result, their value grows much larger as more interactions happen, and so does the corresponding photon splat.

Also note that the RMSE and FLIP graphs starts off at very different values for the three scenes. This is because we are measuring error across the entire image, and some images have a much higher percentage of light coming from caustics than others. The first data point shows a case in which virtually all photon splats are culled, which is obviously undesirable, and as such the FLIP and RMSE

are highest there. Matching the error values, the Pool scene contains the most caustics, as most of the screen is covered by them. The Square scene contains a medium amount of caustics, containing them in three locations. Finally, the Cornell scene contains the least caustics, only a single small spot below the sphere is covered by them.





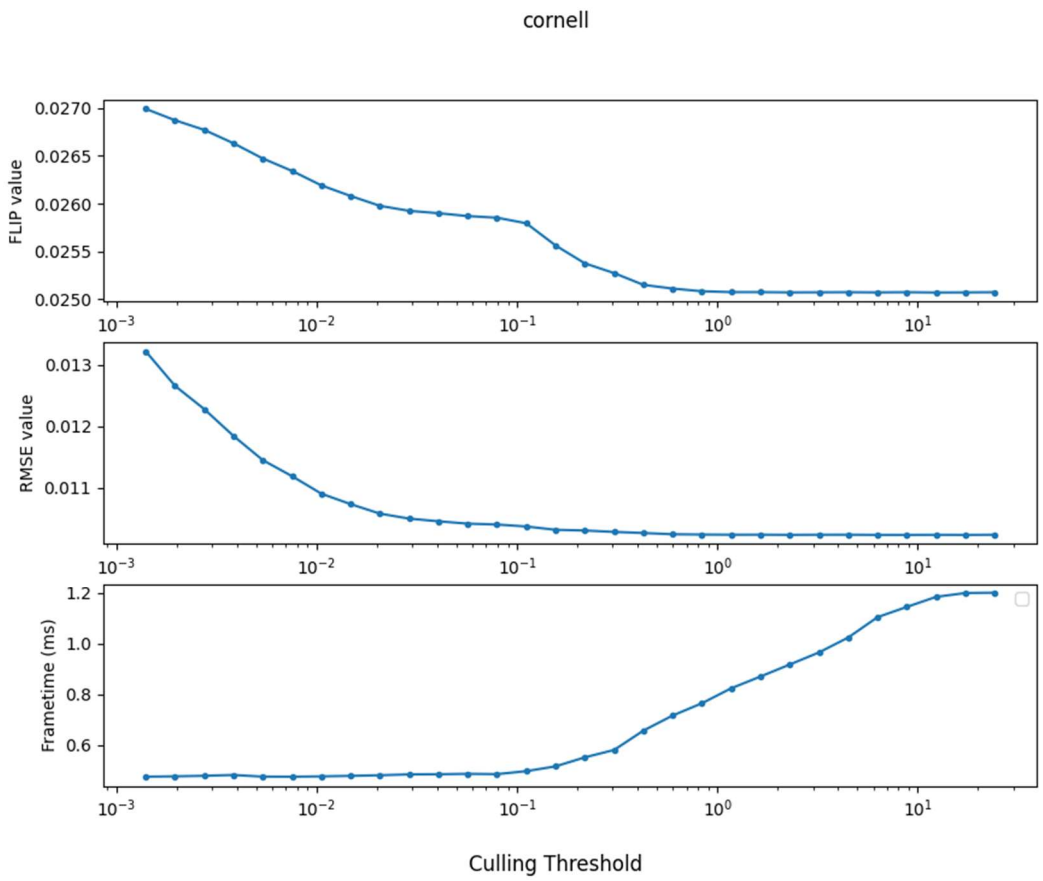


Figure 22 - The effect that culling large photons has on performance, RMSE, and FLIP.

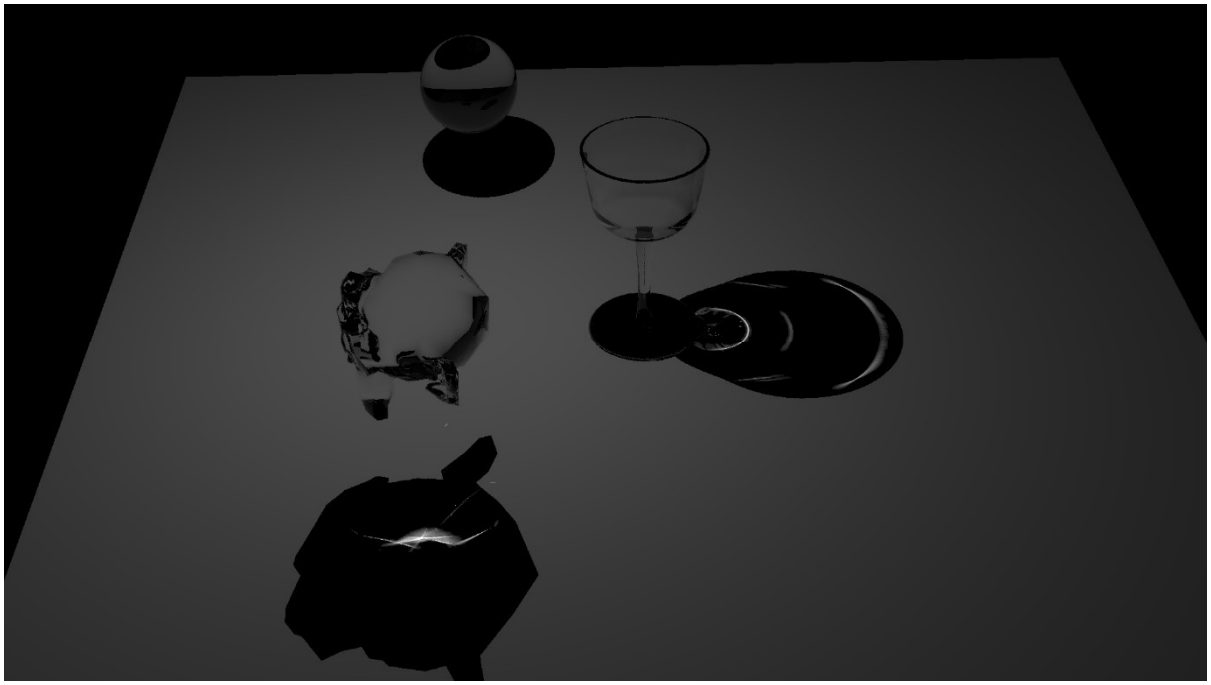


Figure 23 - The image with the lowest culling threshold (0.0014). Only the parts where the photon differential is smallest and the caustics are the most concentrated survive the culling. As a result, most important caustics features are lost.

4.4 Performance

Due to limited existing work, and the recency of AAPS, the previously available performance data for it was very little. In order to improve understanding of the method, we have done measurements measuring performance of our base version of AAPS on different types of hardware. For additional insights, we also include details on how much of the frame is spent on every part of the algorithm. All performance data was recorded at a resolution of 1920x960.

Acceleration structure building is an important part of any ray tracing application, especially for scenes with high dynamicity. The cost of this is always the same irrespective of whether AAPS is applied or not. We do not include building of the acceleration structures and the direct lighting computation time in our execution times, as they are considered to be general rendering work, and are outside of the scope of this article.

4.4.1 Total render time

In line with expectations, the time it takes to render the image in its entirety depends mostly on the number of photons that are emitted. Performance distribution was similar between AMD and NVIDIA, though AMD performed slightly better at the low-end of quality settings, while NVIDIA performed slightly better at the higher-end of quality settings.

4.4.2 Baseline performance

For the Cornell scene, when the target area is set to the maximum value, the performance is close to the baseline performance as the density texture contains the minimum value almost everywhere. What mostly influences performance here is the density texture resolution, in our experiments this was set to 256x256. By looking at the tail end of the Cornell scene's graph in Fig. 27, we can inspect this. The duration is around 0.47ms on the AMD Radeon RX6900 XT, and 0.52ms on the NVIDIA GeForce RTX 3080. On AMD the ray tracing time spent was 0.24ms, the task buffer update took 0.16ms, and copying the caustics buffer to the render target took 0.03ms. On NVIDIA, the ray tracing time spent was 0.24ms, the task buffer update took 0.23ms, and copying the caustics buffer to the render target took 0.03ms. Another factor that influences this baseline performance is scene complexity, which was on the very low end.

4.4.3 Ray tracing time

The ray tracing performance is directly dependent on emitted photons, as all emitted photons have to be ray traced. Though adding emitted photons and splatted photons is perhaps a better indicator, because the only computation time that is guaranteed to be necessary for emitted photons is finding the closest hit. At the same time, every splatted photon guarantees that at least one more ray was traced to find it.

Ray tracing always took up more than half of the cost of the algorithm but became a slightly bigger part as the target area is decreased and the emitted photons is consequently increased. This is because the cost of other parts of the algorithm does not scale as much with emitted photons. It ranged from 52% to 58% of the total cost in the Pool scene. For the Square scene, it ranged from 50% to 67% of the total cost. And for the Cornell scene, it ranged from 52% to 55% of the total cost.

4.4.4 Rasterization time

The rasterization time is largely determined by the number of splatted photons, as seen in Fig. 24, 26 and 27. However, it does not scale as much as ray tracing for instance. This is because when more photons are recorded, the size of photons is also decreased in equal proportions, to allow details to form. As a result, the rasterization pipeline has less work to do, because smaller triangles leads to fewer pixel shader invocations, and less raw rasterization work required to find them.

4.4.5 Task buffer updates

Most of the updates done in the task buffer update consist of the same amount of work every frame. We kept track of the recorded area in the ray tracing shaders, so that did not influence recorded task buffer update duration. Similarly, updating the ray density texture and building the quadtree consists of the same number of operations every frame, just with different input values.

Because most parts of the task buffer update are static, we do not see much change in the duration of it. However, the variance texture update is not static, as it has to process every single photon. This is why high splatted photon counts also leads to a slightly longer task buffer update duration.

4.4.6 Applying caustics buffer to render target

This is the fastest part of AAPS. Its duration was around 0.03ms with every scene, parameter configuration, and hardware type. We noticed a very slight increase of duration of this pass in some cases, up to 0.04ms. This did not happen for tests ran on the Cornell scene, possibly due to the caustics buffer consisting of mostly black pixels, while other scenes had much more variation in pixel values.

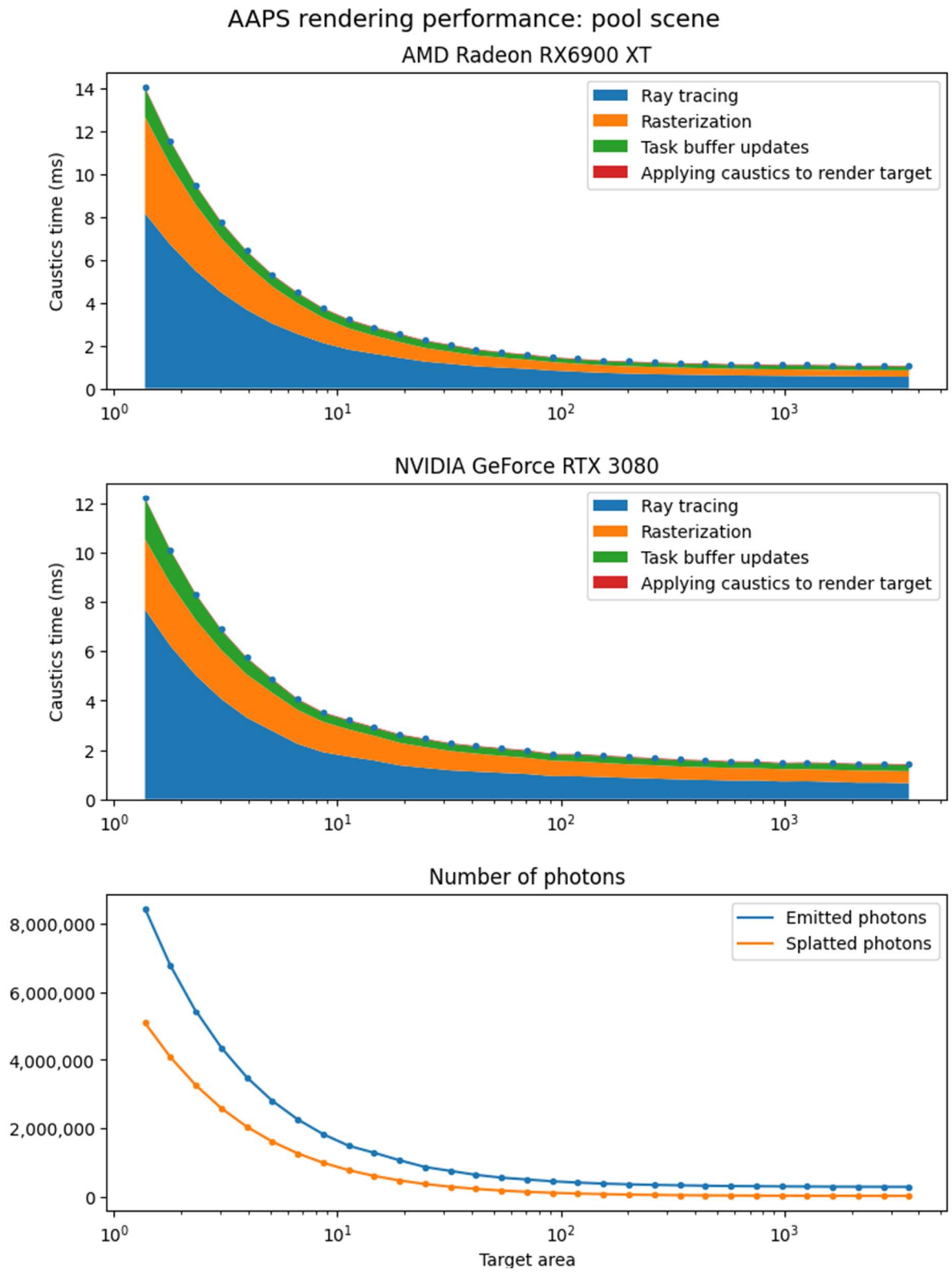


Figure 24 - The performance of AAPS rendering the Pool scene across a range of quality levels is displayed.

