



LUND UNIVERSITY

Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures

Moreau, Pierre

2022

Document Version:

Publisher's PDF, also known as Version of record

[Link to publication](#)

Citation for published version (APA):

Moreau, P. (2022). *Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures*. [Doctoral Thesis (compilation), Faculty of Engineering, LTH]. Department of Computer Science, Lund University.

Total number of authors:

1

Creative Commons License:

Unspecified

General rights

Unless other specific re-use rights are stated the following general rights apply:

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: <https://creativecommons.org/licenses/>

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures

by Pierre Moreau




LTH
FACULTY OF
ENGINEERING

Thesis for the degree of Doctor of Philosophy in Engineering
Thesis advisor: Assoc. Prof. Michael Doggett
Thesis assistant advisor: Docent Jacob Munkberg
Faculty opponent: Assoc. Prof. Veronica Sundstedt

To be presented, with the permission of the Faculty of Engineering, LTH of Lund University, for public criticism in E:1406 at the Department of Computer Science on Friday, the 14th of January 2022 at 13:00.

Organization LUND UNIVERSITY		Document name DOCTORAL DISSERTATION
Department of Computer Science Box 118 SE-221 00 Lund Sweden		Date of disputation 2022-01-14
Author(s) Pierre Moreau		Sponsoring organization Swedish Research Council, and ELLIIT Excellence Center at Linköping-Lund in In-formation Technology
Title and subtitle Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures		
Abstract The improvements in GPU hardware, including hardware-accelerated ray tracing, and the push for fully dynamic realistic-looking video games, has been driving more research in the use of ray tracing in real-time applications. The work described in this thesis covers multiple aspects such as optimisations, adapting existing offline methods to real-time constraints, and adding effects which were hard to simulate without the new hardware, all working towards a fully dynamic surface illumination rendering in real-time. Our first main area of research concerns photon-based techniques, commonly used to render caustics. As many photons can be required for a good coverage of the scene, an efficient approach for detecting which ones contribute to a pixel is essential. We improve that process by adapting and extending an existing acceleration data structure; if performance is paramount, we present an approximation which trades off some quality for a $2-3\times$ improvement in rendering time. The tracing of all the photons, and especially when long paths are needed, had become the highest cost. As most paths do not change from frame to frame, we introduce a validation procedure allowing the reuse of as many as possible, even in the presence of dynamic lights and objects. Previous algorithms for associating pixels and photons do not robustly handle specular materials, so we designed an approach leveraging ray tracing hardware to allow for caustics to be visible in mirrors or behind transparent objects. Our second research focus switches from a light-based perspective to a camera-based one, to improve the picking of light sources when shading: photon-based techniques are wonderful for caustics, but not as efficient for direct lighting estimations. When a scene has thousands of lights, only a handful can be evaluated at any given pixel due to time constraints. Current selection methods in video games are fast but at the cost of introducing bias. By adapting an acceleration data structure from offline rendering that stochastically chooses a light source based on its importance, we provide unbiased direct lighting evaluation at about 30 fps. To support dynamic scenes, we organise it in a two-level system making it possible to only update the parts containing moving lights, and in a more efficient way. We worked on top of the new ray tracing hardware to handle lighting situations that previously proved too challenging, and presented optimisations relevant for future algorithms in that space. These contributions will help in reducing some artistic constraints while designing new virtual scenes for real-time applications.		
Key words Computer graphics, Real-time rendering, Ray tracing, Caustics, Global illumination		
Classification system and/or index terms (if any) UKÄ: Natural Sciences → Computer and Information Science → Computer Science		
Supplementary bibliographical information		Language English
ISSN and key title ISSN: 1404-1219 Dissertation 67, 2022; LU-CS-DISS: 2022-01		ISBN 978-91-8039-138-2 (print) 978-91-8039-137-5 (pdf)
Recipient's notes	Number of pages 138	Price
	Security classification	

Distribution by (name and address): Department of Computer Science, Box 118, SE-221 00 Lund, Sweden
I, the undersigned, being the copyright owner of the abstract of the above-mentioned dissertation, hereby grant to all reference sources the permission to publish and disseminate the abstract of the above-mentioned dissertation.

Signature 

Date 2021-12-09

Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures

by Pierre Moreau



LUND
UNIVERSITY

LTH

FACULTY OF
ENGINEERING

A doctoral thesis at a university in Sweden takes either the form of a single, cohesive research study (monograph) or a summary of research papers (compilation thesis), which the doctoral student has written alone or together with one or several other author(s).

In the latter case the thesis consists of two parts. An introductory text puts the research work into context and summarizes the main points of the papers. Then, the research publications themselves are reproduced, together with a description of the individual contributions of the authors. The research papers may either have been already published or are manuscripts at various stages (in press, submitted, or in draft).

Cover illustration front: Found within the shadow of this teapot made of glass, the complex light patterns remain challenging to compute for real-time applications. This is one effect of light simulation covered in this thesis, with Paper v investigating how to visualise them when they are indirectly visible (e.g. via a mirror).

The teapot and scene were modelled using Blender [10], and rendered using Falcor [25].

Cover illustration back: Computing the contributions of tens of thousands of light sources, as is the case in this scene, was previously very hard to achieve in real-time without biasing the results or adding large constraints. Through Paper III and IV, we made it possible to render such scenes without bias nor specific constraints, and even for animated scenes.

The scene was rendered using Falcor [25], and credits for the different assets used are found in the Acknowledgements section.

Funding information: The thesis work was financially supported by the Swedish Research Council under grant 2014-519 and ELLIIT.

© Pierre Moreau 2022

Faculty of Engineering, LTH, Department of Computer Science

ISBN: 978-91-8039-138-2 (print)

ISBN: 978-91-8039-137-5 (pdf)

ISSN: 1404-1219

Dissertation 67, 2022

LU-CS-DISS: 2022-01

Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund 2022

Dedicated to my family
Clélia – Dada – Isabelle – Jean-Jacques

Contents

List of publications	ii
Acknowledgements	iii
Popular science summary	iv
Populärvetenskaplig sammanfattning	v
Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures	I
1 Overview	I
2 Virtual Scenes	3
3 GPU Hardware	10
4 Light Transport	19
5 Evaluation and Methodology	27
6 Research Projects	29
7 Contributions	40
8 Conclusion and Looking Forward	41
9 References	43
Scientific publications	49
Author contributions	49
Paper I: Photon Splatting Using a View-Sample Cluster Hierarchy	51
Paper II: Path Verification for Dynamic Indirect Illumination	65
Paper III: Importance Sampling of Many Lights on the GPU	75
Paper IV: Dynamic Many-Light Sampling for Real-Time Ray Tracing	107
Paper v: Real-Time Rendering of Indirectly Visible Caustics	115

List of publications

This thesis is based on the following publications, referred to by their Roman numerals:

- I **Photon Splatting Using a View-Sample Cluster Hierarchy**
P. Moreau, E. Sintorn, V. Kämpe, U. Assarsson, M. Doggett
In proceedings of Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics, 2016
- II **Path Verification for Dynamic Indirect Illumination**
P. Moreau, E. Sintorn, M. Doggett
Technical report on arXiv:2111.06906
- III **Importance Sampling of Many Lights on the GPU**
P. Moreau, P. Clarberg
Book chapter in Ray Tracing Gems, pp. 255–283
- IV **Dynamic Many-Light Sampling for Real-Time Ray Tracing**
P. Moreau, M. Pharr, P. Clarberg
In proceedings of High-Performance Graphics - Short Paper, 2019
- V **Real-Time Rendering of Indirectly Visible Caustics**
P. Moreau, M. Doggett
To be presented at GRAPP, 2022

All papers are reproduced with permission of their respective publishers.

Acknowledgements

Ph.D. studies can result in countless hours sitting in front of a computer, reading papers, learning new skills, or desperately looking for that last™ remaining bug. However this journey is far from a lonely one, and therefore I would like to acknowledge and thank those persons who helped me through this experience.

I would like to start by giving my deepest gratitude to my thesis supervisor, Michael Doggett, for sharing his experience, guiding me while still allowing me to explore and experiment, and always having an open door to talk about research projects, courses or other thesis-related topics. Those thanks also extend to my thesis assistant supervisor, Jacob Munkberg, for his advice, experience, and insightful comments about on-going projects or when reviewing our papers, and availability. I am deeply thankful to the Swedish Research Council and ELLIIT for funding this thesis and making it a reality.

A huge thank you to Gustaf Waldemarson and Rikard Olajos in the graphics group for all the discussions about graphics-related stuff, and various research-project ideas. I am grateful to Marcus Klang for all the help given with the computer graphics labs and conversations, and all the students I had the chance to supervise, for their interesting questions (and bugs too!) and allowing me to improve my Swedish.

I would like to thank everyone at Unity Labs' Grenoble office as well as at NVIDIA's Real-Time Rendering Research team for their welcome and integration, discussions, and everything I learned there during my internships. Extra thanks to my coworkers from NVIDIA's Lund office for really helping my proficiency at talking in Swedish.

Even if work took a large amount of time, it was not everything. I cannot thank enough all my friends, especially Angeltjee, Cuppcaake, Eggyscrambelini, and Icetronixa, for all the time spent together, laughs, keeping me sane, and providing a safe and welcoming environment, far away from triangles, rays and GPUs. Lastly, I want to express my eternal gratitude to my family, for always being there and supportive, curious about what I am working on, and believing in me so much more than I ever do.

Throughout the thesis, there are multiple renderings of 3D assets produced by different persons, and I would like to thank them for their work and making it available. In Figure 14 and in the back cover illustration, the following assets were used. The Bistro scene [2] is based on assets kindly donated by Amazon Lumberyard. The car model [6] was made by Turbosquid user barteks2, and the helicopter asset [16] by Sketchfab user f3nix and licensed under CC-BY 4.0.

Popular science summary

A lot of work has been put into making visuals in computer games and interactive visualisations look as close to reality as possible. This development mostly occurred thanks to the use of dedicated hardware able to accelerate those applications.

To compute the content of each frame displayed by a computer game, multiple algorithms are used to approximate how the light interacts with objects in a three-dimensional virtual scene. Due to limitations from the dedicated hardware those algorithms rely on, they can only support a limited amount of the different lighting effects found in the real world. With the recent introduction of a new and more flexible hardware unit, a wide range of more precise light-simulation methods can be developed. Several of these techniques were designed and implemented as a part of this thesis, and are described in the following paragraphs.

While walking at night in a city, there will be many different light sources around contributing to what one sees, such as street lights, headlights of cars or buses, and illuminated shop signs. Taking into account all those lights each frame would take too much time, so in practice applications only evaluate a few of those. However, the selection performed is often suboptimal, which led us to look at lighting techniques used in film production. We adapted such a technique which could pick the most important light sources on-the-fly, while still being computable quickly enough. Additional changes were also made to support lights that can freely move about the scene.

One interesting light phenomenon occurs when light interacts with a reflective or refractive medium, and becomes focused into small areas; it can be commonly seen as the moving light patterns at the bottom of swimming pools. This effect has often been faked by real-time applications due to its expensive computation cost. The situation has been changing recently and existing methods can represent the phenomenon when it is directly visible. We developed a way to also process that lighting effect when it is indirectly visible, for example when seen via mirrors or behind transparent objects.

All computations needed to simulate light take time, even when the results do not change from frame to frame in most parts of a scene. We therefore studied which amount of data could be reused over short amounts of time with minimal image quality degradation.

Through the work presented in this thesis, we have shown that more complex lighting effects can be approximated in real-time, and without compromising too much on the results of the simulation. We hope that this will lead the way for new approaches, and that it will be reflected in which light phenomena are simulated by interactive computer graphics applications such as computer games or visualisation.

Populärvetenskaplig sammanfattning

På senare tid har mycket arbete utförts för att öka realismen in datorspel och interaktiva visualiseringar. Framförallt utvecklades detta genom att använda dedikerat hårdvara som kan accelerera dessa applikationer.

För att bestämma innehållet av varje bild som visas av ett datorspel på skärmen, har olika algoritmer använts som approximerar hur ljuset interagerar med objekt i en tredimensionell datormodell. Dessa algoritmer använder dedikerad hårdvara, men hanterar enbart en begränsad mängd ljusfenomen. Denna situation har nyligen förändrats, i och med introduktionen av en ny, mer flexibel, hårdvaruenhet och detta möjliggör en mängd av nya ljussimuleringsmetoder som är mer exakta. Flera sådana tekniker skapades och implementerades som en del av denna avhandlingen, och de beskrivs i de följande styckena.

På en nattpromenad i stadsmiljö kan man se många olika ljuskällor i omgivningen som bidrar till vad man ser, t.ex. gatubelysning, strålkastare på bilar eller bussar och upplysta butiksskyltar. Om ett datorspel skulle försöka behandla alla dessa ljuskällor, hade det tagit för lång tid med traditionell teknik för ljusberäkningar. Därför kan endast ett fåtal av dem utvärderas för varje bild i praktiken. Vilka som väljs är däremot ofta suboptimala, vilket ledde oss till att titta på tekniker som används i filmproduktion. Som ett exempel anpassar vi en teknik som effektivt kan extrahera ett subset av de mest relevanta ljuskällorna för en scen. Vi presenterar också tekniker för att hantera animerade ljuskällor.