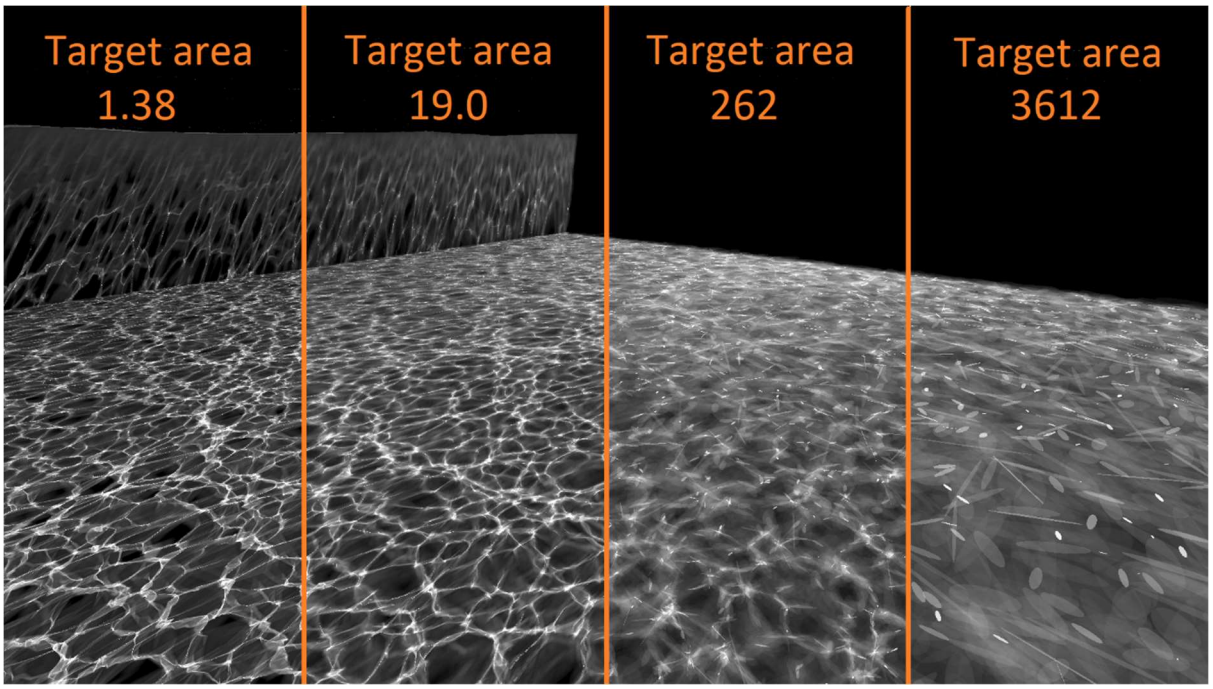


Figure 25 - A comparison of rendering quality with different target area settings. Caustic rendering times were 14ms, 2.6ms, 1.2ms and 1.0ms respectively.

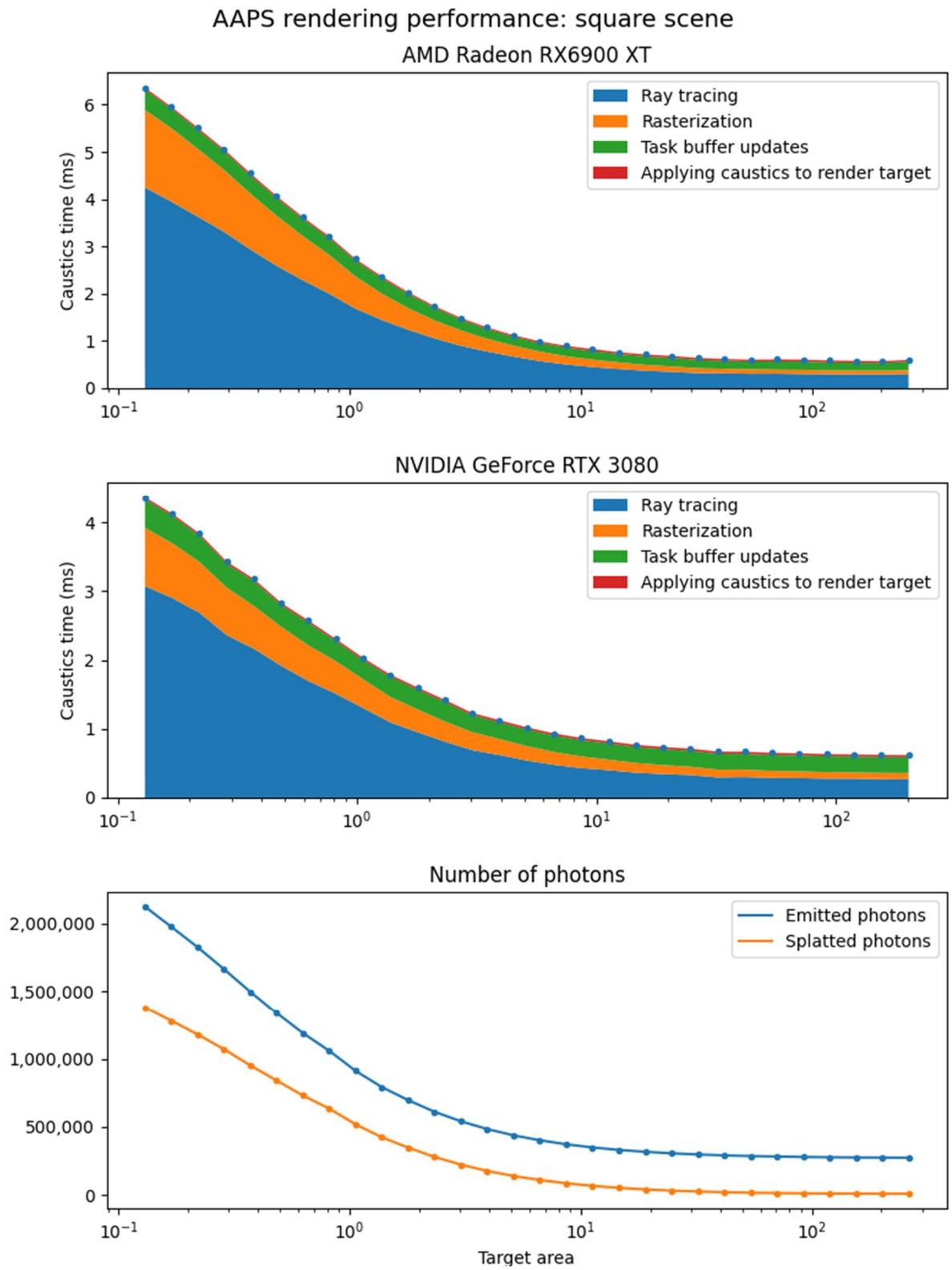


Figure 26 - The performance of AAPS rendering the Square scene

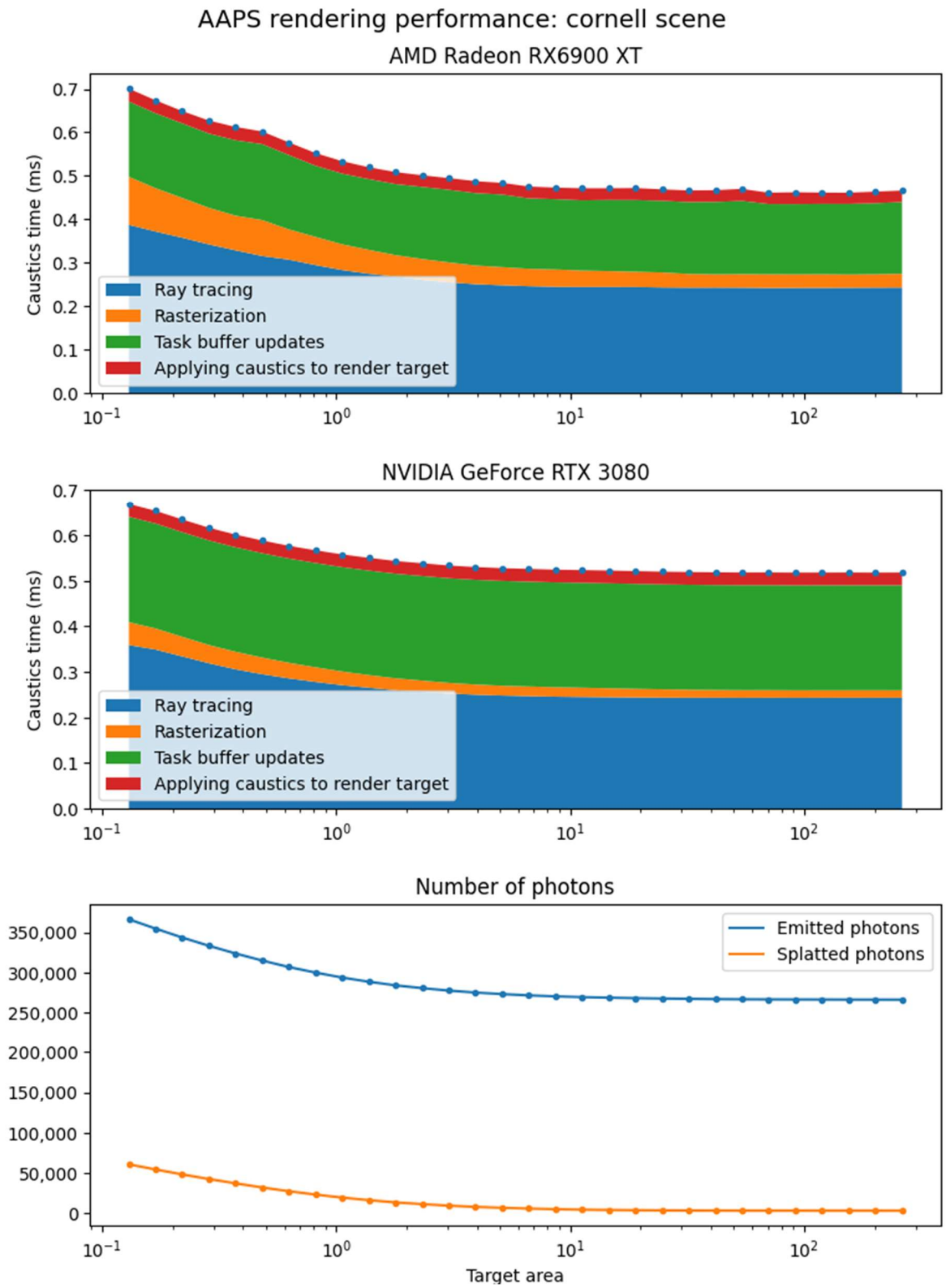


Figure 27 - The performance of AAPS rendering the Cornell scene.

4.5 Shortcomings of AAPS

4.5.1 Indirect caustics

When the rasterization pipeline is invoked to render a photon splat, its depth is compared to the camera's depth buffer, to check whether or not it is behind another object. This means that only caustics that are within the viewport and at the front of the image are rendered. When looking at a specular or dielectric object, we can still render caustics by looking their value up in the caustics buffer when evaluating light on a diffuse surface. However, this only works for caustic light that is directly visible from the camera. So in Fig. 28, the two photons would not end up being visible with AAPS.

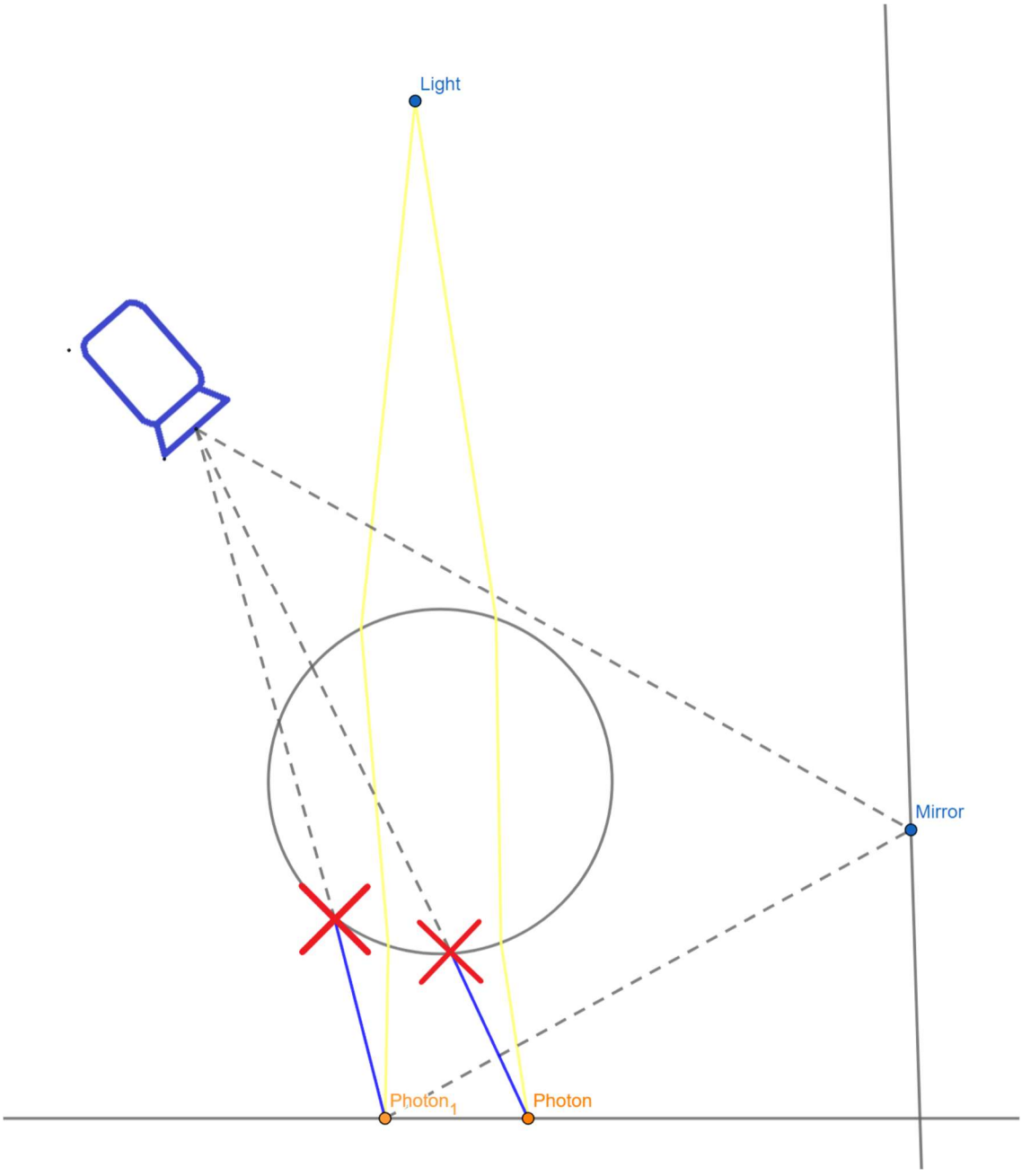


Figure 28 - An example that showcases how rasterization fails to connect photons to the image.

4.5.2 Limitations with rasterizing small primitives

Rasterizing small photon splats can come with some limitations. To understand why rasterizing small ellipses (or small primitives in general) can cause problems, we have to go into how rasterization works. As visible in Fig. 29, whenever an ellipse is rasterized, the question that is essentially answered is this: Which pixel centers are contained within this ellipse? With very small ellipses, there is a chance that even though the photon splat is bright and should be visible, it still does not contribute to the image.

Fireflies

One of the problems we saw in the images produced by AAPS, was fireflies. Our reasoning for why these are visible is as follows: A photon's intensity is divided by its area to maintain the same amount of energy. This means that photons with a miniscule footprint area end up very bright, if they happen to cover a pixel by coincidence.

Convergence

One of our experiments showcased a problem with convergence and AAPS. It measured the root mean square error (RMSE) per pixel, plotted against the target projected area of the splats. Recall that with AAPS a small target area directly leads to a higher sample count. It was our initial expectation that a higher sample count would also lead to a better image, however, our experiments showed that this was not the case, after a certain point. As is visible in Fig. 30, for the Pool test scene, the RMSE actually starts high, meaning the image contains a higher error on average. This happens for small values of the target area.

To further investigate what was going on, we performed an experiment to find out what percentage of photons actually contributed to the image. Measuring this was done by adding a flag to every photon, which records whether or not it has contributed to the final image. When a photon is initially recorded, this flag is set to false. Whenever a photon contributes to the final image, in other words, whenever the pixel shader returns a non-zero value, the flag is set to true for the corresponding photon. Finally, the number of contributing photons is summed up and saved.

Our experiments showed that the percentage of non-contributing photons significantly grows as the target projected area is reduced. We believe this to be due to the inherent inaccuracies that come with rasterizing small triangles without multisampling. The data from these experiments can be seen in Fig. 32.

To further understand what was going on we did another experiment with the target area set to the smallest value. In this experiment, we culled photons that were smaller than a certain threshold. We discovered that of the smallest 12% of photons splatted, only 22% actually contributed to the image.

So, while classical photon mapping does converge to the correct image, because the size of photons converges to an infinitely small value, this is not the case by default for approaches based on rasterizing photons. This is caused by triangles gaining higher chances to not contribute to the image at all, resulting from their decrease in size. However, it is possible to adjust the way we rasterize photons to improve convergence behavior, as we will describe in section 5.1.

Another limitation for convergence with AAPS is caustics caused by complex light paths. Light paths which have refractions chained with reflections are disabled in our renderer, for performance reasons. A way of culling photons with low contributions that works well with AAPS and scales well with quality does not currently exist: Setting a culling threshold will always cull the same light paths, even if the total number of samples that would have taken those paths rises. Randomly culling low-

energy samples is also not desirable, as that would disrupt the evenly spaced sampling pattern and also reduce temporal stability.

4.5.3 Complex light sources

Due to AAPS's task buffer texture approach, where the texture coordinates define the parameters, only two parameters are available for configuration. This works well for a point light or a directional light. With the point light source, the direction of a photon can be generated based on two univariate variables. For a directional light, the origin can be shifted based on the two univariate variables, and the direction is always the same.

Area light sources are particularly difficult. Compared to point light sources or directional light sources, instead of sampling a 2D space by sending photons, a 4D space now needs to be sampled. In addition to a complete hemisphere of directions the photons could have from any point, their positions also have two degrees of freedom on area light sources. As a result, the space of potential primary photons is much larger. How to balance such an approach and the feasibility of it are currently unknown.

Alternatively, area light source could be sampled equivalently to a collection of point light sources. This would simplify things as instead of sampling a 4D space, a 2D space would be sampled at a set of points. Each of these points could have their own task buffer, but a single task buffer could also be shared, emitting photons from all points simultaneously. However, in both cases the number of rays to trace and the number of photons to splat would be multiplied by the number of sample points on the area light, leading to poor performance if the same task buffer resolution is maintained.

When AAPS was first introduced, the authors also described a way to approximate caustics produced by area light sources. This method involves adjusting photon differential information on emission, which leads to bigger photon footprints and a blurrier image as a result. The downside is that the samples are all traced from the center of the area light, as opposed to being evenly distributed. This results in some inaccuracy. Caustics produced by small area lights can be closely imitated using this method. Caustics produced by larger area lights would suffer more from the inaccuracy but are also more suitable for rendering with other path tracing methods.

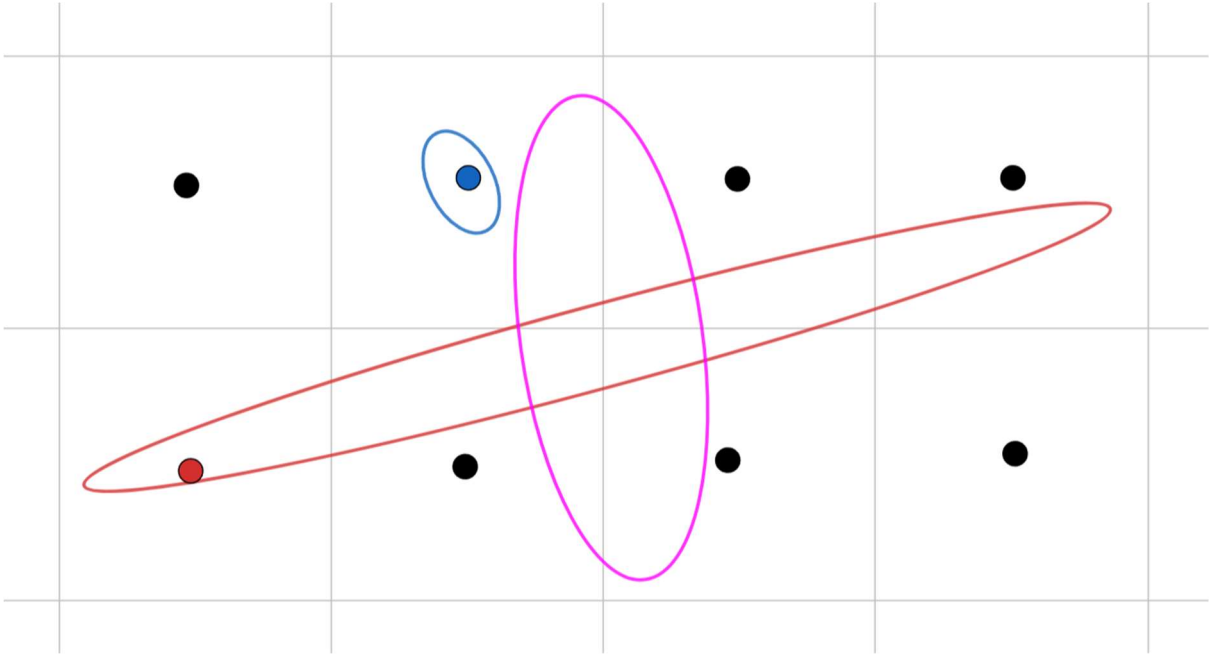


Figure 29 – Rasterization at a standard resolution fails to accurately capture the contribution of small ellipses

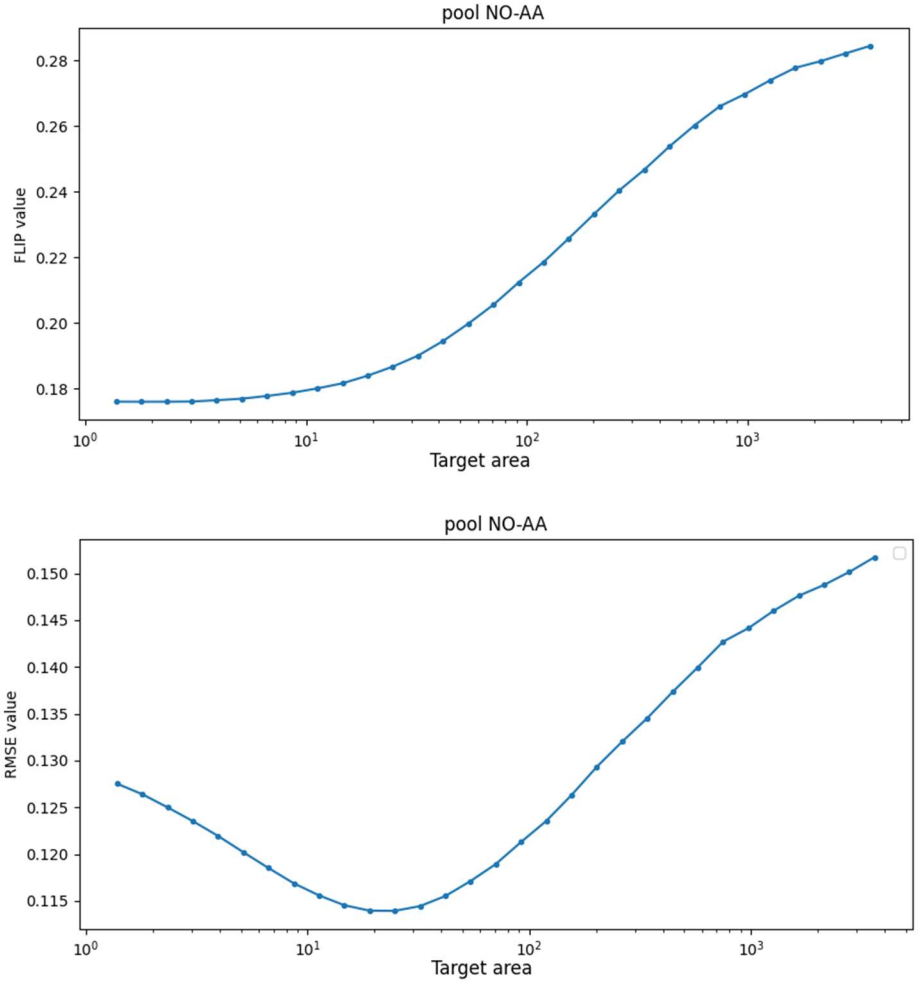


Figure 30 - The quality of the Pool scene rendering with different target area values.

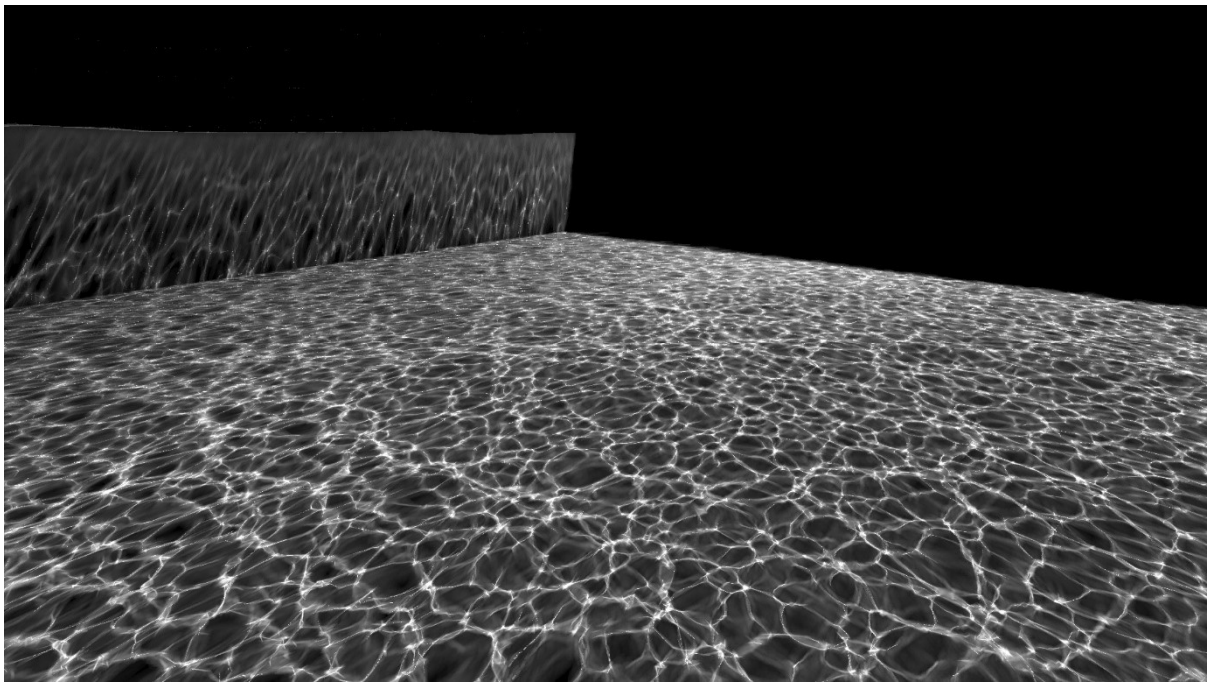


Figure 31 - The Pool scene rendered with a target area of 19 (the lowest and best RMSE value in the graph) using AAPS. The time to render the caustics was 2.6ms.

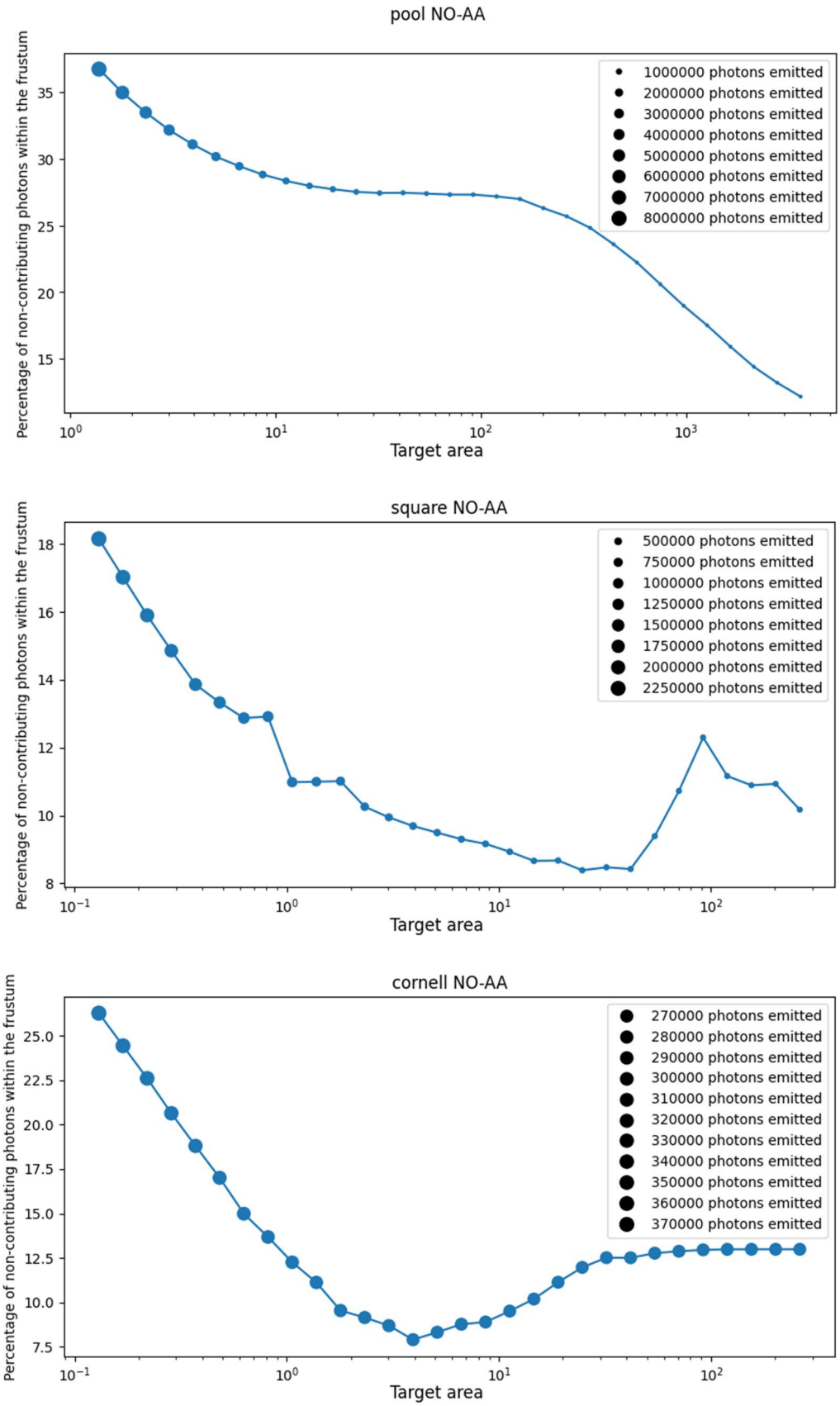


Figure 32 – Graphs showing the percentage of photons that did not contribute to the final image, plotted against the target area, for multiple scenes.