Det finns en del ljusfenomen som ofta approximerats relativt grovt av realtidapplikationer på grund av dess höga prestandakostnad: de ljusa mönster som flyttar sig under vattnet, t.ex. på botten av simbassänger. Detta exempel uppstår när ljus interagerar med reflekterande eller genomskinliga medium och fokuseras till små, lokala områden. Numera kan däremot befintliga metoder producera dessa ljusfenomen, men endast då de är direkt synliga. Därav utvecklade vi ett sätt att också beräkna dessa ljusfenomen även när de syns indirekt (t.ex. de som syns i en spegel eller bakom transparenta objekt).

Alla beräkningar som genomförs för att simulera ljus tar tid, och i de flesta områdena i en scen kommer simuleringens resultatet inte att ändra sig från bild-till-bild. Därför studerade vi i vilken utsträckning data kunde återanvändas över korta tidsperioder med minimal negativ påverkan på bildkvalitet.

I denna avhandling har vi visat att mer komplexa ljusfenomen kan approximeras i realtid utan att förlora kvalitet i simuleringens resultatet. Förhoppningsvis banar detta väg för nya metoder och reflekteras i vilka ljusfenomen som simuleras i interaktiva grafikapplikationer, som datorspel och visualisering.

Towards Fully Dynamic Surface Illumination in Real-Time Rendering using Acceleration Data Structures

I Overview

The research presented in this doctoral thesis was undertaken in the field of *Computer Graphics* (CG), which focuses on generating single or sequences of images — a process known as *rendering* — with the help of computers. In order to produce those images, there are many different aspects to take into consideration: how does light propagate throughout the virtual scene, how it interacts with different objects, how do objects move/are deformed, how are objects represented, etc. Those aspects are complex and form their own research sub-areas. In this particular thesis, the discussion will be centred around the first aspect presented earlier, namely the simulation of light propagation within a virtual scene. A more in-depth explanation will be given as progressing through the thesis, but this initial section will focus on describing the importance and impact of Computer Graphics as well as outlining the organisation of this piece of work.

1.1 Computer Graphics

The field of Computer Graphics has evolved considerably since its debut, back in 1963, with Ivan Sutherland's Sketchpad [44] which allowed sketching with a pen on a computer. Today, it has become prevalent in, among others, advertisements, films, software for architecture, design, or even medical software, all kinds of different simulators, and computer games. In some areas, CG has helped reducing costs and improving turnaround by allowing pre-visualisations of designs (of cars and buildings, for example) without requiring the construction of prototypes or models. For others, it improved safety by letting people train in safe but realistic environments, like pilots spending hundreds of hours in flight simulat-

ors before flying a real plane. It also opened new doors in entertainment, allowing users to be visually immersed in virtual worlds like in computer games.

There are two different types of applications to be distinguished in Computer Graphics: *real-time* and *offline*. Real-time CG aims to provide images as fast as possible, and at least 24 per second to give the impression of a continuous flow of images rather than a succession of different images. This is the domain of computer games or simulation software, where the user needs fast feedback on their actions and updates on their surroundings. At the other end of the spectrum lies offline rendering, where the generation of a single image might stretch from a few seconds all the way to several days. Here, the images are not meant for immediate consumption, but will instead be incorporated into a bigger set, like a film for example. Due to the large differences in time constraints between the two types, there is a difference in terms of precision of the computations performed, with offline striving for being as close to the physical behaviour as possible while real-time needs to approximate those behaviours. Real-time will be the focus in this thesis, but several approaches from offline rendering will be presented.

1.2 Outline

Let us first consider the thesis title, to help describe the main areas that will be discussed later. One of the main aspects, sitting in the middle, is the *real-time rendering* part, as it imposes some heavy constraints on the amount of computations that can be performed for generating an image. It is further constrained by needing to be *fully dynamic*, which allows for everything in a virtual scene to be animated and moving: objects, cameras, and even light sources. *Surface illumination* describes the kind of computations performed: evaluating how much light a given surface in the virtual scene receives; volumetric effects such as fog or smoke are not considered. For estimating the light transport within a scene, physically-based models will be used that approximate some of the light propagation. Finally, designing and using different *acceleration data structures* will be decisive in rendering images under those constraints; acceleration data structures provide more efficient accesses to the data they contain.

The field and the required knowledge for understanding the presented research will be explained in the following sections. First, Section 2 discusses the different elements that make up a virtual scene. Next, graphics cards are described to highlight their capabilities and differences compared to processors in Section 3. This is followed by an introduction to light transport, its terminology, and foundational algorithms in Section 4. Section 5 gives an overview of the evaluations performed and the methodology followed while working on the different research projects covered in Section 6 of this thesis. A summary of the contributions made during those projects is given next in Section 7. Finally, Section 8 concludes and opens up about what to expect in the near future.

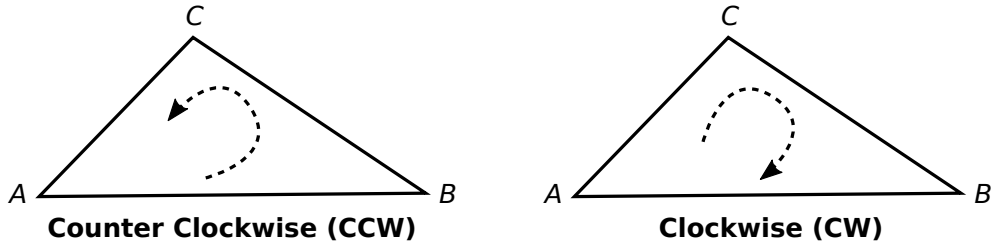


Figure 1: Visualising the difference between counter clockwise and clockwise windings. Starting from the vertex A and for the same three vertices A , B , and C , a counter clockwise winding defines the triangle ABC while a clockwise winding results in the triangle ACB .

2 Virtual Scenes

Before rendering anything, the data to be visualised will be described. At a high level it is a virtual scene, often three-dimensional but that is not a constraint. A scene is made from different types of information, with the main ones being: geometric data (see Section 2.1), materials (see Section 2.2), light sources (see Section 2.3), and cameras (see Section 2.4). All of those are organised by a structure called the *scene graph*, presented in Section 2.5. Finally, an overview of the factors affecting the complexity of a scene will be given in Section 2.6.

2.1 Geometric Data

Geometric data is, along with cameras, one of the mandatory elements needed to render a scene. At its core it consists in points grouped into rendering primitives. For the purpose of this dissertation, only three-dimensional triangles will be discussed as they are the dominant primitive used, but other commonly-used primitives are, among others, quadrilaterals, and Bézier curves.

Points A three-dimensional point P is defined as $P = x \cdot \hat{\mathbf{i}} + y \cdot \hat{\mathbf{j}} + z \cdot \hat{\mathbf{k}} = (x \ y \ z)$, where the unit vectors $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$, along with an origin, form an orthonormal basis. $(x \ y \ z)$ are the coordinates of P in the space defined by that particular basis, and P will have different coordinates when expressed in a different space; Section 2.4 will present commonly-used spaces in Computer Graphics. Points can be used as a rendering primitive, for example when visualising a point cloud generated by scanning a real object using a laser, however in this dissertation they will always be grouped into triangles before being rendered.

Triangles Triangles are the predominant rendering primitive with dedicated support in all existing graphic cards. A triangle has the advantages of being a surface (which most

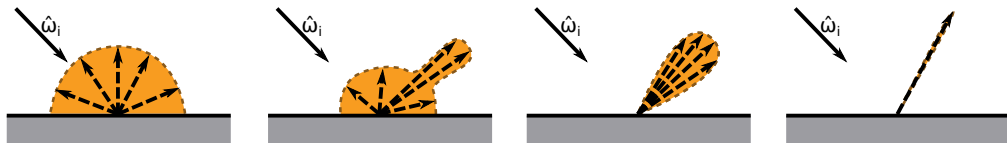


Figure 2: Visualising how an incoming ray $\hat{\omega}_i$ can be reflected by different types of reflective materials. From left to right: diffuse, glossy, near-specular, and specular.

rendered objects are), and all of its vertices will always belong to the same plane (unlike a quadrilateral where the fourth vertex could be sticking out of the plane defined by the three other vertices). A triangle is specified by providing the three points (aka. *vertices*) in space located at its “corners”, and the relative order in which those points are given which is also known as *winding* (see Figure 1 for a description of winding).

Geometric normals With the vertices and winding of a triangle, it is possible to compute its normal $\hat{\mathbf{n}}$ (i.e. a vector of unit length perpendicular to the plane defined by the vertices of the triangle):

$$\hat{\mathbf{n}} = \text{normalise}((P_1 - P_0) \times (P_2 - P_0))$$

where P_i represents the i -th vertex of the triangle. From there, a *front* and *back* side of the triangle can be defined: the front side lies on the same side as the normal, and the other side corresponds to the back. As the normal is computed based on the order of the triangle vertices, misinterpreting the winding of a triangle will result in inverting its normal and swapping its front and back side.

Meshes A mesh is a set of connected triangles, generally forming a single object.

2.2 Materials

While geometric data describes the shape of objects, materials model how the light interacts with objects: for example, is the object reflective and if so how reflective is it, or is it transparent. Only an overview of materials will be given here; for a more in-depth presentation on the subject, textbooks such as *Physically Based Rendering* [38] are recommended.

This interaction is represented by a function called the *bi-directional x distribution function* (BxDF), where x reflects the type of interactions handled: for example, if only reflections are handled the term BRDF is used with the ‘R’ standing for *reflectance*. Other common denominations are BTDF (‘T’ for *transmittance*, i.e. transmission through transparent objects), BSDF (‘S’ for *scattering*, which handles both reflectance and transmittance), and BSSSDF (‘SSS’ for *sub-surface scattering*, used to model among others skins, marble, and liquids such as milk).

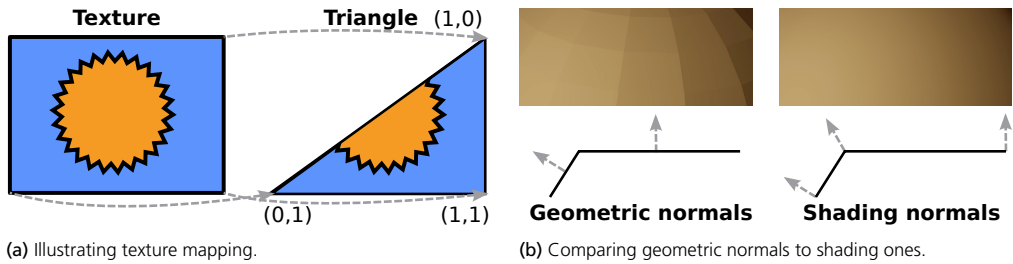


Figure 3: Showcasing what different vertex attributes are used for, here for texture coordinates on the left, and per-vertex normals on the right.

Looking at scattering, three main behaviours are usually described: *diffuse*, *glossy*, and *specular*. A diffuse material reflects or transmits light equally in all directions, giving a matte look as no region will appear brighter than others, regardless on the viewing angle. At the other end of the spectrum are specular materials which reflects or transmits light in a single direction, like a clean mirror for example. Most materials lie in between the two previous types, and are referred to as glossy. Figure 2 visualises how they each, for a given incoming direction, reflect light.

It can be interesting to vary the material properties over an area, to add some rust in some places on a metal object for example. One way to do so is to increase the density of triangles in the area, such that each triangle will only cover a sub-area with identical material parameters. However this can dramatically increase the number of triangles in the scene and the cost of rendering such a scene, and that means editing the mesh every time new material details are added to the model. To alleviate those issues, a technique called *texture mapping* [12] (visualised in Figure 3a) was developed which maps an image (called a *texture map*, or just *texture*) onto the mesh and its triangles via a set of coordinates called the *texture coordinates*. Texture coordinates are often specified per vertex, and the values for all other locations on the triangle are obtained by interpolating between the values given by each vertex. Texture mapping can also be used to add holes or transparency to objects by storing an *alpha* value representing the opacity of an object, with 1 corresponding to fully opaque and 0 to fully transparent. *Alpha testing* checks whether the alpha value is below a given threshold, in which case the current geometry location should be ignored and whatever is located behind it becomes visible.

If the geometric normal is used when evaluating the materials, a high density of triangles will be required to represent curved surfaces without looking faceted due to sudden changes in normal direction. To avoid needing too many triangles, it is common to specify a normal per vertex which can differ from the geometric normal; it is known as the *shading* normal, as it is used during shading, to differentiate it from the geometric normal. Shading is the act of computing the perceived colour of a visible surface, for example by computing how much light it reflects towards the camera sensor. Similar to material properties varying

spatially through the use of texture maps, normals can also be stored in *normal maps* to provide additional granularity than specifying them at each vertex. Figure 3b illustrates the difference between using geometric and shading normals.

2.3 Light Sources

For us to see anything, light has to reach our eyes (or the sensor of a virtual camera) either directly or after interacting with different objects around us; that light is emitted by a light source. There are several types of light sources used in Computer Graphics, and those types belong to one of the following categories: “global”, point, and area lights. The terms “light” and “light source” will be used interchangeably in this thesis.

“Global” Lights “Global” light sources will affect the whole scene, as opposed to the other two categories which are a lot more localised as to their influence range. This group contains two different types: *directional lights*, and *environment maps*. The former is a light source that has no location but emits all of its light as parallel rays along a specified direction (hence the *directional* name). It is used as an approximation for the direct lighting coming from the Sun: all the rays coming from the Sun appear to be parallel as they hit the Earth, due to the large distance between the two celestial bodies.