5. Improvements

5.1 Improving Convergence

As shown by our experiments and theory, rasterizing photon splats stops converging to the correct image as splats become too small to correctly render. We believe that it is possible to let the image converge further, or even indefinitely, for extra high quality caustics rendering. We propose to do this by increasing the resolution of rasterization, or using conservative rasterization, or even single-pixel photons. These techniques would suppress fireflies and improve convergence by better calculating the underlying photon's contribution through improving splat-pixel overlap accuracy. All of our MSAA variants used 8 samples per pixel. If maximizing the improvement to cost ratio is desired, it is possible that using fewer samples per pixel is better.

5.1.1 Multisample Anti-Aliasing

Experiment results

To get a better understanding of convergence behavior and image quality we added multisample anti-aliasing to the photon rasterizer. We expected superior image quality as a result of the higher sampling rate, especially for regions containing small but bright photon splats. Image quality seemed to be significantly improved, especially around fireflies. This is demonstrated in Fig. 35.

Contrary to expectations, our data showed that convergence and RMSE were not significantly affected by the addition of MSAA, and FLIP value seemed to be worse. As visible in Fig. 36, which includes the data with and without MSAA, the RMSE graph for the Pool scene looks nearly identical in both cases. The second surprise was that adding anti-aliasing made the error peak at the smallest target area value higher. Overall though, the RMSE was better with anti-aliasing.

Strictly looking at RMSE, values below 10 for the target area should not be chosen. This is because lowering the value below that point only has downsides. It increases execution time due to a higher sample count, while at the same time reducing image quality. For all data points, enabling MSAA resulted in a better RMSE value. So, assuming that a reasonable value for the target area is chosen, anti-aliasing always resulted in a better image within our experiments.

One notable difference between the images generated by AAPS and AAPS with MSAA is that in the images generated by the latter, the caustics look dimmer. This was also confirmed by measuring total light levels in images for both versions. This indicated a problem, as adding rasterization samples alone should not affect the expected value of the individual samples, and the total should still sum up to around the same value. An example of an image displaying this lower light level can be seen in Fig. 35.

As for convergence, one could argue that AAPS converges better with MSAA because the error rate is lower at its lowest point. However, it also stops converging at around the same value for the target area parameter. What also initially surprised us, was that the percentage of non-contributing photon splats was slightly higher with MSAA enabled for every single parameter setting that we recorded.

The problem with standard MSAA

Further investigation brought more clarity to what caused MSAA to produce dimmer images and worse FLIP values. The same reason for non-contributing photon count to be higher with MSAA enabled, which lead to reduced overall light levels, and what likely also made convergence not go as far as expected. The problem was caused by the elliptical shape of photon splats. The rasterizer still sees the splats as triangles, as it cannot render ellipses directly. So enabling MSAA took multiple

samples of the half-quad, instead of the ellipse. Triangle pairs are only turned into ellipses in the pixel shader. This is done by using an altered version of the barycentric coordinates of the quad, that ranges from -1 to 1. If the sum of the squares of the barycentric coordinates is larger than 1, that means the center of the pixel is outside of the ellipse.

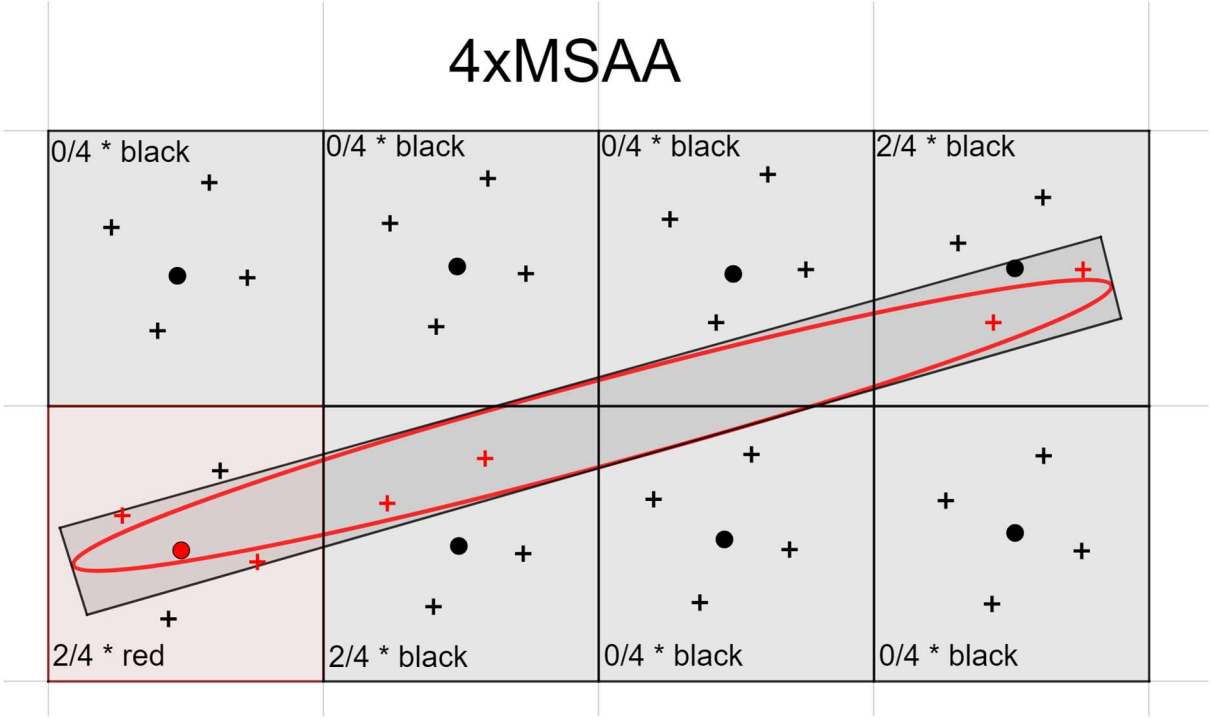


Figure 33 - How MSAA without per-sample invocations deals with quads that turn into ellipses in the pixel shader. The "+" symbols are automatically checked against the triangle. For pixels where at least one "+" was covered, a pixel shader is invoked at the center of that pixel, and scaled by the percentage of samples that were hit, as shown in the image. In reality, the quad is also split up into two triangles, only one of which can cover the center of a pixel, leading to an ever dimmer result. For simplicity, we have displayed the splat as a quad, because the same problems can still be illustrated.

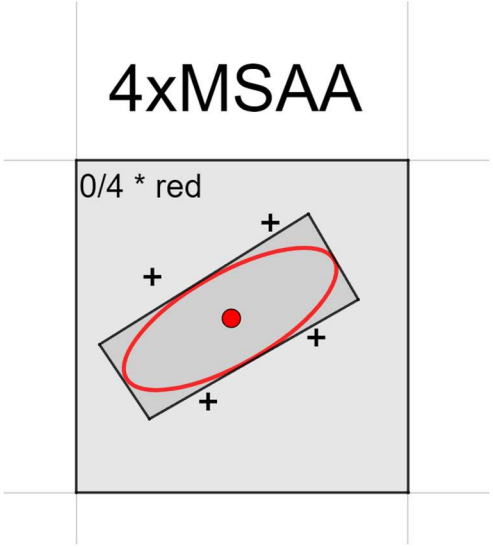


Figure 34 - An example of a pixel that would gain full brightness with no-AA, but does not cover any of the four samples and as a result, ends up not contributing to the pixel.

So the reason that photons have a slightly lower chance to contribute is that adding MSAA does not affect whether the pixel shader rejects samples. If the center of the pixel is not within the ellipse, the pixel shader will return zero light. At the same time, the eight sample points of 8xMSAA are all placed at different positions than the center of the pixel. So there is a slim chance that a splat that would have gotten accepted in the pixel shader, gets omitted because all eight MSAA sample points are outside of the triangle. The reason the overall light level is lower is similar: the pixel shader is responsible for checking whether the pixel is inside the ellipse, just like without MSAA. The only thing MSAA changes, is that the final color value of a contribution can be scaled down if fewer than all eight sample points are within the triangle.

As for convergence, we think that it is not surprising that it does not improve significantly. The ellipses are not sampled more, just the triangles they are built from. This understanding puts into perspective how much potential anti-aliasing has for AAPS. If MSAA is slightly incorrect, but still improves perceived image quality and RMSE, that makes other anti-aliasing approaches such as MSAA with invocations, supersampling anti-aliasing (SSAA), and conservative rasterization more interesting.

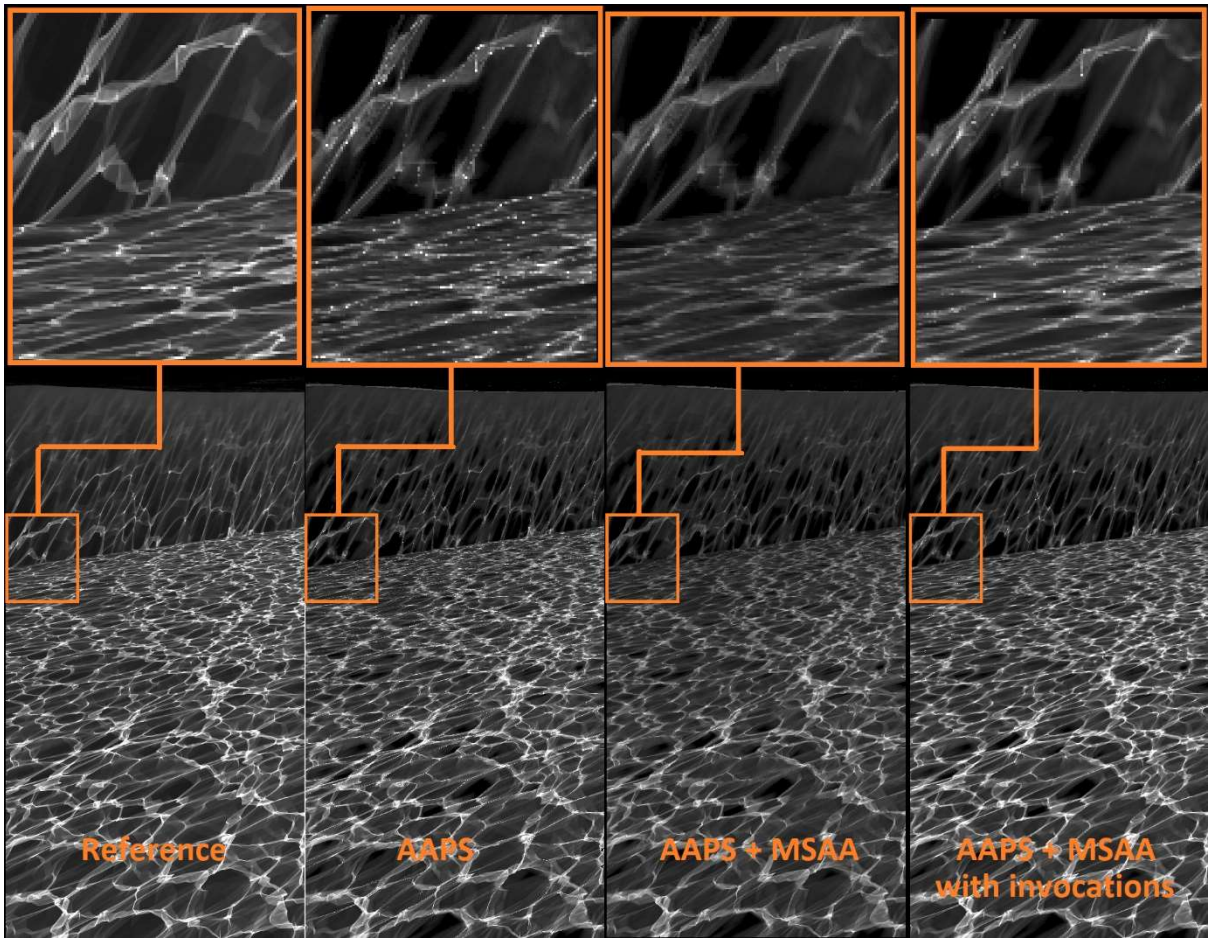


Figure 35 – Comparing AAPS with and without anti-aliasing variants. To highlight anti-aliasing's impact which is most present for high-quality renderings, these test were ran using a small value as target area, shown as the first data point on the Pool figures in further sections. This means that render times were relatively high (14ms, 20ms, 28ms for no-AA, MSAA, MSAA with invocations respectively).

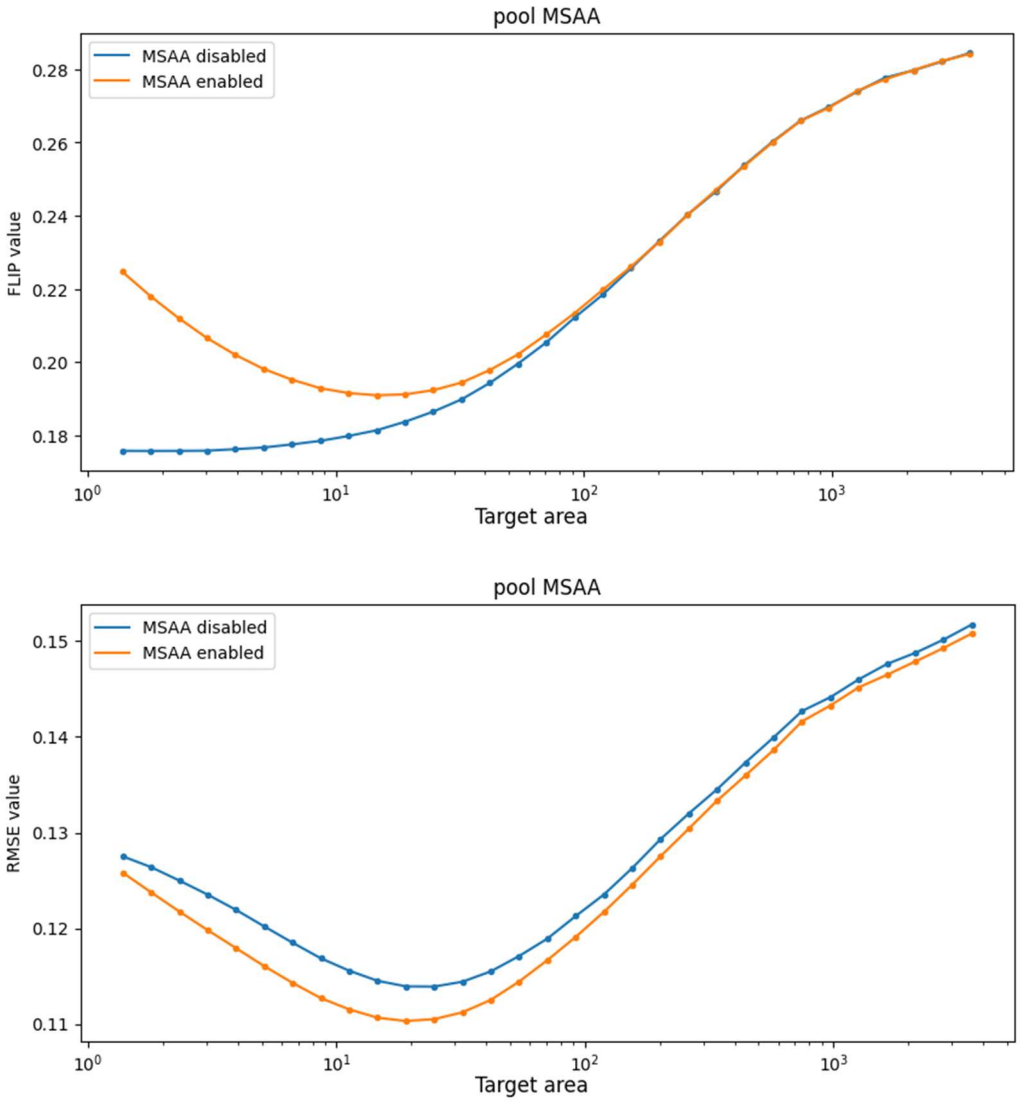


Figure 36 -The quality of images with different values set for the target area parameter. No anti-aliasing is compared to MSAA.

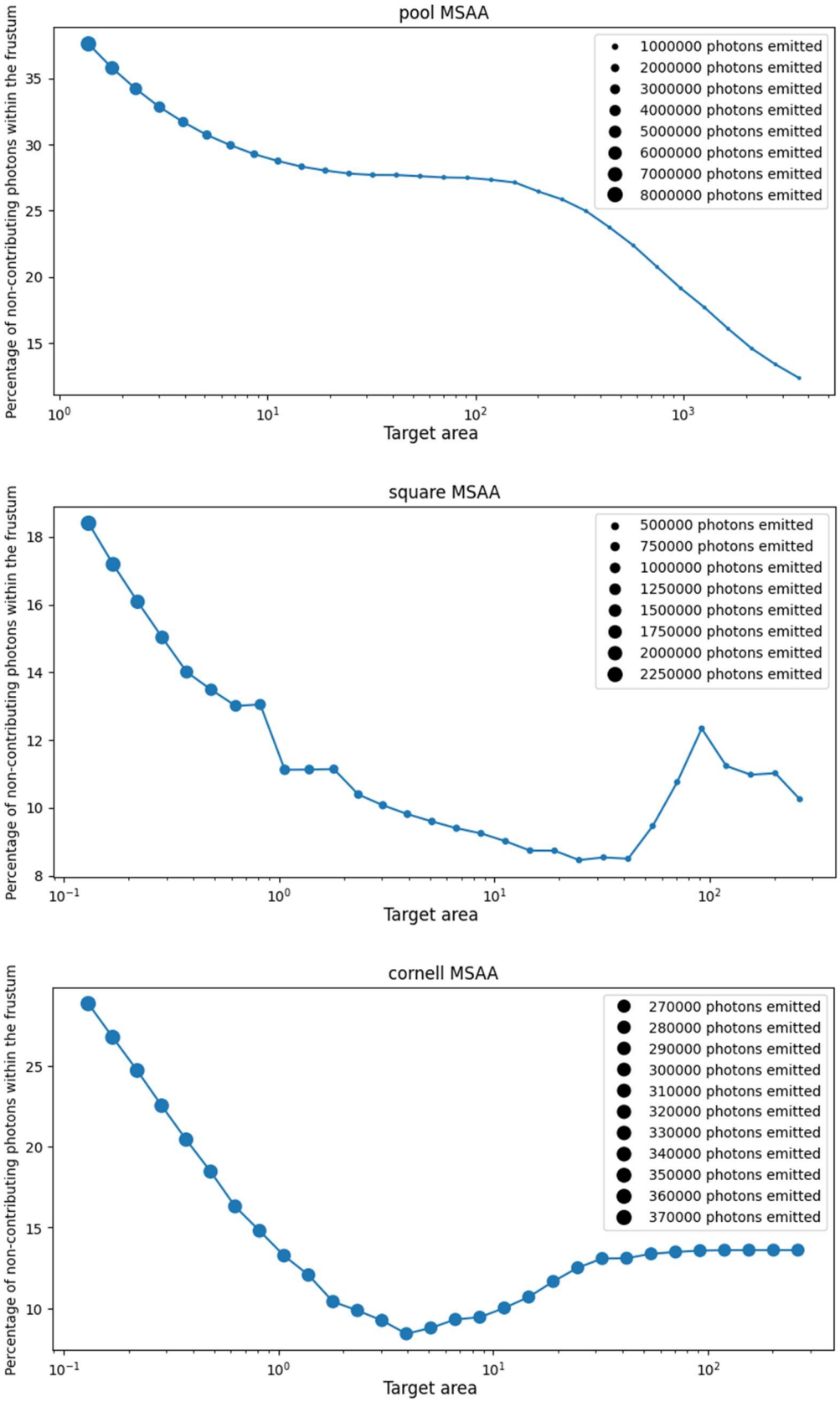


Figure 37 - Percentage of photons that do not contribute to the image. 8x MSAA enabled.

5.1.2 MSAA with per-sample invocations

The results of our experiments with MSAA led us to implement a slightly different version of MSAA. The difference between regular MSAA and MSAA with per-sample invocations is that the pixel shader gets executed for every single sample, instead of just a single time for all samples within a pixel. This results in better sampling, because of the inaccuracies that resulted from sampling an ellipse using the encapsulating triangle pair in regular MSAA.

In line with expectations, fig. 41 displays a significantly higher percentage of contributing photons. The biggest change happens when the target area is the lowest, and emitted photons are highest. Data points where non-contributing photon count was the highest without anti-aliasing saw the most significant gain. The factor of improvement is especially large in the Cornell scene. There, percentage of non-contributing photons decreased by a factor of 6.7 at the smallest target area value in comparison with NO-AA.

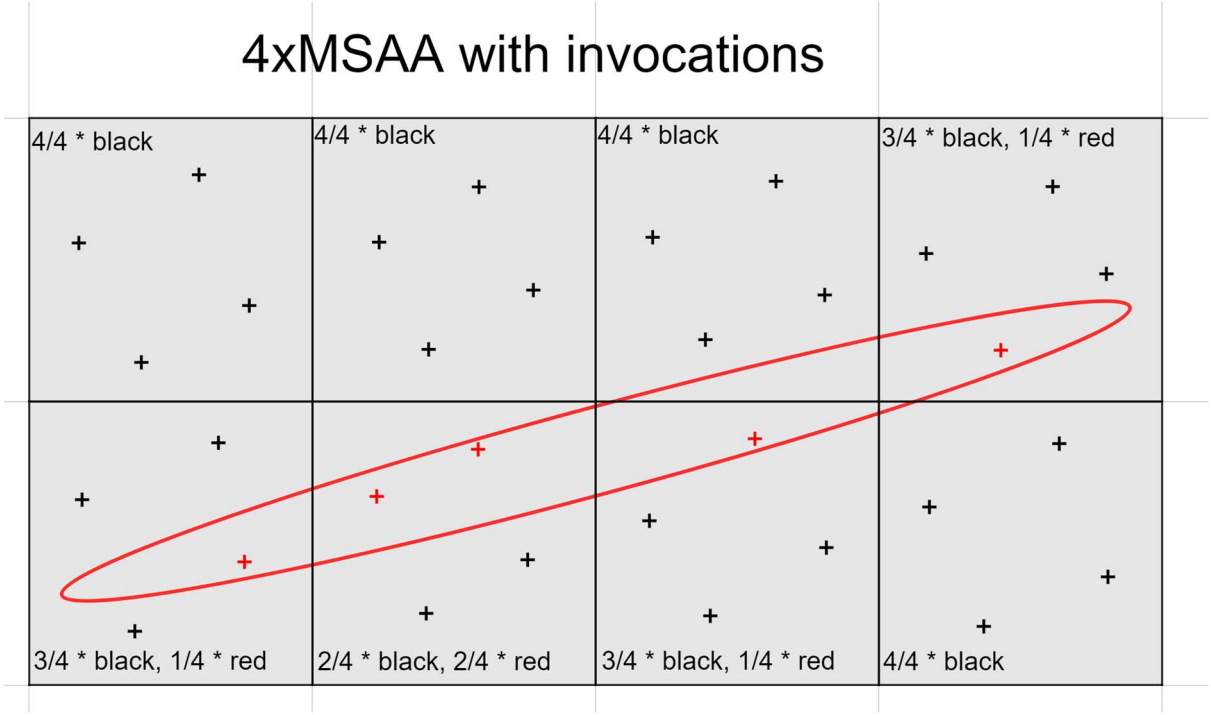


Figure 38 - How MSAA with invocations solves the problems that result from using MSAA without invocations. Ellipses are only formed in the pixel shader.

While MSAA with per-sample invocations led to a significant reduction in the number of **fireflies**, and increased correctness, a number of fireflies remained visible in the anti-aliased images. Similarly, images generated with the largest recorded target area value had small bright splats spread across the image. As the target area value gets smaller, the impact of individual fireflies on the image also gets smaller, but they still persisted, usually affecting single pixels.

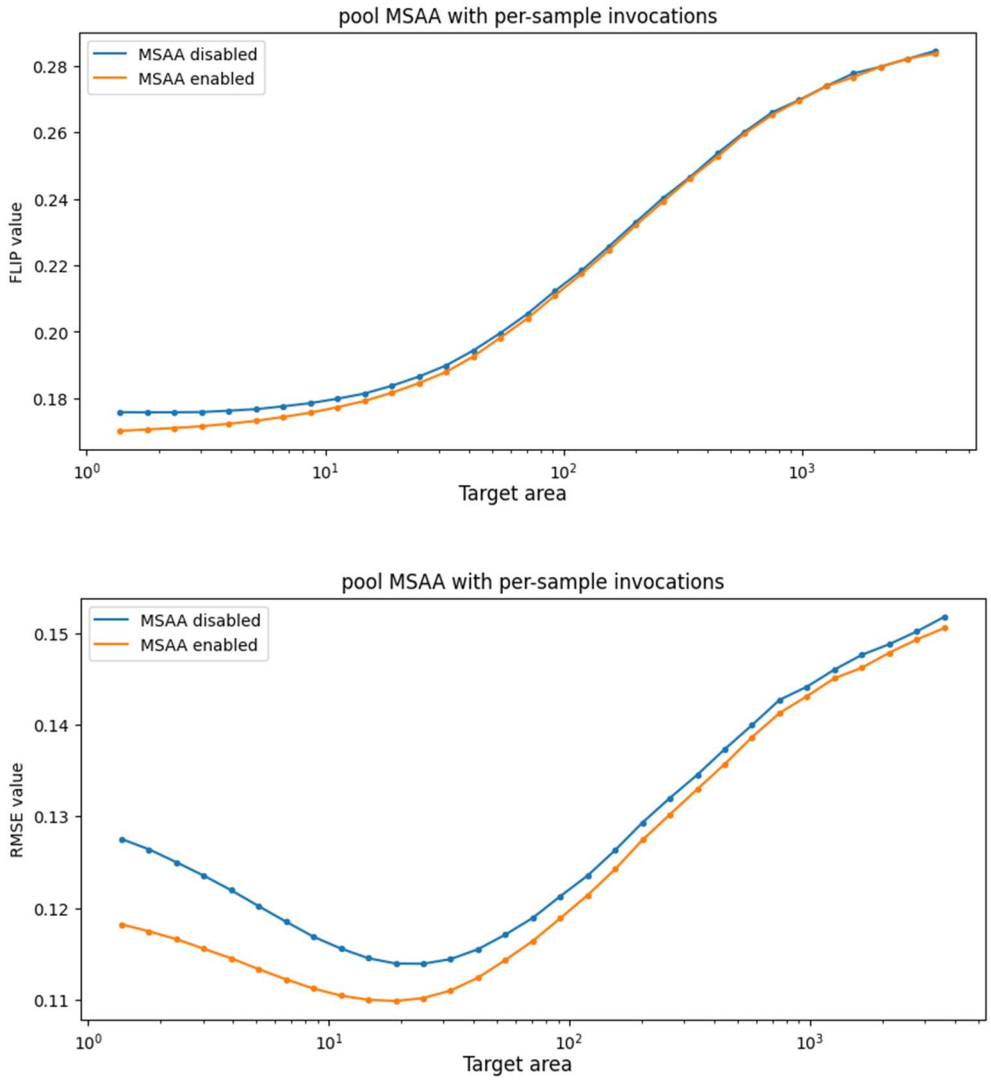


Figure 39 - The quality of the Pool scene rendering shown with different values for the target area parameter. MSAA with invocations is compared to a base version without anti-aliasing

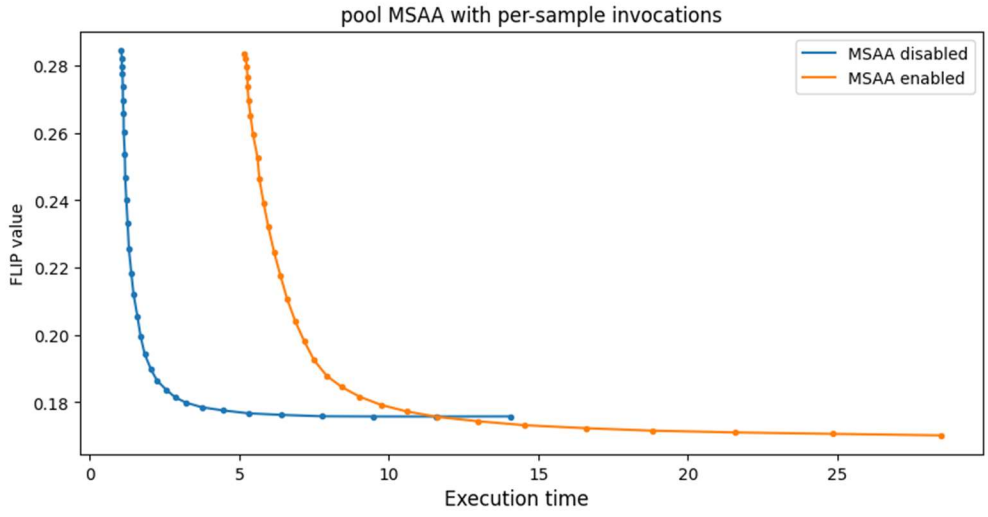


Figure 40 - The FLIP value of the rendered image, reflecting image quality, shown as a function of execution time. MSAA with invocations is compared to a base version without anti-aliasing.