Environment maps consist of one or more images, depending on the representation used. They can be used to cheaply represent what a user can see in the distance, like distant mountains on the horizon, and act as light sources, with the Sun contributing vastly if present in the image(s). The sky-part of the images can also be taken into account to approximate all the light scattering that happens in the atmosphere and still provide some light even right after the Sun has gone down, for example.

Point Lights Contrary to “global” light sources, *point lights* do have a position and will be carefully placed by artists to provide lighting in key locations. The name comes from those lights representing a singular point in space and having no area. They emit light uniformly in all directions, except for a subcategory called *spotlights* which constrain their emission profile to within a given cone. This type of light sources has been widely used in real-time applications like computer games, as it is computationally cheap to evaluate their contributions and the shadows they produce, while still giving good visual results.

Area Lights While point lights are convenient, they can not accurately represent the lights that surround us, especially those that have a large surface area. This is due to the visibility between a location in the scene and the area light varying depending on the selected location on the light source. This adds a third category of lighting conditions from a given

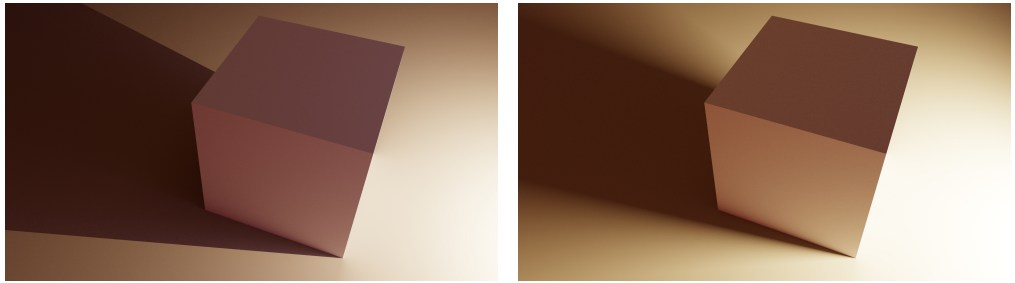


Figure 4: Visualising the difference between the hard shadow cast by a point light, on the left, and a soft shadow from an area light, on the right. Point lights will always create those sharp transitions between being fully lit and being fully in the shadows. Area lights can also create sharp transitions under certain circumstances, but are overall known for their softer transitions.

light source, to the existing *fully lit* (the whole light source is visible) and *fully in shadow* (the whole light source is occluded): *partially lit*, i.e. only part of the light source is visible, resulting in *soft shadows* rather than the *hard shadows* from point light sources. The difference between the two is visualised in Figure 4. Note that the *penumbra* (i.e. the partially lit area) can be very limited to non-existent if the area light subtends only a small angle of the hemisphere of the surface being lit: for example, despite the Sun having a large surface area, it is located so far away from us that it is barely more than a small disc in the sky

One of the area light types, *analytic lights*, is similar to point lights in that it has no underlying geometry and is purely virtual. The other type of area lights are *mesh lights*, which are parts of the geometry itself (see *meshes* in Section 2.1), like individual triangles, that emit light. This provides the most flexibility in terms of lighting as one can, for example, sculpt a very complex neon sign and let it do the lighting without having to painstakingly place analytic or point lights in various places to try to approximate it. The problem is that the number of potential light sources grows dramatically as a result: where one light bulb could be approximated as a single point light, its mesh might be made of twenty or even a hundred triangles depending on the level of detail, increasing by one or two orders of magnitude the number of light sources; this topic will be further discussed in Section 6.1.

2.4 Cameras and Representation Spaces

A virtual camera is the entry point for a user into a virtual scene: it lets them, among others, look around at what is present, move, and zoom in or out. An actual camera, like a *digital single-lens reflex* (DSLR) camera or even our own eyes, are complicated systems whose characteristics affect the generated image. For example, those systems have an aperture that is not infinitesimal, resulting in more light reaching the sensor but also in some objects being out-of-focus and appearing blurrier: *defocus blur*, or *depth of field* effect. As the focus was on surface illumination and not on camera effects, a simple camera model was used

throughout the research: the *pinhole camera* model.

A pinhole camera consists of an infinitesimal hole letting light shine on the sensor placed behind it. The camera has several attributes, such as a position, a direction in which it is looking, and a vector pointing upwards to define how the camera is oriented around the looking direction. As light falls onto the sensor, a projection occurs from a 3-dimensional point onto a 2-dimensional surface. Different types of projection exist for different purposes, but the one most commonly used in Computer Graphics is the *perspective* projection, which projects object similarly to our eyes. For example, perspective projection makes objects further away look smaller than similarly-sized instances located near the camera. The perspective takes several additional parameters, such as a near and far plane to delimit the distances at which objects can be observed, and a horizontal and vertical field of view to delimit how wide the camera can see.

The projection performed by a camera is only one of many space transformations applied to vertices in a scene during rendering. As their properties can influence algorithm and acceleration structure designs, an overview of the most common ones will be given. The first space encountered, *object* or *model* space, is used during the creation of the various objects to be placed in a scene. In this space, the origin will usually sit at the centre of gravity of the object, or any other location that facilitates the design process; it is also the space the geometry of an object is expressed in when being imported into an application. As object space is specific to each object, combining multiple objects into a common environment needs a new space: *world* space. There are no specificities regarding the choice of its origin and axes, the important part being that all geometries, light sources, and cameras will be positioned and oriented in that space. To simplify some of the computations when rendering, the *camera* or *view* space can be used; they are simply a modification of the world space to place the camera at its origin and orientate it in a specific way. Applying the projection transform will move to *clip* space where it is possible to delimit the volume visible to the camera which is known as the *view frustum* and looks like a truncated pyramid with the near plane cutting off the top of the pyramid, and the far plane being its base. From clip space, *normalised device coordinates* (NDC) can be reached via a specific division which transforms the view frustum into a cube where all positions within the view frustum have their coordinates in the range $[-1, 1]^3$. Finally, applying an affine transform on NDC and dropping the depth component will result in *screen space*, a two-dimensional space defined over $[0, w] \times [0, h]$ with w and h the width and height respectively of the rendering resolution.

2.5 Scene Graphs

As seen in the previous section, models will usually be imported in model space but need to be placed in a common environment called world space. Where to position the objects

could be left up to the application, but in most scenarios it is best to leave it to artists to carefully place them. This information is then retrieved by the rendering application thanks to a structure called the *scene graph*. As the name implies, it is not just a list but an actual graph (usually a *directed acyclic graph* (DAG)), meaning any nodes can have children which can have children themselves, and so on. This allows for hierarchical placement and animations: a table node could have a book node as child, resulting in the book only needing to know how it is positioned relative to the table and does not need to know where the table itself is located, as the book world transform can be obtained by combining its own transform (relative to the table) and multiplying it by the transform of all its ancestors. This considerably simplifies the animation of objects that are related: for example, animating the table will automatically result in all of its children being affected by that animation. Additionally, multiple nodes could point to the same mesh with different transforms, allowing for a mesh to be *instanced* (i.e. duplicated) without having to store the same geometry multiple times in the scene file.

2.6 Scene Complexity

As a last point on virtual scenes, an explanation and discussion about scene complexity will be given. It is an important topic while working on a research project, as using more complex scenes will help test how well the method scales.

The parameters emphasised when discussing the complexity of a scene will depend on the use case, and some might not even be considered complex for certain algorithms. A list of different parameters will be presented, along with the main areas they impact.

Triangle count It has a direct impact on rendering performance (and on memory usage), be it for rasterisation or ray tracing.

Depth complexity For rasterisation this would be the number of triangles covering a given pixel, and affects rendering performance in the sense of “wasted” work: a triangle might have been shaded, only to realise afterwards that it is being hidden by another triangle and none of that shading will actually be used in the final image. In ray tracing the number of primitives hit by a ray will impact performance, especially in the case of transmissive objects. There is also the case of rays glancing right past objects, that might still hit the bounding boxes but not any of the actual geometry; Section 4.6 will explain what the bounding boxes are, incurring most of the cost of traversing the acceleration structure for nothing in return.

Materials The number of different materials and their complexity will have a large impact on rendering performance, regardless of the rendering method. Some types of ma-

terial like specular reflective or refractive ones will require additional computations to find out what is being reflected or what is located behind the object.

Light sources The amount of light sources impacts performance if trying to evaluate as many as possible, or image quality if only a subset are considered. The type of light sources also affects performance, as area light sources require more samples than point lights.

Animations Complex animations might require physics simulations to properly animate the cape of a player as they move around, for example, or other types of costly computations. Animated objects will require various acceleration data structures to be updated, like in ray tracing for example, which adds to the total frame time. It may also affect the image quality (if an algorithm relies on temporal reuse), as reusing results from previous frames will be more complicated due to the scene layout changing or the camera now looking somewhere else.

3 GPU Hardware

In this section, an overview will first be given of how a *Graphic Processing Unit* (GPU) differs from a *central processing unit* (CPU) and important concepts (Section 3.1), before diving into the pipelines specific to GPUs (Section 3.2) and how different ones can be used to solve a fundamental problem when rendering: figuring out which surfaces are visible from a given point of view (Section 3.3). Finally, in Section 3.4 the evolution over time of GPUs will be briefly discussed, especially during my Ph.D. studies which saw some drastic changes.

Note: Only NVIDIA GPUs were used throughout the work presented in this thesis, leading to the author being a lot more knowledgeable about that hardware than the one from competitors, and to them being cited more often regarding new features or similar. As a result, the names from the CUDA API will be used to refer to some concepts, so keep in mind that those might be called differently in other APIs.

3.1 Comparing CPUs and GPUs

The use of GPUs has allowed for more detailed and realistic-looking renderings than if CPUs were used for rendering, as rendering is massively parallel and GPUs were designed for that purpose. Due to its predominance in rendering, all algorithms presented in this thesis being specifically tailored for GPUs, and the reader not necessarily being familiar with how GPUs work, this part presents the differences between CPUs and GPUs. If we take a top of the line consumer GPU, at the time of writing, e.g. an NVIDIA RTX 3090,

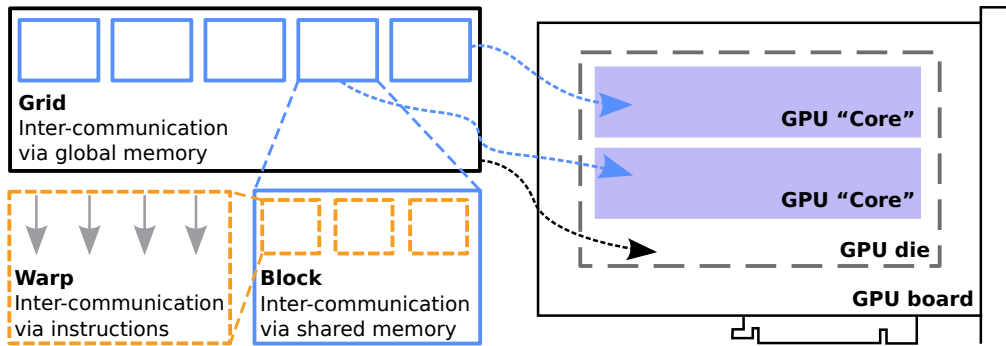


Figure 5: An overview of the execution hierarchy of a compute dispatch on a GPU, and how primitives within a level can communicate between themselves.

it supports 128 threads running concurrently on a single multi-processor which it has 82 of, while a top of the line consumer CPU like an AMD R9 5950X will support 2 threads on each of its 16 cores. Of course this does not tell the whole story, like CPU threads will run at a way higher frequency than GPU ones, but it underlines the differences in terms of parallel computations and how the memory management might be quite different to provide the data to all those threads in a timely fashion.

Execution On both CPUs and GPUs, most of the code is written from the perspective of a single thread (represented as a grey arrow) executing on the hardware; the different shapes mentioned in the text can be seen in Figure 5, which summarises most of the explanation below. While that is indeed the case on CPUs where most threads will work independently from each others, on the GPUs they will run in groups (the orange boxes) i.e. all threads within a group will execute the same instruction at the same time but with different operands; details may vary between vendors and generations, but this remains the main exposed behaviour. Threads within a group can communicate with each other using dedicated instructions. These thread groups, or *warps*, are not on their own either, and are themselves grouped into a *block* (the blue boxes); all warps of a block are guaranteed to be running on the same GPU core (light purple area), which allows them to communicate between each other using some local memory called *shared memory*. Finally, blocks are partitions of a final entity called the *grid* (the black box), which corresponds to the set of all threads requested by the developer to perform some work.

As GPUs can be quite different from CPUs, dedicated APIs have been designed to allow for efficient program execution on GPUs. The most common ones on desktops are DirectX 11/12, OpenGL, and Vulkan for graphics or compute workloads, and CUDA or OpenCL for compute ones.

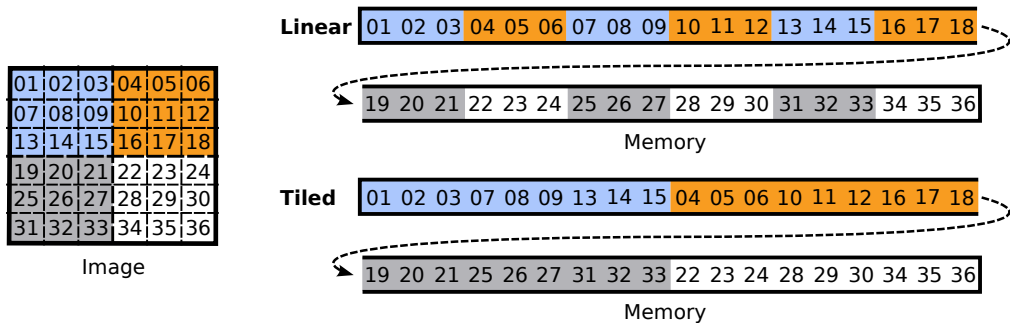


Figure 6: A visual representation of two ways of storing an image. Each 3×3 tile is colour-coded to be more easily distinguished.