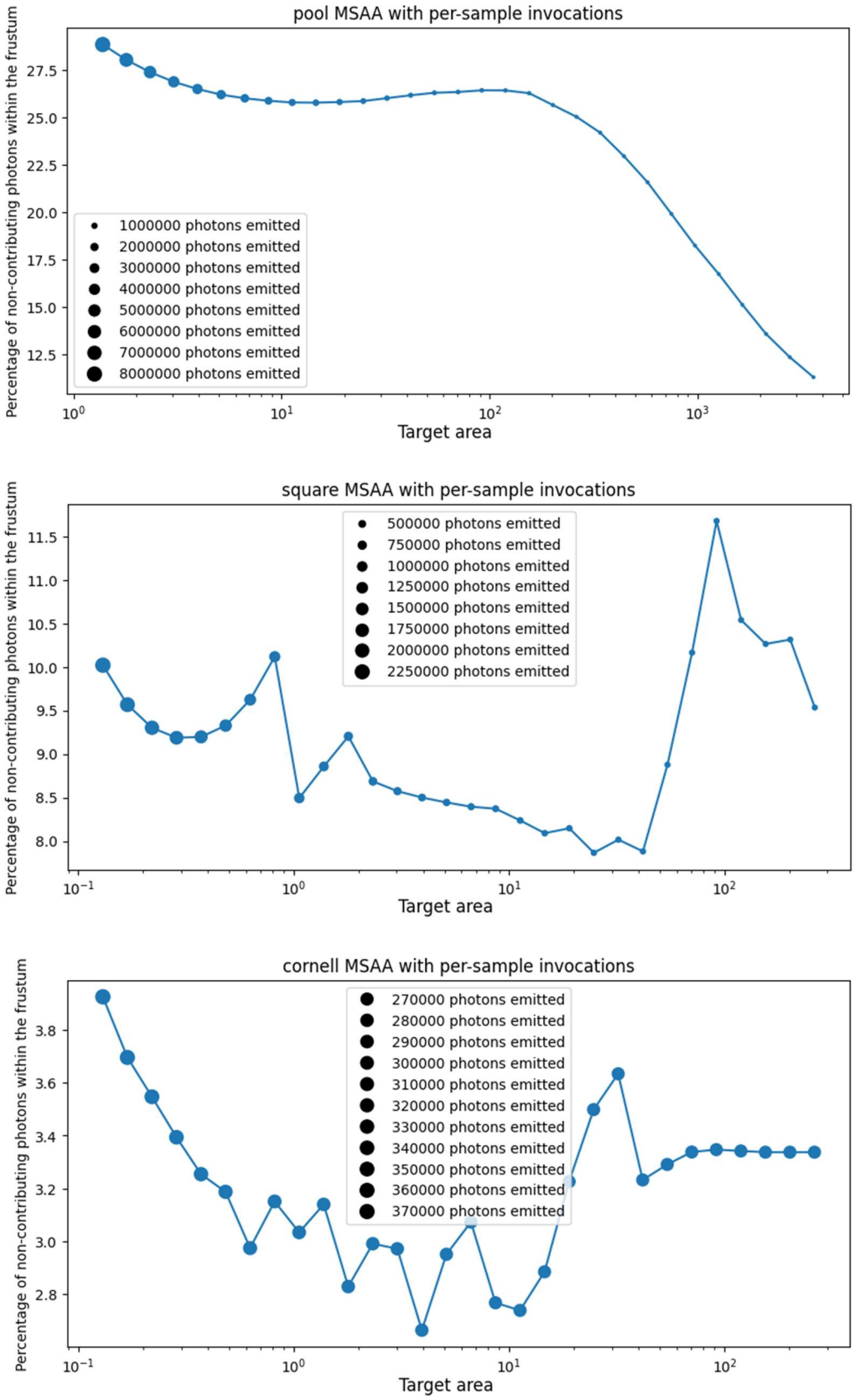


Figure 41 - Percentage of non-contributing photons when per-sample invocations are enabled. A range of different scenes and target area were used to obtain a wider set of results.

5.1.3 Anti-Aliasing specifically for small splats

While MSAA with per-sample invocations does improve image quality, we showed in the previous section that this happens at a high cost. It is possible to split photons into two groups with only minor computational overhead. One group contains photons that will be rendered **without** multisampling, while the other group contains photons that will be rendered **with** multisampling. The overhead this causes consists of an extra resolve step that is needed to resolve from a multisampled render target to a regular one.

Photons whose size in screen space falls below a certain threshold are assigned to the list of photons that get rendered with anti-aliasing. The rest are assigned to the list of photons that gets rendered without anti-aliasing. We experimented with a range of values to find a good value for this threshold, before finalizing its value for later experiments. Fig. 42 shows the results. Note that setting a different value for the threshold would lead to a different image improvement / added costs ratio.

After choosing a suitable value for the MSAA threshold, we did a second experiment which used different values of the target area parameter to obtain a comparison of the base technique with the extended technique at different execution times. This comparison is visible in Fig. 43, and shows that for a budget of execution time that exceeds 3ms, in this scene, the extension improves the FLIP result of the image.

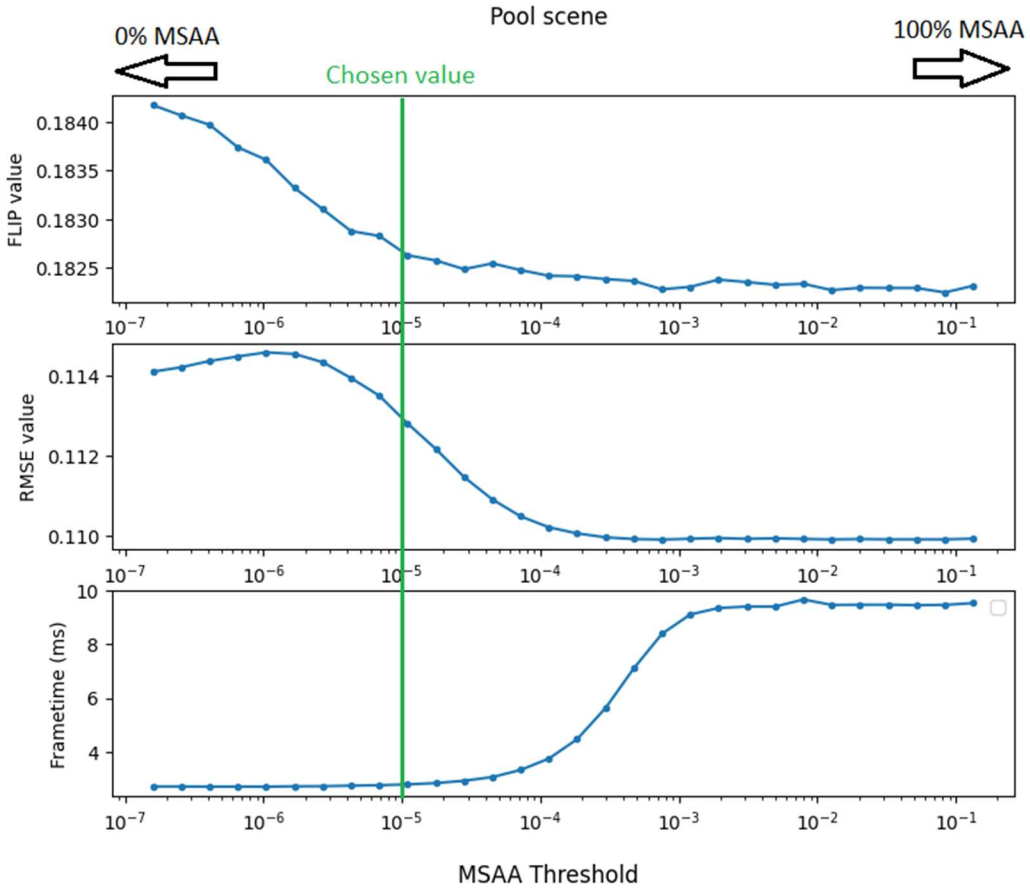


Figure 42 - The effect of the MSAA threshold on FLIP, RMSE, and frame execution time.

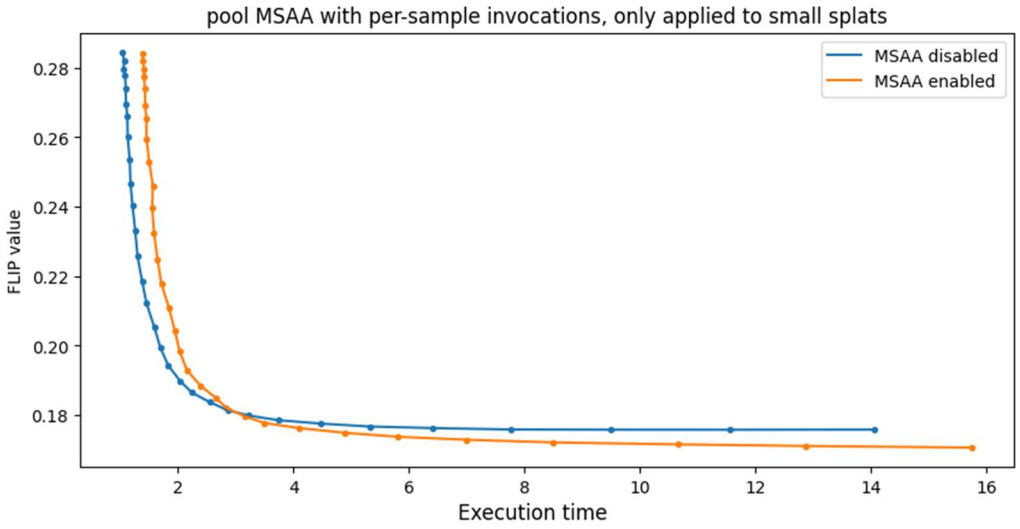


Figure 43 - FLIP value plotted against execution time (in ms). AAPS without anti-aliasing is shown as a baseline, and compared to a custom implementation which applies MSAA with per-sample invocations to small splats only.

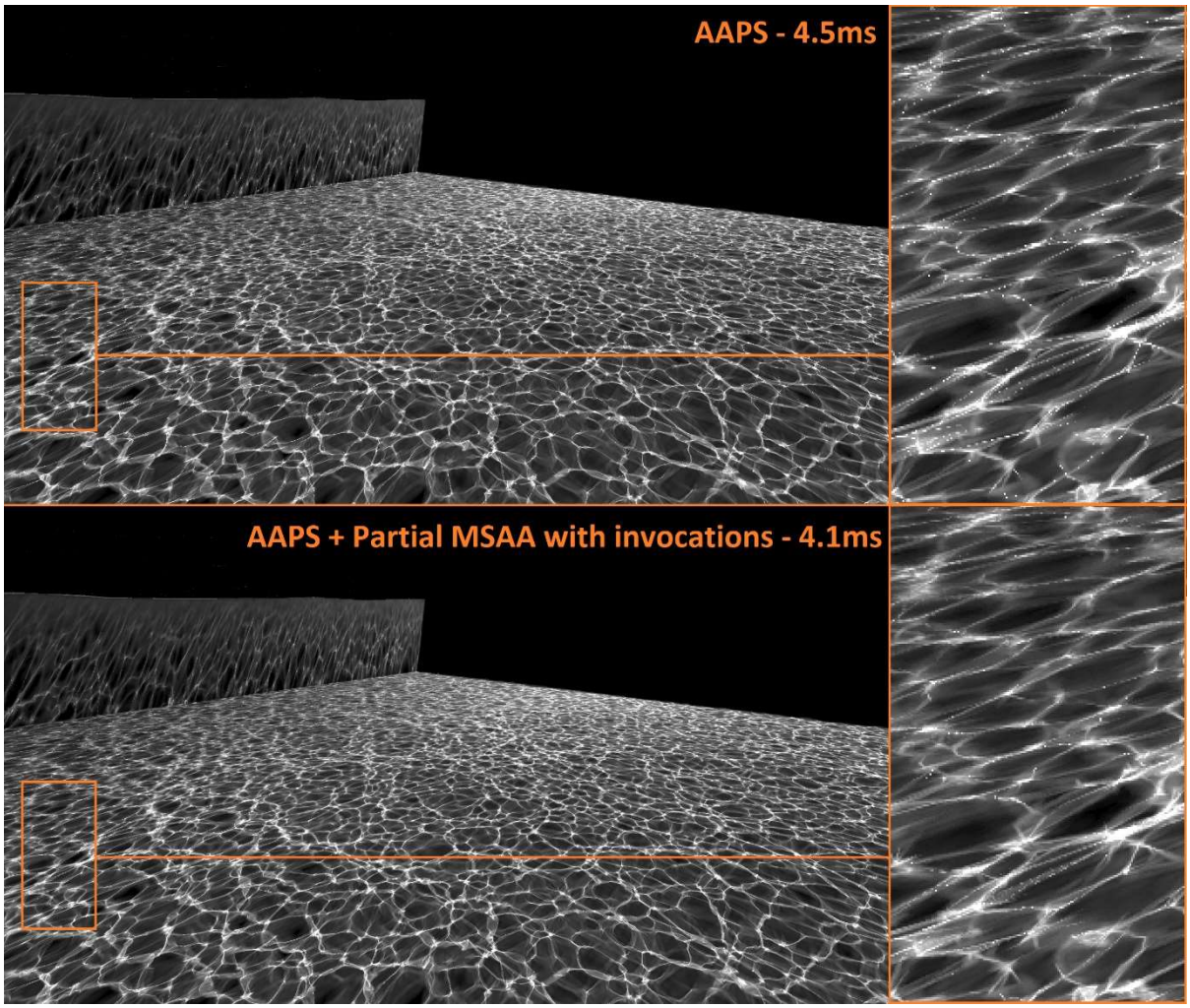


Figure 44 - A comparison between AAPS and AAPS with the extension. The figure shows a use case for our extension of AAPS. Note in particular the improvement in thin triangles and fireflies, which benefit from anti-aliasing.

5.1.4 Conservative Rasterization

While regular rasterization only flags pixels for which the center is covered by a triangle, conservative rasterization flags any pixel that is in any part covered by the triangle for execution by a pixel shader. In our context, this would enable the developer to compute the portion of the splat that overlaps with the pixel and return a corresponding portion of the photon's energy as the splat's contribution to that pixel.

The downside of using conservative rasterization is that this would require the computation of the overlap between an elliptical splat and a pixel for every pixel shader execution. To remedy the adverse effects on performance while benefitting from the improved convergence, it may be advisable to only apply this extra detailed rasterization approach for photons where it makes the most difference. Using conservative rasterization for small photons while using regular rasterization for large and medium sized photons would reduce its performance impact by an order of magnitude, similar to our partial-multisampling approach, while still benefiting from the most important improvements.

5.1.5 Single-pixel photons

A simpler approach than using conservative rasterization for small triangles would be to distribute all of a triangle's energy over a single pixel. Just like with conservative rasterization, it is not known what the performance and convergence behavior would exactly be like using this method. However, just like using conservative rasterization, a limiting factor would be removed that currently makes converging impossible with rasterization based photon mapping.