Memory System Data can be sent between the CPU and a discrete GPU over the *peripheral component interconnect express* (PCI-e) bus, in which the graphics card is plugged in; integrated GPUs (i.e. located on the same die as the CPU) have different characteristics in that regard, and will not be discussed here. Given the large amount of threads, sending all the data required to keep those threads busy over PCI-e each frame would not be energy efficient and would take too much time. As a result, discrete GPUs have their own memory known as *video random access memory* or VRAM, similar to the (system) RAM associated to a CPU. VRAM is designed for massively parallel platforms, and as such prioritises bandwidth to move as much data as possible at once, at the cost of latency (how long it takes for one packet to arrive at the destination). To offset the high latency of VRAM, GPUs are built with over-subscription in mind: they can keep track of more threads than they can run concurrently, allowing them to pause threads while they are waiting on data and switch to a different group of threads which is ready to be executed. GPUs are designed to allow very fast thread switching unlike on a CPU, and so starting more work than your GPU can run at once is not detrimental to performance in most scenarios. Even if VRAM has a high bandwidth, one has to take care of how different threads within a group access memory to try to make the most out of the resources. If each thread is reading from (or writing to) completely random areas in memory, which can easily be the case with ray tracing, the group will be constantly waiting on memory and not getting any progress done.

Threads access two different types of memory resources. The first one are *buffers*, which are just arrays whose elements are stored contiguously, and in order, in memory. *Textures* or *images* make the other type and are often used for actual images. As operations on images will often look at a small 2-d neighbourhood, textures elements (or *texels*) will be stored as tiles to optimise for those accesses; a visualisation of the two different types of storage can be seen in Figure 6. Additionally when accessing a texel, continuous coordinates are used making it possible to try to read in-between two elements: for example if a texture has two elements, one stored at index 0 and one at index 1, it is allowed to try to access the element at index 0.3 even if it does not exist. What happens in those cases can be configured by the user, and common behaviours are returning the closest element (in this case, `texture[0]`),

or returning a linear interpolation of the two closest elements weighted by the distance ($0.3 \cdot \text{texture}[0] + 0.7 \cdot \text{texture}[1]$). This is performed automatically by the hardware.

3.2 Pipelines

When learning about Computer Graphics and how to program a GPU to get an image on the screen, the term “pipeline” will always show up sooner or later. It describes a set of stages performed on the GPU hardware in a pre-defined order, to compute something at the request of the user. Some of the stages might use fixed-function hardware while others can run programs written by the user, but in both cases the input data can only come from previous stages in the same pipeline, or from operations which occurred earlier.

At the time of writing, there are three different pipelines available in modern rendering APIs: rasterisation, compute, and ray tracing. The compute pipeline only consists of a single stage which runs a user-written program and uses GPU memory to get its inputs and store its outputs. This pipeline is commonly used nowadays in computer games or rendering algorithms to compute post-processing operations or other highly-parallel computations. The other two pipelines are more complex and will be described in their own sections further down as a result.

Rasterisation

The first consumer-level graphics hardware, the Voodoo, was first introduced back in October 1996 by 3dfx Interactive, and featured a rasterisation pipeline consisting of only fixed-function hardware with some limited configuration. In 2001–2002, some of the fixed-function stages were replaced with programmable ones which were quite limited at first, but grew more and more powerful. Today it consists of 13 stages, 5 of which are programmable, but even after 25 years of evolution it is still subject to changes with important modifications proposed by NVIDIA in 2018. The current pipeline and the proposed edits can be seen in Figure 7. As they do not play a central role in the research presented in this thesis, only a brief description of the different stages will be given.

Starting with the more traditional pipeline (corresponding to the left and centre column in Figure 7), the *draw* command kicks off the start of the pipeline and might perform some configuration before the first actual stage, the *input assembler*, is run. The input assembler is responsible for fetching the different attributes referenced by the next stage, and making those available, allowing it to be oblivious to how those attributes are stored in memory. For example, the mesh might use indexing so the input assembler needs to first read out the indices used for each primitive in order to provide the correct vertices to the vertex shader as all of that is completely transparent to it. Following the input assembler is the first

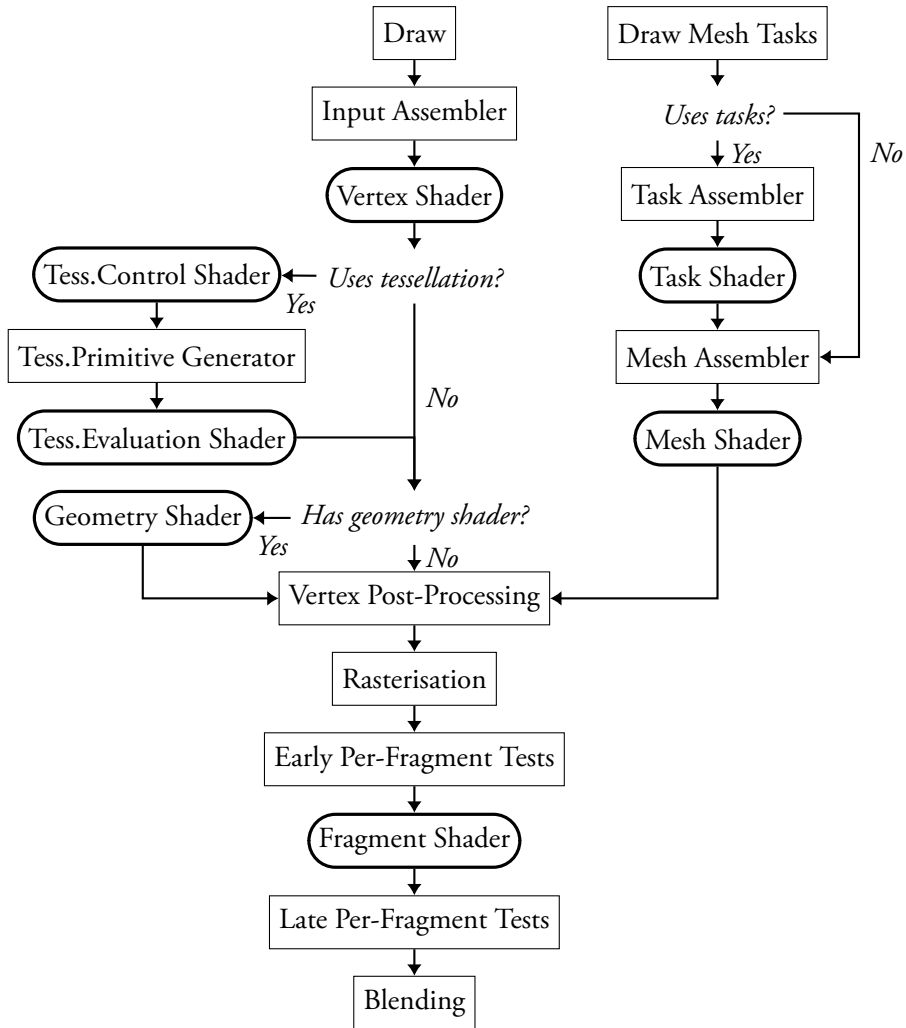


Figure 7: An overview of the different stages of the traditional and newly-proposed mesh shader-based rasterisation pipelines in Vulkan. The traditional pipeline corresponds to the left and centre column, while the mesh shader-based one is made of the right column and the bottom of the centre column (starting with *Vertex Post-Processing*). All stages are represented by rectangles, with rounded corners ones being implemented in programmable shaders, and the other ones in fixed-function hardware.

programmable stage, the *vertex shader* which takes care of transforming each vertex from their current space (usually object space) into clip space for the rasteriser to work with, and generating attributes consumed by later programmable stages. Among the transformations commonly applied here, vertices are translated, rotated, and scaled, to reflect their status relative to the camera in the current frame, as well as apply the projection of the camera. After the vertex shader comes an optional block of three different stages, added in 2009, to perform *tessellation*. Tessellation can dynamically increase the level of detail of a surface by

splitting it into smaller triangles, making it possible to keep low resolution version of the object if it is located far away from the camera, but generate a higher resolution version if it moves closer. The resulting primitives are then sent to another optional stage, introduced in 2006: the *geometry shader*, which can transform a primitive into another type of primitive. It can be used for the rendering of particle effects, where each particle is stored as a single position (its centre) to save on memory space, which the geometry shader then expands into a small quad before it is rendered. The rigidity of some aspects of the pipeline described up to now, as well as the fact that computer games will often run compute shaders, prior to even starting to draw, to discard whole objects that are not visible. This has led to looking for a more compute-based approach to that part of the pipeline. One such proposal is the mesh-shader method proposed by NVIDIA, and visualised in the third column of Figure 7.

Once the position and shape of the different primitives has been finalised, the *vertex post-processing* is run to discard primitives located entirely outside of the view frustum, or *clip*¹ those which are only partly outside. Finally, the *rasterisation* stage — which gives its name to the whole pipeline — is reached. Its primary role is to *rasterise* the primitives, i.e. evaluate which pixels are covered by them, and compute the attributes for each individual pixels. This computation is performed by interpolating between the value associated to each vertex of the primitive, for all the different attributes. The primitive will be diced into *fragments*, where each fragment covers a single pixel. As primitives are not necessarily sorted before being drawn, it can occur that some of its fragments being currently processed happen to lie behind some fragments of another primitive that was processed earlier. If the fragment being processed turns out to be hidden, there is no reason to continue processing it and spending resources on something which will not be used. This is one of the main roles of the *early per-fragments tests* stage. If the fragment passed the early tests, the *fragment shader* will be executed, and is commonly responsible for evaluating the lighting and material at the current world position to compute the colour stored at this pixel. Since the fragment shader is allowed to modify the depth of the fragment, instead of performing the tests right before the fragment shader, they will be run right after it resulting in the *late per-fragments tests*. Once the fragment has been shaded and the tests have been successful, the outputs of the fragment shaders (usually the colour of the pixel) will be written into a texture during the final stage, known as *blending*. Instead of just overwriting whatever value was previously stored at a given pixel, it can instead blend them together (hence the name), to simulate a transparent object for example; it is a fixed-function stage, but the type of blending performed is customisable.

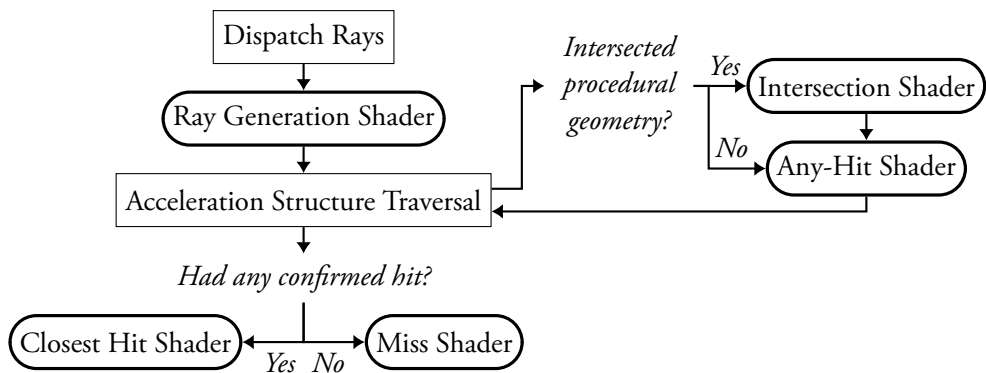


Figure 8: An overview of the different stages of the current ray tracing pipeline in Vulkan. All stages are represented by rectangles, with rounded corners ones being implemented in programmable shaders, and the other ones in fixed-function hardware.

Ray Tracing

The current ray tracing pipeline used on GPUs initially debuted in 2010 with OptiX [37]. However this might change to better suit applications, as ray tracing is seeing increase usage in computer games and other real-time applications following the introduction of dedicated hardware in GPUs for accelerating it. Nonetheless, this section will present the ray tracing pipeline as it is found today, in 2021, in DirectX Raytracing (DXR) and Vulkan ray tracing (VKRT) extensions.

The ray tracing pipeline is composed of five different programmable stages; an overview of the pipeline is presented in Figure 8. The first stage, called *dispatch rays*, is responsible for starting the appropriate number of threads and configuring the hardware before the next stage, the *ray generation shader* which is the “brain” of this pipeline, computes and traces the initial batch of rays against a specified acceleration structure.

The hardware will then traverse the acceleration structure with each ray in search for intersections. In case the acceleration structure contains *procedural* primitives, they are represented as *axis-aligned bounding boxes* (AABBs) inside the acceleration structure and the hardware will call an *intersection shader* to resolve whether the custom primitive stored in the AABB was actually hit or not; in the case of triangles, the hardware will resolve the intersection test itself, usually leading to increased performance when using triangles over procedural primitives. Once an intersection has been found, the *any-hit shader* will be invoked, if specified; it is mainly used to perform *alpha testing* (see Section 2.2) and tell the pipeline to ignore the current hit if the test fails, but can serve other purposes, like accumulating the contribution of a photon for each collection point in range as presented in

¹Clipping will either modify the primitive, or replace it by multiple new primitives, while ensuring that the final primitive(s) describe exactly the shape of the initial primitive for the part located inside the view frustum.

Paper v. Note that this shader will not be invoked for every possible intersection along a given ray, nor in any defined order, but only for those located in the distance range specified on the ray and which are closer than the closest intersection found until now.

At the end of the search and before handing back control to the ray generation shader, either the *miss shader* will be invoked if no intersection was found nor kept, or the *closest hit shader* otherwise. Data can be communicated between the different stages via a special structure called a *payload*, which can be read and written to from all except for the intersection shader. The closest hit shader is the only other stage, apart from the ray generation shader, which can trace a new ray, allowing for recursion. It is however not recommended in current best practices, and one should rather trace new rays from the initial ray generation shader or by launching a new ray tracing pipeline that will continue from where the previous left off.

3.3 Solving Primary Visibility

Being able to evaluate point to point visibility between two random positions in space, A and B , for example to know if a point lies in shadow or not, is an important type of query for rendering. To compute the distance to the closest surface seen from A along the vector \mathbf{ab} , all triangles potentially involved would need to be rasterised. If many visibility queries from different origins are required, rasterisation is not able to efficiently produce those results.

On the other hand, the rasteriser hardware was designed for solving primary visibility, which consists in figuring out all positions that are directly visible from the camera. Indeed, it can make use of the high coherency between nearby pixels to efficiently perform its work; this also applies to primary visibility from light sources, which is why shadow maps have been used for so long. Rasterisation remains the most performant way of solving primary visibility in most scenarios.

As part of improving the performance of rendering a scene using rasterisation, *Geometric buffers* (or G-buffers) were introduced [39]. A problem with rasterisation is that a triangle might successfully pass the z -depth and run all of its fragment shaders, only to be later occluded by another triangle, resulting in all those computations going to waste. Fragment shaders are by far the costliest stage to run in modern games due to the complex lighting and materials in use. To avoid that, the idea is to first rasterise all the triangles as usual, but instead of shading in the fragment shader, only *geometry*-related data will be stored into *buffers* on a per-pixel basis, like the normal, material ID, texture coordinates, etc. The shading takes place in a second pass that will read that data, and perform its computations only once per pixel; as the shading is postponed to a later pass, this technique is called *deferred shading*. Some of the data in the G-buffers will be overwritten multiple times, but it remains cheaper than the alternative.

This technique of generating G-buffers is applicable to ray tracing applications, for example to solve primary visibility using the rasteriser for increased performance (the G-buffers are then used to initialise the ray tracing from that first hit), or to be used in a post-process step, like filtering, which usually rely on some geometry information (like normals) alongside colours.

3.4 Evolution

GPUs have changed since I embarked on my Ph.D. studies, and this section will provide a brief summary of their evolution, starting with new features and improvements. One of the first new feature which had an impact on the research, was *pre-emption* being supported, which allowed a GPU to switch from executing one pipeline, to executing a completely different one. This is different from the switching of threads described in Section 3.1, as that only takes place between threads from the same pipeline dispatch. From a user perspective, pre-emption meant that if a shader in an application took too long to run, it would no longer result in the whole screen freezing until it completed. From a developer perspective, it was now possible to debug a graphical application on the same machine one was using, without needing to run the application on a different GPU than the one to which the screen was connected. New hardware was introduced to efficiently run machine learning algorithms, helping the democratisation of real-time machine learning-based denoisers for example, but the major change for the work presented in this thesis, was the introduction of custom hardware to accelerate ray tracing.

Comparing GPUs in 2021 to those in 2015, the number of streaming multi-processors per GPU has increased by a factor of four, resulting in similar changes in the amount of single precision floating point operations per second, while memory bandwidth lagged a bit behind and ended up being $3\times$ higher today. The amount of VRAM available has been changing suddenly and by larger amounts: for example, on the very high-end side, it started at 12 GiB and stayed there for two generations, then doubled and remains there for the second generation in a row. On the lower end, it started at 2 GiB, tripled to 6 GiB for two generations, to now end up at 12 GiB. One of the possible reason for the increase in VRAM for the current generation, could be the new consoles (released around the same time) shipping with 16 GiB VRAM. All the numbers above were from looking at the first $x60^2$ and TITAN released per generation by NVIDIA, from the GeForce GTX 960 to the GeForce 3060, and GeForce GTX TITAN X to the GeForce RTX 3090.

²i.e. card models ending with 60, such as 960, 1060, 2060, and 3060

4 Light Transport

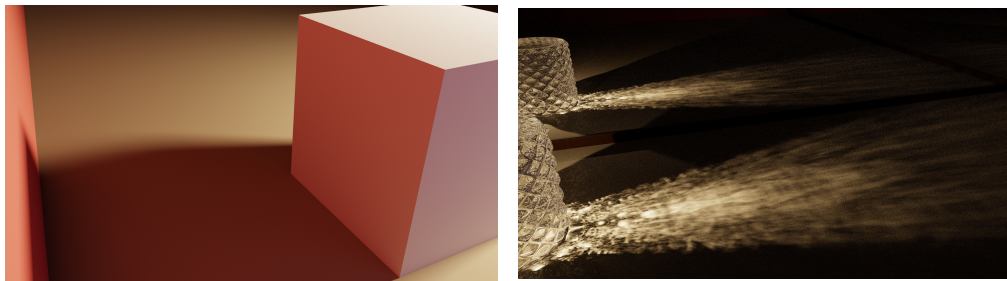
The work presented in this thesis revolves around the efficient simulation of light transport, i.e. how light travels within a scene, from where it is emitted, to where it enters our eyes or a camera sensor after having interacted with different objects in that scene. While real-time computer graphics aims to stay close to physically-based rendering, it does not simulate how atoms are excited to emit photons nor how complex optical sensors work, instead relying on coarse approximations. Similarly, light is assumed to travel in straight lines in a vacuum, only changing direction as it interacts with volumetric effects (absent from this thesis) or when changing medium, and therefore no wave optics is present.

Many different approaches have been proposed over the years to efficiently simulate light transport, looking at different aspects of the problem. Only the ones directly relevant to the methods described in this thesis will be presented. First, the terminology specific to computer graphics will be introduced in Section 4.1. To symbolise the problem at hand, an overview of the *rendering equation* will be given in Section 4.2, along with some of the tools used to estimate it. In Section 4.4 and 4.5, an explanation will be given of the two main classes of algorithms used in evaluating the rendering equation, the first one starting from the camera and the second from light sources. Those algorithms would run much slower if it was not for the dedicated acceleration data structures that will be described in Section 4.6.

4.1 Terminology

When evaluating light transport, different types of light quantities are used. Their names come from radiometry which focuses on measuring electromagnetic radiation. A light source emits a certain amount of energy per unit of time, known as the *radiant flux*, Φ , and often simply referred to as *flux*. *Flux density* or *irradiance*, E , provides how much flux is received by a surface per unit area. The main quantity used is called *radiance* and is denoted L . It represents the amount of flux emitted (in which case L_e will be used rather than L), outgoing (L_o , i.e. reflected and transmitted), or incoming (L_i , i.e. received) by a surface, per unit of direction and per unit of projected area. Radiance remains constant as a ray travels through a vacuum, only changing as it interacts with matter.

As light interacts with objects, different effects can appear as illustrated in Figure 9. *Colour bleeding* is one of those and takes its name from the colour of an object *bleeding*, i.e. being carried over, onto another object. One such example is sunlight hitting a red brick wall before being reflected onto a white wall. The cube in Figure 9a is similarly affected: it is of a grey–white colour (seen on the top face), but has its left face tinted red by the red wall to the left of the image, and front face lightly tinted purple from the similarly-painted wall



(a) Colour bleeding

(b) Caustic

Figure 9: Various common global illumination effects.

The transparent glass object used in Figure 9b was made by Simon Wendsche, and is provided as one of the scenes for the 3rd edition of *Physically Based Rendering* [38].

found behind the camera. Reflective or refractive surfaces, on the other hand, can create *caustics* by concentrating light and generating those brighter areas. These are commonly seen at the bottom of swimming pools, but can also be found at home like in Figure 9b from a block of glass.

Different rendering resolutions are used in this thesis and in the attached papers. The two main ones are *1080p* which corresponds to 1920×1080 pixels, and *4k* which here refers to 3840×2160 pixels.

4.2 The Rendering Equation

Sections 2.2 and 2.3 presented some of the representations of light–material interactions used in Computer Graphics, as well as the emission of light, but no formulation of the light transport was given at the time. This is where the *rendering equation* (1) from James Kajiya [24] comes into play, providing an equation for computing the outgoing radiance off a particular point, P , and along a specific direction, $\hat{\omega}_o$:

$$L_o(\hat{\omega}_o, P) = L_e(\hat{\omega}_o, P) + \int_{\hat{\omega}_i \in \Omega_p} \rho(\hat{\omega}_o, \hat{\omega}_i, P) L_i(\hat{\omega}_i, P) (\hat{\mathbf{n}} \cdot \hat{\omega}_i)^+ d\hat{\omega}_i \quad (1)$$

where $\hat{\mathbf{n}}$ is the normal of the surface at location P , Ω_p the set of all unit vectors located in the hemisphere surrounding $\hat{\mathbf{n}}$ and pointing away from P , $\rho(\hat{\omega}_o, \hat{\omega}_i, P)$ the BRDF describing the interaction of light and material at P , and $(a \cdot b)^+$ the dot product between a and b clamping negative values to 0. The formulation given here ignores refraction, for simplicity.

$L_i(\hat{\omega}_i, P)$ is estimated by evaluating Equation (1) again, but this time at the first hit H found when tracing a ray from P along $\hat{\omega}_i$, so by computing $L_o(-\hat{\omega}_i, H)$. This recursivity and the fact that any surface in the scene could theoretically contribute to any other surface are what make physically-based rendering a challenge for real-time rendering. By replacing $L_i(\hat{\omega}_i, P)$



Figure 10: Visualising the impact of indirect illumination in certain scenarios. On the left, only direct lighting is present, while both direct and indirect are computed in the image on the right. The renderings were taken from the author’s master thesis. The Sponza scene was created by Frank Meinel and is available under the CC-BY-3.0 license from Morgan McGuire’s Computer Graphics Archive [32].

with the evaluation of $L_o(-\widehat{\omega}_i, H)$ and refactoring the resulting equation, the contributions of *direct* and *indirect* lighting can be highlighted:

$$\begin{aligned}
 L_o(\widehat{\omega}_o, P) &= L_e(\widehat{\omega}_o, P) \\
 &+ \int_{\widehat{\omega}_i^P \in \Omega^P} \rho(\widehat{\omega}_o, \widehat{\omega}_i, P) (\widehat{\mathbf{n}}^P \cdot \widehat{\omega}_i^P)^+ L_e(-\widehat{\omega}_i^P, H) d\widehat{\omega}_i^P \\
 &+ \int_{\widehat{\omega}_i^P \in \Omega^P} \left(\rho(\widehat{\omega}_o, \widehat{\omega}_i, P) (\widehat{\mathbf{n}}^P \cdot \widehat{\omega}_i^P)^+ \times \right. \\
 &\quad \left. \int_{\widehat{\omega}_i^H \in \Omega^H} \rho(-\widehat{\omega}_i^P, \widehat{\omega}_i^H, P) L_i(\widehat{\omega}_i^H, H) (\widehat{\mathbf{n}}^H \cdot \widehat{\omega}_i^H)^+ d\widehat{\omega}_i^H \right) d\widehat{\omega}_i^P
 \end{aligned} \tag{2}$$

The second line in that equation corresponds to all the radiance emitted by all light sources in the scene that reaches P without interacting with anything else on the way, hence the term *direct* lighting. While the third and fourth lines compute all the radiance being reflected off all objects in the scene and towards P , so there will always be at least one indirection between the light being emitted and reaching P , giving that component the name of *indirect* lighting. Figure 10 shows how much indirect illumination can contribute to the final image in more closed environments.

4.3 Sampling and Monte Carlo

The rendering equation presented in the previous sub-section is too complex to be solved analytically, except in simple scenarios. The approach taken in Computer Graphics has been to use Monte Carlo integration which relies on random sampling to evaluate different outcomes. As the number of samples taken increases, it will converge to the correct answer and do so at a rate which does not depend on the dimensionality of the integral being estimated. If applying the Monte Carlo integration to the integral I of function f over the domain D , the following is obtained with the Monte Carlo integrator positioned to the

Table 1: Visualising the impact made by using different PDFs when estimating the integral of the function f (see Equation (4)). For each interval, the given value represents the result of the Monte Carlo integration with the specified PDF using a single sample taken from that interval.

Interval	Expected value		
	Using p_1 (Eq. (5))	Using p_2 (Eq. (6))	Using optimal PDF
$x \in [0.00, 0.25[$	4	50	42
$x \in [0.25, 0.75[$	80	41.7	42
$x \in [0.75, 1.00[$	4	50	42

right of the limit:

$$I = \int_D f(x) dx = \lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{i=1}^n \frac{f(x_i)}{p(x_i)} \quad \text{with } x_i \in D \quad (3)$$

p is the *probability density function* and it gives the probability of obtaining a sample (e.g. x_i) within a specified domain (e.g. D) according to a selected distribution (e.g. uniform).