5.2 Indirect caustics

As explained in the previous chapter, photon rasterization approaches are not capable of rendering light paths where caustics are exclusively visible indirectly. Existing work has not demonstrated the ability to render this type of light in a real-time setting, yet. For offline rendering, there are multiple known ways these paths can be rendered. Most of these methods have never been tried using modern hardware, with real-time rendering in mind.

Indirect caustics can manifest themselves in many different types of scenes. Whether a method is sufficient or not depends on many criteria. AAPS would be extendible for flat mirrors, for example, by rasterizing photon splats twice. Once onto the screen, and once onto the mirror. Similarly, if all caustics are concentrated onto a single surface, such as the floor in the Square scene, they could also be rasterized onto the surface instead of onto the screen, to be rendered through a texture lookup.

Another potential way to extend AAPS would be to store lighting information on the triangle's surface, instead of on the caustics buffer. This would be similar to baking but applied in a real-time setting. Most of these methods have significant problems, though, which is why we look into using an acceleration structure next.

The option to use an acceleration structure to connect photons to pixels as was originally done with photon mapping still remains. As we described earlier, AAPS benefits from three factors of improvement over basic photon mapping. The task buffer and the anisotropic footprints based on photon differentials, with minor adjustments, could both be applied to regular photon mapping. Just the factor of improvement resulting from rasterizing instead of using an acceleration structure would not be applied.

In practice, the rasterization pipeline is effectively used by AAPS as a fast method to connect photons with the pixels that it might affect. Whether they are affected is verified in the pixel shader,

where a black color is returned if the interpolated point falls outside of the ellipse, or if the photon's distance to the camera is a mismatch with the recorded depth value for the corresponding pixel.

From a real-time perspective, a method to connect photons with potential pixels needs to fulfill three requirements:

- It must get no (or very few) false negatives. We define false negatives as a failure to connect a photon to a pixel, when they should be connected. False negatives affect the quality of the image, as light gets lost whenever photon-pixel combinations are incorrectly discarded.
- It must be as fast as possible. Rasterization's performance is likely unbeatable by a large margin. However, considering the photon count is typically not that large, it may be possible to get this part under control.
- It must not produce an exorbitant number of false positives. We define false positives as an unnecessary connection between a photon and a pixel, that will later be discarded and costs some extra computation time.
  False positives already exist in AAPS. An ellipse contained by a quad covers a fraction of $\frac{\pi}{4}$ of the quad's area, which is about 79%. As a result, at least 21% of the combinations the rasterizer finds are false positives. Apart from that, any splats that are behind other geometry are too deep in the scene and should not be added to the image. These are also false positives. The last type of false positives occurs in parts of splats that extend beyond the geometry they were splatted on, when the adjacent geometry (if any) has a different depth as seen from the camera.
  Most of our scenes do not have many false positives, but scenes where they occur are important to keep in mind. Our experiments with AAPS have already demonstrated that verifying all positives, including false positives, is very manageable. This is done in the pixel shader of a rasterization approach. So as long as the number of false positives stays within a reasonable range, we conclude there should be no problems determining the real positives quickly.

In a scene where only a small portion of the screen is covered by ray-transforming objects such as mirrors and dielectrics, it would be easier to fulfill these three requirements. Rasterization could be used to render the direct caustics, and the secondary method for indirect caustics would only need to be applied for the leftover points in need of shading.

To be applicable for the general case would be much more challenging. For example, looking at underwater caustics from above the water would be one of the most difficult scenes to meet these three requirements for. This is because no rasterization can be applied, meaning all photon-pixel combinations need to be found using the acceleration structure. In addition, the number of emitted photons will be larger than ever. This is because the entire screen would get covered in caustics which would manifest itself as a large amount of reinforcement received by the task buffer.

One thing that needs to be improved to fully benefit from caustics that fall outside the frustum is the projected area feedback to the task buffer. Assuming rasterization is used, calculating the projected area is very simple and can be done per photon right before it is recorded. Otherwise, it may be more difficult to find a suitable approach. An experiment with directly measuring the projected area of photons by incrementing a counter in the pixel shader pointed out major problems. These problems were caused by the interlocked approach used. It is possible that removing the lock and allowing a race condition would not cause significant reduction in task buffer quality. In any case, the rasterization based projected area calculation can still be used for all photons, supplementing it with

measurements used only for indirect observations of caustics. Further research is required to find out what is a good way of measuring target area for indirect caustics.

5.2.1 Replacing rasterization with ray tracing

Rasterization can be replaced with ray tracing in AAPS. Just like ray tracing enables you to render light that is indirectly visible through glass by refracting camera rays and intersecting triangles, the same principle can be applied to photon splats. They are stored as triangles, after all. All that needs to be done is building a DXR acceleration structure containing these splat triangles and traversing it. Instead of cutting off ray tracing at the first intersection though, intersections would need to be evaluated and additively blended to the final color, with a depth range that is based on the depth of the scene. In a DXR implementation, this would be done by using an any hit shader and calling *IgnoreHit()* on every single execution. This approach lends itself well to new graphics hardware and DXR, because of ray tracing and acceleration structure support.

This approach would fit two of the three requirements easily if the start and end of the ray is chosen to cover a correct depth range. This depth range could be chosen in exactly the same way as with rasterization. For the first requirement, false negatives would be zero, just like with a rasterization approach. For the third requirement, false positives would be 21%, the part of the triangles that is outside of the ellipse. Because the correct depth range, and nothing more, is covered by the ray's origin and length, the only false positives that are found are the points that fall inside the splat's quad, but outside the splat's ellipse.

To better understand the cost of building an acceleration structure with DXR, we performed brief experiments. They pointed out that the naïve approach of building a brand new acceleration structure containing all photons every frame is not ideal, taking around 8ms, more than 67% of the total frame time, to build 250,000 photons. Additional research is required on how to accelerate this. Refitting most of acceleration structure while rebuilding small parts at a time or grouping photons by light atlas ID are both interesting areas for future work.

Our final experiment involved rendering these indirectly visible caustics. This was done by splitting it up in two parts, using the earlier described photon acceleration structure which is built every frame. For the first part, the caustics that were directly visible were rendered as they normally are with AAPS, namely by using rasterization. Secondly, the caustics that were not directly visible but visible via reflections or refractions, were rendered by ray tracing against the photon acceleration structure. The implementation of this used an any hit shader akin to the pixel shader in the caustics rasterization pipeline. A few example renderings that use this technique and their reference image and AAPS counterparts can be seen in Fig. 45-51.

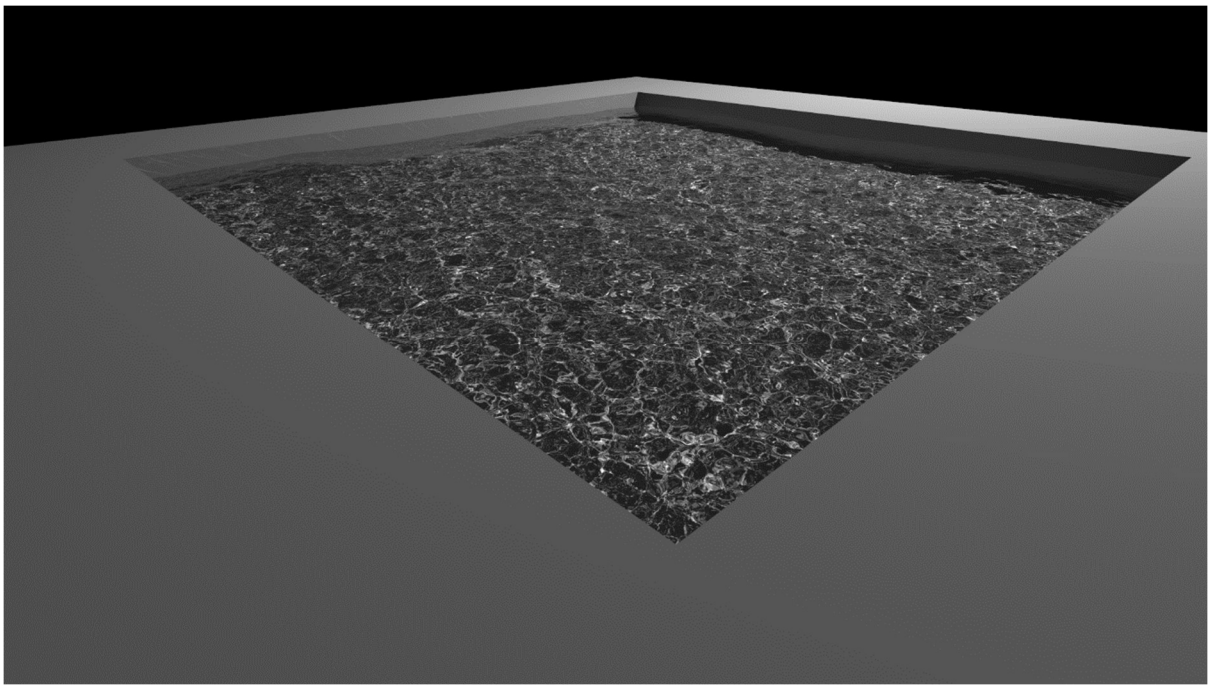


Figure 45 - The reference image of the pool scene rendered from a different viewpoint that showcases caustics that are observed via an indirect path. 31000+ samples were used to generate this image.

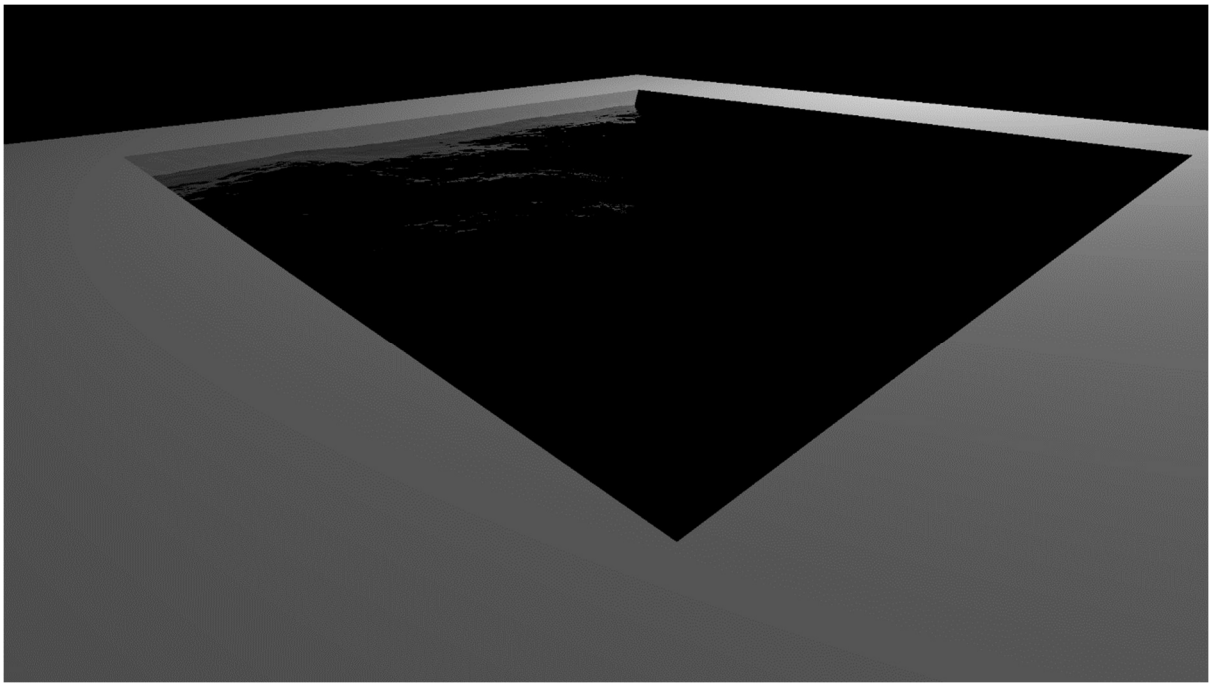


Figure 46 - The Pool scene rendered from a different viewpoint with standard AAPS. As most of the caustics are obstructed from the camera by the objects, virtually none of them are visible.

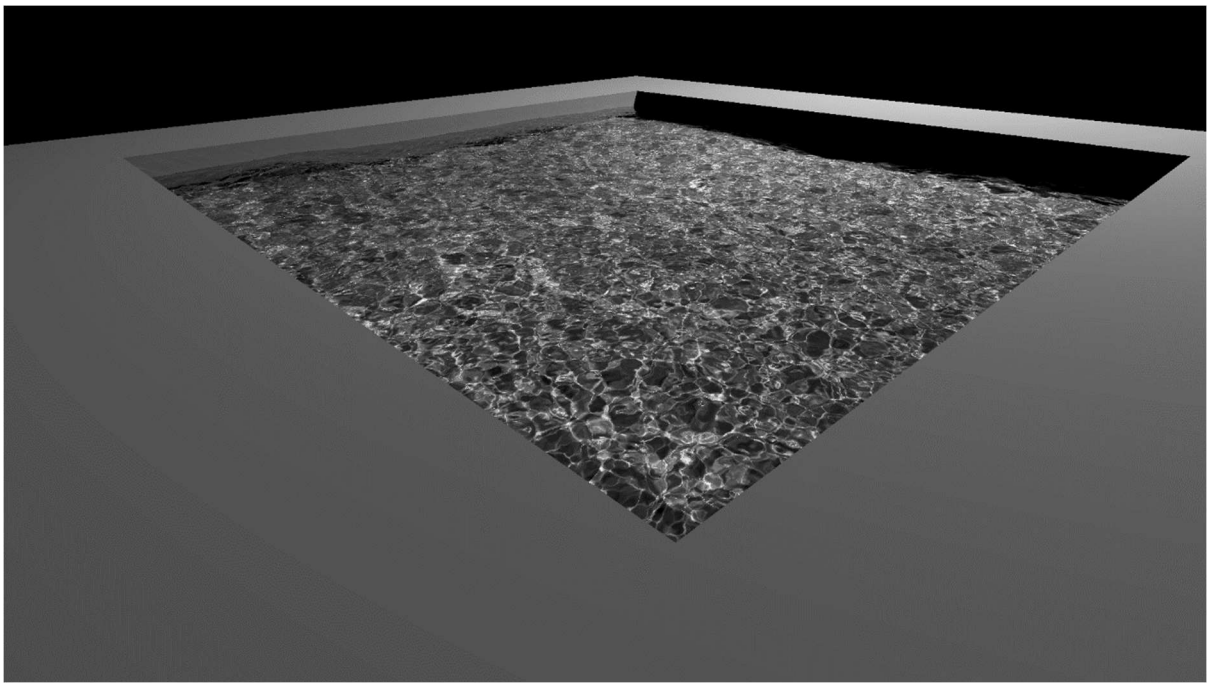


Figure 47 - The Pool scene rendered from a different viewpoint with AAPS + our indirect caustics extension. Render time was 27ms.