To limit the variance (i.e. how distant it is from the expected value) in the estimation, p should be as proportional as possible to f so that the ratio $f(x_i)/p(x_i)$ stays as constant as possible regardless of the sample x_i chosen; that constant value would be $1/I$. In the limit, if p was perfectly proportional to f , only a single sample would need to be taken to get the correct answer. The further away p deviates from f , the more samples will be needed to reach the answer, which is not ideal. Changing the probability distribution to distribute the samples in the places where they matter most, is called *importance sampling* and was used in Paper III; other methods exist for reducing variance.

The function f (Equation (4)) will be taken as an example to demonstrate importance sampling; it integrates to 42 over the interval $[0, 1]$. If a constant probability is used (i.e. p_1 , Equation (5)) the Monte Carlo integration will return widely different estimations depending on which sample was selected, as can be seen in Table 1. On the other hand, if a PDF is chosen that closely matches the original function, like p_2 (Equation (6)), the returned results are a lot closer to each other and to the expected value of 42.

$$f(x) = \begin{cases} 4 & \text{if } x \in [0, 0.25[\cup [0.75, 1[\\ 80 & \text{if } x \in [0.25, 0.75[\end{cases} \quad (4)$$

$$p_1(x) = 1.0 \quad \text{if } x \in [0, 1[\quad (5)$$

$$p_2(x) = \begin{cases} 0.08 & \text{if } x \in [0, 0.25[\cup [0.75, 1[\\ 1.92 & \text{if } x \in [0.25, 0.75[\end{cases} \quad (6)$$

Algorithms used to estimate the rendering equation can be qualified by different terms to highlight their behaviour. If an algorithm is *consistent*, it will converge to the expected

result as the number of samples gets closer to infinity; a Monte Carlo estimator is consistent. Additionally, the algorithm can be *unbiased* if there is no difference between its expected value and the expected value of the function it is estimating, or *biased* otherwise; an algorithm can be biased and still be consistent.

4.4 Tracing Rays From the Camera

The rendering equation formulates how to compute the amount of radiance travelling towards each pixel of the camera sensor, and Monte Carlo integration provides a way to evaluate this complex equation without needing an analytical solution to it. Now remains the task of performing the random experiments at the core of Monte Carlo algorithms, and averaging their results. This is where *path* tracing comes into play, but first *ray* tracing will be discussed as its precursor and the term most people have been hearing recently as it became hardware-accelerated on new GPUs.

Ray tracing was initially described by Arthur Appel in 1968 [4] to compute shading and shadows cast by objects from solid geometry. The version of ray tracing used today comes from the reformulation proposed by Turner Whitted in 1980 [55] that transformed ray tracing into a recursive approach. Instead of stopping at the first hit encounter, three new rays would be traced: a shadow, a reflective, and a refractive ray; the contribution of each ray is weighted based on material properties. However, this creates a tree of possibilities with an exponential number of rays traced per pixel, up to $1 + 3 \cdot 2^n$ with n being the number of hits encountered.

To alleviate that exponential growth, James Kajiya introduced *path tracing* in 1986 [24] alongside the rendering equation. Path tracing traces one or more *paths* per pixel, with a path corresponding to multiple ray segments in direct succession. This means that at each hit, a single ray will be generated for continuing the exploration. Rather than evaluating both the reflection and refraction similarly to ray tracing, path tracing will stochastically decide which one of the two to estimate. This selection could be done with equal probability, or based on material properties to provide better results (i.e. performing importance sampling).

The most common form of path tracing does not trace any shadow rays and instead relies on (hopefully) hitting a light source while propagating through the scene. As the probability for the paths to hit the light sources can be quite low, especially if the lights are small, the variance in the image will be very high. Most path tracing implementations today rely on the *next event estimator* (NEE) which allows for a direct sampling of the light sources from each hit, replacing the shadow ray from Whitted's ray tracing. This estimator is similar to the Monte Carlo integrator described in Equation (3), where the integral that needs to be computed is the one from line 2 of Equation (2).

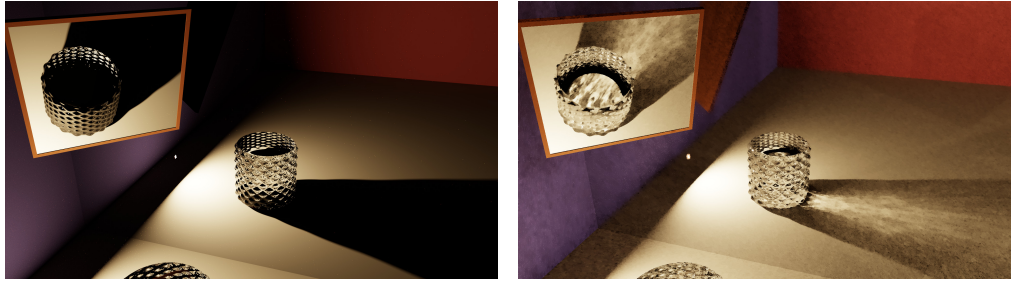


Figure 11: Highlighting the impact of caustics on a rendering. The left image is rendered with caustics disabled, while the rendering on the right has them enabled. Photon mapping and other “backward ray tracing” methods are better suited for rendering caustics than forward ones. The rendering on the right is taken from Paper v. The transparent glass object was made by Simon Wendsche, and is provided as one of the scenes for the 3rd edition of *Physically Based Rendering* [38].

4.5 Tracing Rays From the Light Sources

At the same time as path tracing was being presented, James Arvo introduced the idea of *backward ray tracing* [5], i.e. starting the ray tracing from the light sources rather than from the camera. The goal was to address certain limitations of ray tracing, such as diffuse inter-reflections (e.g. colour bleeding) as well as computing “direct” illumination from under transparent objects³. This approach was meant as a pre-process pass to ray tracing and not as a replacement, leading to a multi-pass algorithm. First, rays were traced from the light sources and the radiance they carried would be stored in textures at each diffuse surface intersected (except for the first one). Then, ray tracing would be performed and take into account the values stored in those textures.

Photon mapping [23] was proposed by Henrik Wann Jensen in 1996 as a more efficient way of rendering global illumination and caustics compared to existing photon-based approaches. Similar to backward ray tracing, the first pass starts from the light sources and traces light paths (rather than rays). At each intersection, instead of storing the radiance into a texture, a small structure called *photon* is stored into one of two maps: a caustic photon map if the photon is part of a caustic, or a global photon map. The algorithm uses photons for different things, such as estimating shadows and helping guide a path tracer towards important locations, but the focus here will be on using photons for estimating incoming radiance. To do so, a gathering step is performed to find the k nearest photons around the point where incoming radiance should be estimated, and the flux of each photon is combined and divided by the area over which they were gathered to estimate the incoming irradiance.

To accelerate the gathering of nearby photons in order to render images interactively, Stürzlinger and Bastos [43] used the rasterisation hardware of GPUs to “splat” the photons. As

³As the shadow ray is traced towards one of the light source and hits a transparent object, the shadow ray will be reflected or refracted, possibly multiple times. This will cause the ray to deviate from its original trajectory and most certainly fail to hit the light source, resulting in the shaded point being considered in shadow.

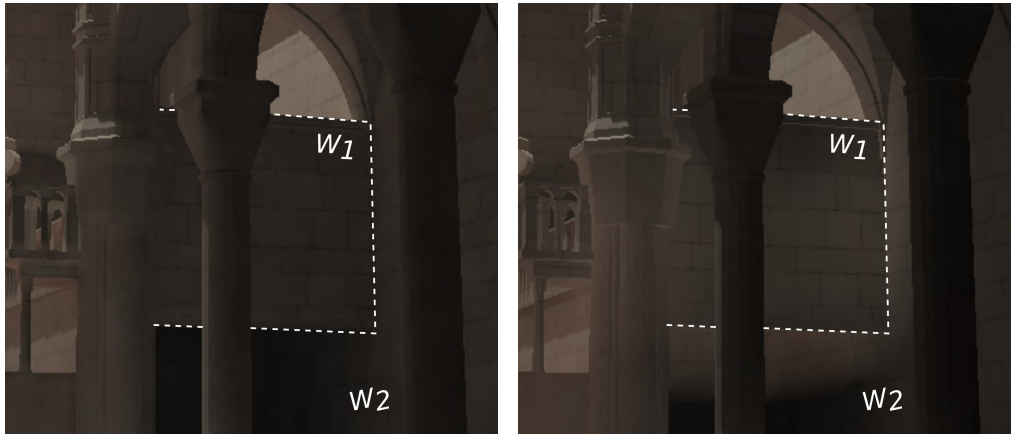


Figure 12: A highlight of light leakage due to too large photons. On the left a rendering with small enough photons that they do not leak through walls, which is not the case for the image on the right as the size of photons was increased. The main difference can be seen to the right and below the wall W_1 at the centre of the image: despite that wall connecting directly to the wall W_2 to its right, photons which landed behind W_1 still end up contributing some flux to W_2 in the image on the right. Close-ups from renderings taken from Paper 1. The Sibenik scene was created by Marko Dabrovic and is available under the CC-BY-NC license from Morgan McGuire's Computer Graphics Archive [32].

GPUs were not programmable at the time, gathering nearby photons on the GPU using a k -d tree was not an option. In photon splatting, a shaded point no longer gathers nearby photons but instead receives the contribution of a photon if the shaded point lies within the sphere of influence of the photon. The idea is then to render photons as spheres or disks on top of the scene, to let the rasteriser compute which pixels are covered by each photon and accumulate the contribution of such photons at each pixel. Using an analogy to describe photon splatting, this would be similar to painting an image on a wall by throwing small paint balls (the photons) at it. As the paint ball hits the wall, it will spread paint over a neighbourhood of the hit location. If an area gets hit by multiple balls, it will contain some combination of the paint from those balls.

Photon-based approaches like photon mapping are mostly associated to the rendering of caustics (see Figure 11), where they excel. They can also be used for estimating the indirect lighting in a way that is hard when tracing rays from the camera.

1. The tracing of the photons is not dependent on the position of the camera, meaning a single photon map can be reused for many different view points. If the photon map is reused between frames, it will work just fine even with large camera movements from frame to frame, unlike a screen-space-based temporal reuse which is commonly used when tracing rays from the camera.
2. From a single light path, one photon can contribute to one or more pixels while another might contribute to a completely different set of pixels, leading to one light

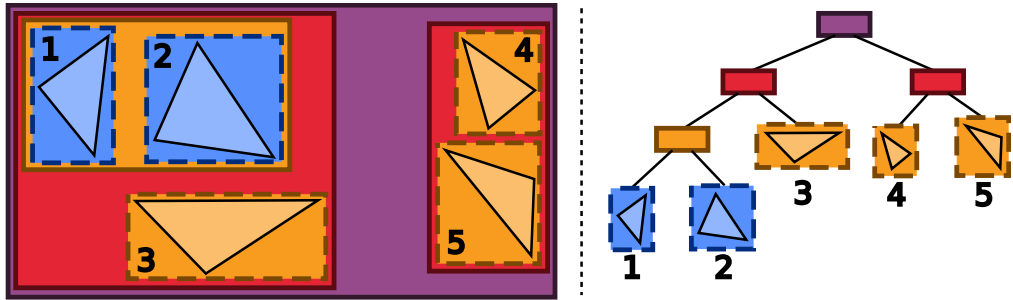


Figure 13: A visualisation of a binary BVH built around five triangles. On the left, the AABBs of each node of the BVH is represented around the triangles. On the right, a representation of the tree is shown. Leaf nodes have a dashed border, while internal nodes have a solid border; each colour corresponds to a different depth in the tree.

path potentially affecting multiple pixels unlike one camera path which will only affect the very pixel it was emitted from.

3. The tracing of the photons is also independent from the screen resolution. The cost of generating a photon map for 1080p or 4k will be the same, and only the gathering or splatting computational cost will change. This can be a large advantage over path tracing when long light-to-camera paths are involved.

Photons are not exempt from issues though, like not taking into account what is currently visible by the camera so they could end up impacting some other part of the scene currently hidden to the viewer, and if made too large, *leaking light* (i.e. contributing to surfaces it should not be able to reach, for example through walls as seen in Figure 12).

4.6 Acceleration Structures

Different types of computations performed to simulate light transport were presented in Sections 4.4 and 4.5. If, for example, a single ray is traced per pixel in a scene with a million triangles, at a resolution of 1080p, and each ray had to be tested against all triangles, it would end up requiring about $2 \cdot 10^{12}$ ray-triangle intersection tests. This would be expensive to compute, and would not scale to larger scenes or longer paths. Several acceleration data structures were developed over the years that significantly reduce the number of intersection tests to run. A single acceleration data structure will be covered in this thesis as it was used in several of the research projects, but there are many others used in Computer Graphics, some of them have a general usage while others are dedicated to a particular issue.

The *bounding volume hierarchy* (BVH) is a tree-like hierarchy where each node also stores a custom bounding volume containing the union of all its children's own bounding volume. Different volumes are used depending on the application, with the most common being AABBs and spheres; an example of a BVH using AABBs is shown in Figure 13. The BVH is

made of two different types of node: *internal* nodes (with a solid border) can have children but do not contain any geometry, while *leaf* nodes (with a dashed border) are the exact opposite (childless, but has the geometry); both types do have a bounding volume. Individual leaf nodes contain a variable, though capped, amount of primitives (usually triangles, but not always). On the other hand, the number of children an internal node has, is often fixed and hard-coded for efficiency reasons, and it will often be reflected in the name: a *binary* BVH when internal nodes only have two children, or of a *16-ary* (alternatively, *16-wide*) BVH if they instead have sixteen ones.

Traversing the tree can vary slightly depending on the application, but remains conceptually the same. Finding the closest triangle intersected by a ray will be the application showcased here, accelerated by the binary BVH visualised in Figure 13. First, the ray is tested against the bounding volume of the root node. If it misses, then it will not intersect any of the geometries found inside the tree either, so the traversal can stop right away and the miss shader (or equivalent) will be called. Otherwise, the ray will be tested against the bounding volume of each of the root node's children (i.e. the red nodes). For each intersected child, its own children will then be tested, and this will continue recursively until either no children get intersected, or a leaf node is intersected. In the latter case, for example leaf node number 4, the ray will then be tested against the individual triangles stored in that node.

5 Evaluation and Methodology

As new approaches are developed, they will be compared against existing ones to determine their pros and cons. In Computer Graphics, and especially in the real-time sub-area, the following aspects are usually evaluated: performance, output quality, memory utilisation, and generalisation. An explanation will be given in the next paragraphs on how those different evaluations are performed. Since not all research fields use the same research methodology, the one followed throughout this thesis will be presented later in this section.

Given the time-sensitivity of real-time rendering, the runtime performance of the algorithms used is really important, to the point of accepting some degradation of the image quality in exchange for lower frame times. As many different aspects can influence the timings, such as the hardware used (mostly the GPU) and its drivers, the BSDF model and its implementation, the path tracing implementation, extrapolating based on the performance results mentioned in previous work is not an easy task. This usually leads to the main competing previous work being re-implemented in the framework used by the new algorithm, which can be challenging as the original source code is not always available. Performance will be measured for different scenes, potentially different view points within the same scene, and if applicable, over a time period while the scene is being animated as some components can be impacted differently by the motion of objects. Out of the generated data, two different

measures are commonly presented: the total frame time (i.e. how long it takes to render an image from start to finish), and a breakdown of the frame time (i.e. how long each step in rendering the frame takes).

The usual trade-off to performance is image quality, which is harder to evaluate. On the one hand, it is possible to render a reference image \mathbf{R} and then compare the per-pixel difference with the current image \mathbf{I} using different measures such as *mean squared error* (MSE) $\frac{1}{n} \sum_{j \leftarrow 0}^n (\mathbf{R}[j] - \mathbf{I}[j])^2$ or *mean absolute error* (MAE) $\frac{1}{n} \sum_{j \leftarrow 0}^n |\mathbf{R}[j] - \mathbf{I}[j]|$, with n being the amount of pixels in \mathbf{R} and \mathbf{I} . Generating a single reference image can take from half a day up to several days, even when running on high-end GPUs, depending on the complexity of the scene and desired effects. On the other hand, not all of the differences are necessarily *perceived* by a user, while some errors with a lower magnitude might be perceived as more distracting than others with a higher magnitude. For real-time purposes, where time is of the essence, minimising the perceived error is more valuable and achievable than trying to match exactly a reference image. As a result, perceptual metrics were developed, such as *structural similarity* (SSIM) [54], or more recently FLIP [3] which also provides error maps to help visualise the location and intensity of the differences. However those approaches only look at images in isolation, ignoring the possible differences found between consecutive images. For example if we have two renderings of the same scene under the same conditions, maybe the error at pixel i in the first image is only -0.1 , and only 0.1 in the second image also at pixel i . Those values are low, but when going from the first image to the second one the difference is now 0.2 , and if the first and second images are alternating in a loop, pixel i will be perceived as blinking which can be a quite distracting artefact. These artefacts are known as *temporal instabilities* and can be mitigated using temporal filters. There are no standardised measures for the temporal stability of a sequence of images, and it ends up only being discussed in terms of how the authors found it distracting/annoying; a video is usually submitted alongside the paper, so that reviewers and readers can judge for themselves. Temporal measures are a difficult but active research area.

The two remaining aspects will be presented in the same paragraph. Even if memory is not as much of a concern as five to ten years ago, not all platforms have access to large amounts of VRAM, and whatever amount is present might already be used to store textures, geometry data, or acceleration structures. As such, limiting the memory footprint of a rendering algorithm is still desirable, and can have some positive impact on the performance as memory bandwidth is limited (see Section 3.1). To maximise their performance, some algorithms are tailored for rendering specific effects, or may rely on scene characteristics such as there only being diffuse surfaces for performing certain optimisations, for example. Therefore, it is important to know how well an approach can generalise to different materials being used, large amounts of light sources, etc., and it can be interesting to have a method that is a bit slower and/or uses more memory but handles more scenarios.

When working on the different research projects presented in the next section, the following

research methodology was used. First a problem is chosen based on the current research context and limitations of existing methods; the selection might be guided by ideas which occurred while working on a previous project. Then, a more detailed analysis of the state of the art regarding the selected problem is performed, to identify the main approaches which should be compared against, as well as their pros and cons. A first draft is made, specifying the aspects that should be improved by the new method, and which limitations are considered acceptable, as not all issues can necessarily be solved at once. The new algorithm is then worked on, alongside with its implementation to validate its behaviour and quickly discover any potential setbacks, generating a feedback cycle between designing the algorithm and implementing it. Once the new method has been finalised and validated by the results obtained when running its implementation, it is ready to be submitted to one of the different Computer Graphics conferences or journals.

6 Research Projects

The work discussed in this thesis focuses mainly on two different areas within the context of real-time rendering: evaluating direct lighting in the presence of thousands or even millions of light sources in a scene (Paper III and IV), and the use of photons for rendering caustics (Paper V) and on how to improve that usage (Paper I and II).

In the following sub-sections, an overview of the different papers will be given to help the reader understand the state of the art (at the time they were written), the approaches proposed and the problem(s) they were aiming to solve, and their pros and cons.

6.1 Direct lighting and the challenge of using thousands of light sources

Direct lighting has been around since the start of computer graphics, as the primary source of illumination in outdoor scenes and lit interiors, as well as being easier to compute than indirect lighting. The latter does not imply that it is an easy task, as can be attested by the amount of work spent on evaluating shadows, which remains an active research area to this day.

In this part, a different challenge of computing direct lighting will be presented: which light source to sample in scenes with a large number of lights. Depending on the scene, it can easily reach the tens of thousand, and evaluating all of their contributions to all surfaces of interest is not an option, even for offline rendering. Instead one or more lights are chosen, and only those will be used when computing the lighting; a single subset could be used for all pixels, or a different one could be created for each individual pixel. This selection process has a direct impact on performance and image quality, and will be discussed in more details

below.

Paper III — Importance Sampling of Many Lights on the GPU

As the amount of computations that can be performed per frame is usually too short for real-time applications, several techniques have been developed which rely on pre-computing (aka. *baking*) some of the data into textures: the time taken to look up during runtime that pre-computed data is a lot shorter than performing all the computations. One such method, first used in the video game *Quake* [1], consists in baking the light emitted by static light sources into an image called a *light map*. It has however some significant drawbacks: (1) it is expensive to compute and must be redone from scratch every time a light is modified, (2) it increases the memory footprint of the game both on the disk and at runtime, (3) it does not work well with glossy surfaces, and (4) it is completely static so toggling the status of a single light between on and off requires switching to a different light map. Hybrid approaches have been used in computer games, where all the lighting that will stay constant throughout a level will be baked, and individual light sources that can change dynamically will be evaluated at runtime.

If the scene has a high presence of glossy materials (which can not be easily pre-computed) or has many dynamic light sources, a different approach is needed. One common trade-off has been to (artificially) restrict the influence of each light source to a given volume, making it possible to disregard all lights located too far away [8, 36]. However, this can result in darker regions due to important lights not having been considered as they were deemed to be too distant. Additionally, the region of influence of each light needs to be set manually and carefully; Tokuyoshi and Harada [45, 46] suggest a way to stochastically compute the region of influence for each light. Finally, shadow maps can not necessarily be computed for all the selected lights, so only a sub-selection of them might actually cast shadows in the scene.

How to pick which lights to evaluate at a given surface has also been looked into for offline rendering. There, the focus is on consistent methods and taking into account more than just the distance to the light source or its intensity to estimate its importance. Multiple approaches have been tried like *lightcuts* by Walter et al. [52, 51], and the work from Vorba et al. [47], and Conty Estévez and Kulla [13]. The latter work builds a BVH around all *light sources* found in the scene; Bikker [8] also built a BVH, but instead around the *region of influence* of the light sources. Among the existing methods for offline rendering, the work by Conty Estévez and Kulla [13] looked promising for a potential use in a real-time rendering context thanks to relying mostly on a BVH, which has been a popular spatial acceleration data structure in that space for ray tracing usage. As the work presented in Paper III revolves around their algorithm, it will be described in more details in the next paragraph. Note that lightcuts also has the potential for being used in real-time, and this

was later done by Lin and Yuksel [30].

Conty Estévez and Kulla constructed a BVH around all light sources in the scene, supporting point and analytical light sources, as well as emissive meshes within the same framework. Note: we will refer to this type of BVH as a *light BVH*, to differentiate it from a BVH that is built around geometry and used to accelerate ray-primitive intersection test, which we will call *geometry BVH*. When building the tree, the *surface area heuristic* (SAH) [42] (applied to BVHs by Goldsmith and Salmon [17]) is traditionally used for deciding how to group primitives or nodes together in an optimal way for accelerating ray-triangle intersection tests. However since here the tree is built around light sources, they introduced a variation of SAH, called *surface area orientation heuristic* (SAOH), which also takes the orientation of the light source into account and its intensity. The traversal of that BVH is analogous to the one described in Section 4.6, but bears some differences which will be highlighted. Instead of testing for an intersection when looking at all children in a node, the importance of each child is computed before a single one of them is stochastically selected. The traversal continues recursively for that selected child, until a leaf node is reached. The authors also presented an adaptive split when traversing, which we ignored as potentially too expensive for real-time applications. There again, one triangle will be stochastically sampled based on its importance. That importance function takes into account (1) the distance between the node/light and the surface being shaded, (2) the intensity of the node/light, (3) its orientation, and (4) the orientation of the surface being shaded.

The work presented in Paper III focuses on evaluating Conty Estévez' and Kulla's approach in a real-time environment, to demonstrate its potential and give some recommendation about its usage. As such, different construction setups and their impact on the ray tracing performance was investigated, and similarly how the different terms in the importance function affected the results. On the construction side, the following were tested: (1) SAH versus SAOH, (2) using the volume of a node instead of its surface area, (3) the amount of bins⁴ used and number of axes considered [48], and (4) the number of triangles per leaf node.

Overall, the SAOH did not affect runtime performance but did improve the image quality compared to SAH, and while having all terms for the importance function did increase the render time noticeably, it increased the image quality even more. It is a clear improvement over more naive but consistent approaches that are suitable for real-time. Compared to the light maps mentioned earlier, it is more computationally expensive at run time, but it supports glossy surfaces better, can be used in dynamic scenes (as long as the light sources themselves are static), does not require long baking times, and does not increase the amount

⁴When building a BVH, considering all the triangles when performing the different computations would be too computationally expensive. Instead, the space containing all the triangles is partitioned into multiple parts called *bins* [48], and each triangle is associated to a single bin. Computations are then performed on the bins.

of space taken on the disk. However its high base frame cost makes it unsuitable for high refresh rates (120 Hz and above) for now. The approach could be applied to scenes with dynamic light sources by keeping the same topology and just updating the bounds (aka. *refitting*), similar to approaches for accelerating ray-triangle intersection tests [28, 50]. Building the acceleration structure on the GPU would significantly reduce its high computation cost, further helping with dynamic scenes.

Paper iv — Dynamic Many-Light Sampling for Real-Time Ray Tracing

Dynamic light sources were briefly mentioned in Paper III, but had not been studied. In this follow-up work, they are the main topic. Most of the context described in Section 6.1 still applies to this work as most of the mentioned methods already took dynamic lights into account, and so will not be repeated here. However, there is some unrelated (to lighting evaluation) work on acceleration data structures which is directly relevant.

Regardless of the lack of optimisation of the BVH builder used in Paper III, the limiting factor for handling dynamic lights in that work, was the computation cost of re-building the acceleration data structure being too high. This had been a problem for geometry BVHs before, and had been partially solved by refitting as mentioned earlier. However, refitting comes with its own issues, the main one being that the *quality* of the BVH (i.e. how good it is at efficiently finding the intersections) can degrade dramatically over time if the geometry moves a lot. As a result, a two-level approach was proposed by Wald et al. [49]; a similar architecture is still being used today in modern ray tracing APIs such as DirectX Raytracing (DXR) or Vulkan Ray-tracing (VKRT). The main observations from that work, are that

1. once a BVH has been built around static geometries, there is no need to ever update it;
2. for certain types of animations, objects are not deformed so their topology and the relative location of their triangles remain the same;
3. some objects are duplicated many times with different transforms (aka. *instancing*) but the underlying geometry is still the same so a single BVH could do.