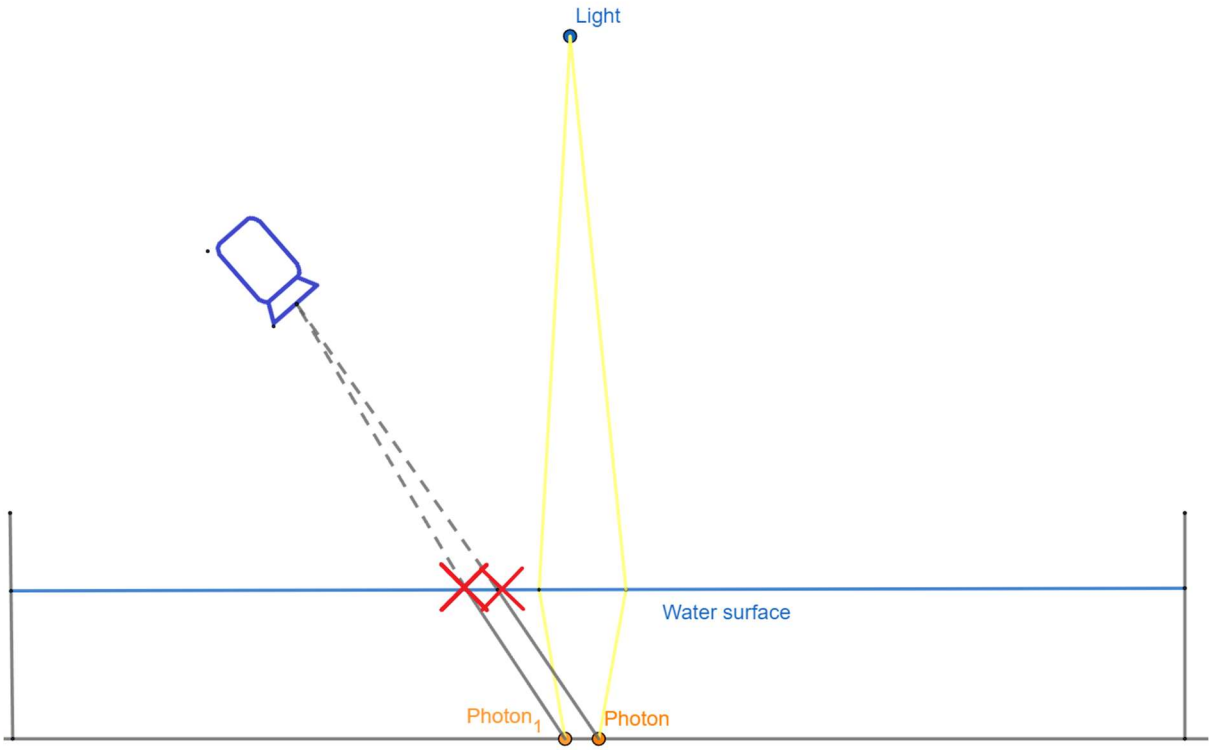


Figure 48 - How photons are unable to get rasterized to the camera, because of the water that obstructs them. Rasterizing them would lead to incorrect results, as the transformation caused by the water surface would not be recorded.

Figure 49 - Reference image showing how the indirect caustics from a specific camera angle are supposed to look like. 250 samples per pixel were taken.

Figure 50 - Base AAPS rendering the Square scene from an angle that exposes a flaw: A lack of indirect caustics if they are obstructed by objects. Render time was 1.5ms. Note that an existing AAPS extension created by the original authors could still render parts of the indirect caustics. They are omitted as our implementation of AAPS did not support this.

Figure 51 - AAPS with our indirect caustics extension, rendering the Square scene from a different viewpoint which exposes the new capabilities. Render time was 15ms.

Additional research is required to judge the exact effectiveness of ray tracing against a DXR acceleration structure containing photon splats on quality and performance. In addition, research on how to accelerate the performance of ray tracing against photons is also prudent. Examples of areas of improvement include culling and ray ordering.

It may be advantageous to reduce photon counts and ray tracing samples, as this would likely allow both the building of the photon acceleration structure and the tracing against the photon acceleration structure to be sped up by a large factor.

5.2.2 Rendering onto a texture

Instead of rasterizing the photons directly to the screen, it would also be possible to rasterize them to the object's texture. This would then allow a regular ray tracer to look up caustics values in the textures of triangles, leading to visibility of caustics, even when observed indirectly. Aside from suffering from issues with the task buffer when photons fall outside the frustum, this would come with limitations:

- A mismatch between texture resolution and screen resolution could lead to increased cost or reduced quality. This could be alleviated using LODs or mipmaps.
- The portion of the splat that falls outside of the triangle would be discarded. This might be particularly problematic with small triangles, as a large part of light contributions could get discarded.

5.3 Future improvements

5.3.1 Z-ordering

The ray tracing performance in AAPS is already great due to high coherence. However, it might be possible to improve it even further by changing the order within a pixel of the task buffer from a 2-dimensional loop to a Z-order. This might positively affect rays beyond the primary rays, too.

5.3.2 Organizing Secondary Rays

AAPS does not come with a system to organize secondary rays. Two naïve ways to do so would be to use recursion or a simple array with a counter. However, to further enhance ray tracing performance it may be worth it to attempt to organize the secondary rays in a manner that improves coherence.

One way to achieve a good ordering of secondary rays this would be to create an additional quadtree, for secondary rays. Recall that the quadtree is responsible for pointing threads to the correct rays. By applying the same idea to secondary rays, ray tracing performance might gain a significant boost. As parallel to the ray density texture, this quadtree would be based on a simple ray count, which is recorded at the same resolution.

One may also consider putting reflections and refractions on separate quadtrees, ensuring that executions of one do not start until the other quadtree is depleted. Doing this would even further improve coherence for rays beyond the first ray.

6. Conclusion

We have provided guidance and necessary details on how to implement AAPS effectively. In addition, aspects of AAPS that were previously unknown, have been investigated and reported.

6.1 Performance of AAPS

Performance-wise, we found AAPS to be very suitable for real-time caustics rendering on modern AMD and NVIDIA architectures. With well-tuned parameters, the performance is predictable and stays in a range that depends mostly on the chosen target area. The overhead of AAPS is minimal, and most time is spent on ray tracing, particularly for low target area settings.

6.2 Applicability of AAPS on single-event underwater caustics

We found that AAPS is suitable for rendering caustics caused by a body of water, as long as the camera is not pointing at surfaces behind the water. Looking at underwater caustics from above the water, or looking at caustics above the water from underwater, are both not supported by AAPS.

6.3 Convergence limit

We found the effect of increasing the sample count in AAPS as guided by the task buffer to have diminishing returns. After increasing the budget and samples beyond a certain point, it becomes impossible to achieve significant improvements within a real-time setting. We have investigated and shown one factor that contributes to this, which is that small ellipses do not get rasterized well.

6.4 Improving convergence with anti-aliasing

We have implemented several forms of anti-aliasing and demonstrated its improvement on caustics quality with AAPS. We showed that using anti-aliasing with AAPS allows it to converge further with the same number of photons. Using FLIP value as a metric, we found that basic MSAA with invocations becomes worth using only with high photon counts, at AAPS execution times of over 12ms on the RX 6900 XT.

We also presented a method that benefits from most of the improved convergence, while keeping the increase in rendering cost to a minimum. This is achieved by performing MSAA with invocations exclusively on the smallest photons. We found that using this method becomes worth it at AAPS execution times of over 3ms on the RX 6900 XT. We also note that our test scenes were low in complexity which likely led to a higher than normal percentage of execution time going to rasterization. This is important because it means that anti-aliasing would be even more cost effective in more complex test cases where taking more samples has a higher associated ray tracing cost.

6.5 Enabling indirect caustics in real-time

The last experiments that were performed showed that it is possible to render caustics that are obstructed or outside the frustum in addition to directly visible caustics, in real-time. This was done with a technique which alters AAPS by replacing rasterization of photons with ray tracing. While this technique is right on the edge of real-time, it is not currently suitable for use in real-time applications at this time as it leaves virtually no time for other rendering or techniques that require GPU execution time. The execution time of this technique is roughly tenfold compared to regular AAPS.

6.6 Future work

6.6.1 Temporal analysis

While we performed thorough experiments collecting many data points about AAPS, we did not collect data about temporal stability. Temporal stability ranged from poor to great, mostly

depending on parameter values. Another potentially interesting aspect we did not collect data on is ghosting of undersampled areas. We did not encounter this with our parameter settings.

6.6.2 Larger scenes

Most of the test cases we used had small scales. However, some games have much different requirements. Modern open world games can have multiple square kilometers, or even more, of playable area. We have not verified the applicability of AAPS in such scenarios, though the task buffer approach is unlikely to work well for the sun if its resolution is too small to pinpoint objects that should cast caustics.

6.6.3 Firefly reduction

While fireflies did not affect the rendered images much, they were observable in most cases. Further research is required to find out the cause of them, and the best way to deal with them. This might be as simple as setting a minimum size for splats or suppressing anything brighter than a certain value.

7. Acknowledgements

I would like to thank Matthäus Chajdas (AMD) for his invaluable guidance throughout all steps of creating this thesis, and providing me the opportunity to meet many great minds.

Furthermore, I would like to thank Dominik Baumeister (AMD) and Michael Kern (AMD) for their guidance with learning DirectX12, and the process as a whole.

I would also like to thank Jacco Bikker for introducing me to ray tracing in such an enthoustiastic way, and supervising my thesis after teaching me the basics.

Thank you to Marcus Rogowsky (AMD) and Benedikt Kessler (AMD), in particular for the insight I've gained into the graphics and hardware industry because of them.

Finally, I would like to thank Max Oberberger (AMD) for his excellent support in troubleshooting.

ANDERSSON, P., NILSSON, J., AKENINE-MÖLLER, T., OSKARSSON, M., ÅSTRÖM, K., AND FAIRCHILD, M.D. 2020. FLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2, 1–23.

ENCHEV, H. 2020. Cheap Water Caustics in UE4. https://medium.com/hri-tech/cheap-water-caustics-in-ue4-ee1d3ac0cae1.

ESTEVEZ, A.C. AND KULLA, C. 2020. Practical Caustics Rendering with Adaptive Photon Guiding. *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks*, ACM, 1–2.

GEFFROY, J., GNEITING, A., AND WANG, Y. 2020. Rendering the Hellscape of Doom Eternal. https://advances.realtimerendering.com/s2020/index.html.

GEORGIEV, I. 2012. Implementing Vertex Connection and Merging. .

GEORGIEV, I., KŘIVÁNEK, J., DAVIDOVIČ, T., AND SLUSALLEK, P. 2012. Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics 31*, 6, 1–10.

HACHISUKA, T., OGAKI, S., AND JENSEN, H.W. 2008. Progressive photon mapping. *ACM Transactions on Graphics 27*, 5, 1–8.

IGEHY, H. 1999. Tracing ray differentials. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, ACM Press, 179–186.

JENSEN, H.W. 1996. Global Illumination using Photon Maps. In: X. Pueyo and P. Schröder, eds., *Rendering Techniques '96*. Springer Vienna, Vienna, 21–30.

JENSEN, H.W. 2001. *Realistic image synthesis using photon mapping*. A K Peters, Natick (Mass.).

KAJIYA, J.T. 1986. The rendering equation. *ACM SIGGRAPH Computer Graphics 20*, 4, 143–150.

KAPLANYAN, A.S. AND DACHSBACHER, C. 2013. Adaptive progressive photon mapping. *ACM Transactions on Graphics 32*, 2, 1–13.

KIM, H. 2019. Caustics Using Screen-Space Photon Mapping. In: E. Haines and T. Akenine-Möller, eds., *Ray Tracing Gems*. Apress, Berkeley, CA, 543–555.

LAFORTUNE, E.P. AND WILLEMS, Y.D. 1993. BI-DIRECTIONAL PATH TRACING. .

MA, V.C.H. AND MCCOOL, M.D. 2002. Low Latency Photon Mapping Using Block Hashing. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, 89–99.

MARA, M., LUEBKE, D., AND MCGUIRE, M. 2013. Toward practical real-time photon mapping: efficient GPU density estimation. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '13*, ACM Press, 71.

MCGUIRE, M. AND LUEBKE, D. 2009. Hardware-accelerated global illumination by image space photon mapping. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*, ACM Press, 77.

MÜLLER, T., EVANS, A., SCHIED, C., AND KELLER, A. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics 41*, 4, 1–15.

NGUYEN, H., ED. 2008. *GPU Gems 3*. Addison-Wesley, Upper Saddle River, NJ Munich.

NICHOLS, C. 2019. What are caustics and how to render them the right way. https://www.chaos.com/blog/what-are-caustics-and-how-to-render-them-the-right-way.

PHARR, M., JAKOB, W., AND HUMPHREYS, G. 2017. *Physically based rendering: from theory to implementation*. Elsevier, Morgan Kaufmann Publishers, Amsterdam Boston Heidelberg London New York Oxford Paris San Diego San Francisco Singapore Sydney Tokyo.

SCHJØTH, L., FRISVAD, J.R., ERLEBEN, K., AND SPORRING, J. 2007. Photon differentials. *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE '07*, ACM Press, 179.

SHAH, M.A., KONTTINEN, J., AND PATTANAIK, S. 2007. Caustics Mapping: An Image-Space Technique for Real-Time Caustics. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 272–280.

SMITS, B.E., ARVO, J.R., AND SALESIN, D.H. 1992. An importance-driven radiosity algorithm. *ACM SIGGRAPH Computer Graphics 26*, 2, 273–282.

STAM, J. 1996. Random caustics: natural textures and wave theory revisited. *ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96 on - SIGGRAPH '96*, ACM Press, 150.

STÜRZLINGER, W. AND BASTOS, R. 1997. Interactive Rendering of Globally Illuminated Glossy Scenes. In: J. Dorsey and P. Slusallek, eds., *Rendering Techniques '97*. Springer Vienna, Vienna, 93–102.

VEACH, E. AND GUIBAS, L. 1995. Bidirectional Estimators for Light Transport. In: G. Sakas, S. Müller and P. Shirley, eds., *Photorealistic Rendering Techniques*. Springer Berlin Heidelberg, Berlin, Heidelberg, 145–167.

VEACH, E. AND GUIBAS, L.J. 1997. Metropolis light transport. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*, ACM Press, 65–76.

WALD, I. 2022. GPU-friendly, Parallel, and (Almost-)In-Place Construction of Left-Balanced k-d Trees. http://arxiv.org/abs/2211.00120.

WALD, I., GÜNTHER, J., AND SLUSALLEK, P. 2004. Balancing Considered Harmful - Faster Photon Mapping using the Voxel Volume Heuristic -. *Computer Graphics Forum 23*, 3, 595–603.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.

WYMAN, C. 2005. An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics 24*, 3, 1050–1053.

WYMAN, C. AND NICHOLS, G. 2009. Adaptive Caustic Maps Using Deferred Shading. *Computer Graphics Forum 28*, 2, 309–318.

YANG, X. AND OUYANG, Y. 2021. Real-Time Ray Traced Caustics. In: A. Marrs, P. Shirley and I. Wald, eds., *Ray Tracing Gems II*. Apress, Berkeley, CA, 469–497.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5, 1–11.