Following point (1), they build a BVH around all static geometry. Based on (2), each animated object receives its own BVH and is associated with a matrix representing its current transformation. For example if the object rotates, one can simply modify the matrix without having to touch the BVH. If one wants to instantiate an object (point (3)), each instance will have its own transform matrix, but they can all point to a single BVH for the underlying model. A generic way to handle all three cases is to have a pair containing a

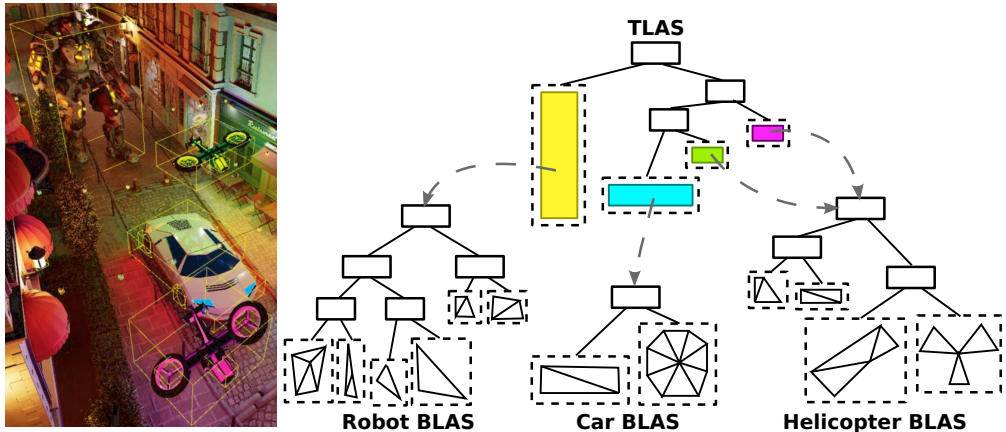


Figure 14: Visualising the two-level BVH (on the right) for the scene presented on the left; for simplicity, only the robot, car, and helicopters will be represented. The robot is static and as a result is in a separate BLAS than the car and helicopters, which are dynamic, avoiding the cost of updating the acceleration structure for such a big object. As two different instances of the same helicopter geometry are present in the scene, it needs to be placed in a different BLAS than the car so its BLAS can simply be referenced twice by the TLAS rather than duplicating the geometry of the helicopter twice to have the helicopter and car share a common BLAS. Internal nodes have thick, continuous, borders, while leaf nodes have dashed borders. Rendering (on the left) taken from Paper iv; the robot model was kindly donated by Epic Games, and credits for the other models used can be found in the Acknowledgements section.

transform matrix and a reference to a BVH: for example, one pair for the static geometry, one pair per animated object, and one pair per instance. Since that might result in hundreds of pairs that have to be considered when tracing the rays, one final BVH is built around those pairs to accelerate that search. The BVHs referenced in the pairs are called *bottom level acceleration structures* (BLAS), while the top BVH is the *top level acceleration structure* (TLAS); this nomenclature is used in both DXR and VKRT where the acceleration structure is not mandated to be a BVH and could be something else. A visual representation of a two-level BVH can be seen in Figure 14.

The two-level approach was chosen for being straightforward to implement, efficient, and commonly used for geometry BVHs, meaning that many are familiar with it, and there have been several follow-up works to improve on/extend it. Indeed, as shown in Paper iv and in earlier ones discussing two-level BVHs, it allows for nearly the same quality as a one-level BVH that is rebuilt from scratch every frame, for only a small overhead compared to just refitting the whole BVH. Refitting and two-level hierarchies were validated on light BVHs, and some differences between geometry and light BVHs, that are worth keeping in mind, were noticed. While it is recommended to have as few BLASes as possible for geometry BVHs⁵, this does not apply to light BVHs as no rays are actually traced against the BLASes and a single one of them will be selected and evaluated. Additionally, having a

⁵A ray has to be tested against all BLASes it intersects to find the closest intersection, which could be expensive if there are many. This might be the main reason why one is advised to have all static geometries within a single BLAS.

separate BLAS for different static lights could improve the quality of the generated TLAS, especially when static light sources are found all over the scene, resulting in an AABB that will encompass nearly the whole scene and most dynamic light sources. Another insight, absent from the paper, is that a light BVH might need an update even if the light is not transformed, unlike a geometry BVH. Indeed, that will be the case if the intensity of the light is animated (like a pulsating LED indicating that a laptop is sleeping) or the spread angle of a spotlight, as the SAOH cost function used during the build is dependent on those parameters. That is probably not an issue if a handful of BLASes are affected. If it is predominant, it might be worth considering using SAH when building those BLASes to avoid the need for an update, though at the cost of some lower BVH quality.

This architecture allowed for unbiased direct lighting evaluation from over five thousand dynamic light sources in fully dynamic scenes in real-time, with temporally stable results thanks to the SVGF spatio-temporal filter by Schied et al. [40]. Even with the two-level approach, the light BVH can still also be used to improve direct light estimation at secondary bounces with the same efficacy. This work spearheaded research in the area which resulted in NVIDIA's RTX Direct Illumination [35]. The main limitation with the current approach is not taking visibility into account when computing the importance of a light source. This could result in the algorithm often (due to its high importance) selecting a particular light source, only to realise after tracing the shadow rays that it is occluded, even if the same situation already occurred a few frames ago.

6.2 Using photons, for global illumination and caustics

Evaluating the light transport by starting from the light source and emitting photons is not always the most efficient approach, but there are situations where it is, as illustrated in Section 4.5. Those situations where photons are beneficial, have usually not received the same attention as situations where ray or path tracing works best. For one, the focus was on effects affecting most scenes, and to a larger degree. Additionally those effects can be handled using a single set of related techniques, making it appealing to reduce their computational cost and make them more real-time friendly. Finally, using the resources available at the time was prioritised, and rasterisation was not well adapted for effects requiring more than one bounce from a light source.

However that should not be a deterrent to working on those topics and trying to make them more prevalent and accessible. In this part, two different optimisations when rendering with photons will be described, followed by a method allowing for caustics that are indirectly visible to finally be rendered in real-time.

Paper 1 — Photon Splatting Using a View-Sample Cluster Hierarchy

One popular technique in computer games to approximate dynamic indirect lighting has been the use of *virtual point lights* (VPLs) [26]. From the perspective of a surface that needs to be shaded, they will act as a regular point light. However, rather than emitting a radiance pre-defined by an artist, they will instead capture the radiance flowing towards them. Using rasterisation, it is relatively cheap to compute the amount of direct illumination reflected off nearby surfaces and store that in a VPL, allowing for a one-bounce indirect illumination. More recent approaches have been tracing rays from VPLs to store multiple bounces of indirect illumination, with the idea that, for example, tracing 100 rays from a VPL which is then used by a 1,000 pixels, will be less computationally expensive than tracing a single ray per pixel (100 rays versus 1,000) and have a higher image quality than only tracing 1 ray every 10 pixels and using a spatial filter to share the results between neighbouring pixels. These methods are however not without issues. For example, their placement in the scene is crucial for providing good visual results without wasting resources, but positioning them manually is not ideal while doing it automatically is not easy and has been a research area of its own. Additional methods exist for estimating indirect lighting which will not be presented here, but are discussed in the paper.

Apart from the generation of the photon map, the major computational cost each frame is the association of photons and pixels. Reducing that cost is the main focus of Paper 1. As mentioned in Section 4.5, the gathering of photons is not well suited to being accelerated on GPUs, leading to splatting algorithms. Mara et al. [31] evaluated various existing splatting approaches as well as presenting new ones. The most efficient approach described partitions the screen into several *tiles*, and for each, computes the bounding volume around all visible surfaces within that tile. When testing whether a photon contributes to a pixel, it can first be tested against the bounding volume of the corresponding tile and discarded if it fails, quickly reducing the amount of photons considered by each pixel. However, in the case of large depth discontinuities, this method will fail to cull some of the photons that lie in between various pixels without contributing to any of them, as seen in Figure 15a.

To avoid that shortcoming, a different acceleration data structure was used, which was first presented by Olsson et al. [36] for light sources, and later built around view samples (but for a different application) by Sintorn et al. [41]. The acceleration data structure partitions the whole view frustum, which is 3-dimensional, rather than just the screen which is 2-dimensional; each 3-d partition is called a *cluster*. Now, in case of depth discontinuity, view samples of a single 2-d tile can end up in different partitions, resulting in tighter bounds (see Figure 15b for such an example). As this can lead to too many clusters to iterate upon, a BVH-like hierarchy is built around them, which will then be traversed by all the photons. Having a hierarchy allowed for additional optimisation opportunities, such as tracking the orientation of the surfaces within each node and discard early on photons landing below

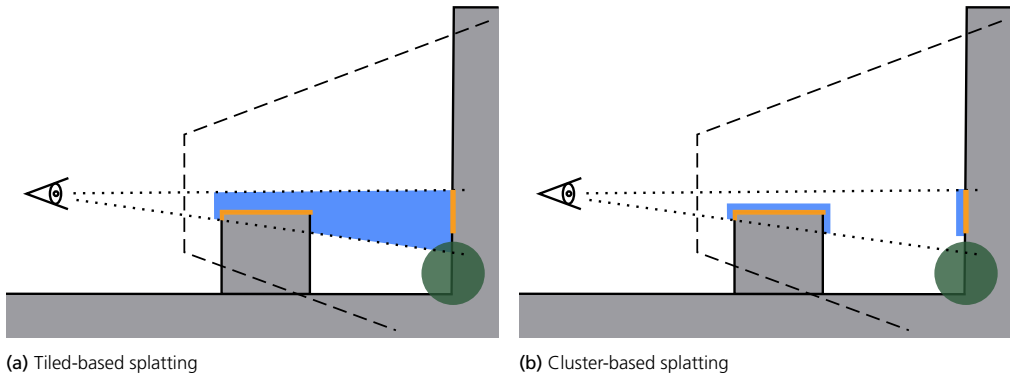


Figure 15: Visualising two different photon splatting techniques in the presence of depth discontinuities. This is a 2-d side view, with the eye on the left representing the position of the camera, and the long dashes showing the outlines of the view frustum. The photon, represented as a green sphere, intersects the bounding volume (in blue) of the tile, despite not contributing to any of the view samples (highlighted in orange).

the horizon of the surfaces, or if a photon affects a whole node consisting of diffuse surfaces, accumulate its contribution directly at the node level without traversing further down. In a similar spirit to the early accumulation, and to further reduce the computational cost of each frame, each cluster can have its set of incoming directions partitioned and photons will directly contribute to the corresponding partition. When shading using this approximate method, each pixel will perform a weighted combination of those partitions, rather than having to go through a list of photons.

By culling photons more effectively, only a small portion of the photons considered during shading were rejected. While the culling is more computationally expensive than previous approaches, this is completely offset by the total frame time being reduced thanks to the higher efficiency, and without affecting the image quality. The introduced approximation is able to further lower the total computational cost of a frame and provide a near-constant frame rate, for a minimal image quality degradation. The presented algorithms, as well as existing ones, are suitable for few and large photons, but do not translate as well to many and small ones. This is not an issue for approximating diffuse global illumination, but will be for higher frequency phenomena like caustics. Due to how the storage of photons sent for shading is handled by our implementation, the splatting is being run twice, nearly doubling its computational cost.

Paper II — Path Verification for Dynamic Indirect Illumination

As part of the research in Paper I, we realised that one of the main time-consuming tasks had become the tracing of the photons into the scene, especially when long paths (3+ bounces) were involved. As many rays are needed in order to produce a relatively converged image, there had been several approaches for reusing previous results. Bekaert et al. [7] proposed

sharing traced paths between neighbouring pixels, allowing a single pixel to increase its sample count for a low additional computational cost and without introducing any bias. This works well for path tracing, but not for photon tracing as it would not lead to any additional photons being created nor additional information from the point of view of the camera. In the context of area light sources, a similar sharing would not be as easy. Dmitriev et al. [14] presented a different approach, suited for photon tracing. They have two different “types” of photons: *pilot photons* which are uniformly sampled and traced into the scene, and *corrective photons* that are generated based on the periodicity of the Halton sequence of their corresponding pilot and as a result will explore the space around their pilot. Photons are partitioned into multiple groups, each group consisting of photons from both types, and having its own importance. Pilot photons are traced first, and if they intersect a dynamic object or the *phantom* of one of them (i.e. a dynamic object was located there in the previous frame but has now moved somewhere else), the importance of the corresponding group will be increased. This importance is then used to prioritise which photons will be traced in the next frame, and ensures that larger changes will be tackled first in case there are too many updates to be processed during a fixed time. However not all pieces of that algorithm can be efficiently transposed to the GPU, which is a necessity today for rendering algorithms to run in real-time.

Our approach was to start by keeping all the light paths traced in the previous frame. First, the initial segment of each light path is validated against the light source which emitted it, to ensure it could still have been generated if emitted in the current frame. To detect changes in the scene, each segment of a light path is tested, in order, against the AABBs for all moving objects. If a segment intersects such an AABB, the rest of the light path is invalidated and this segment will be scheduled for being re-traced. However if light sources move, their light distribution will no longer be correct, as shown in Figure 16. To avoid that, each light has two different buffers associated to it: one describing how many light paths should be emitted per area and set of outgoing directions, while the others tracks what the actual situation is. If a cell is found to emit too few light paths, the missing amount will be scheduled to be generated during the next tracing event, while in cells with too many light paths, each path will be stochastically ended in a way to reach the expected amount; this is actually performed before testing against the dynamic objects. For diffuse surfaces, a slight change in incoming direction will only have a small impact on the outgoing radiance; that information can be used to increase the amount of segments reused. When a dynamic object is hit, instead of invalidating the rest of the path right away, the new intersection point is computed and the visibility between that new position and the next segment of the path is validated. If it is, and the radiance reflected by this new position is similar to the amount reflected in the previous frame, the rest of the path will be kept as-is (assuming it passes the other validations).

Our method was able to reuse a large amount of rays from frame to frame, even as light

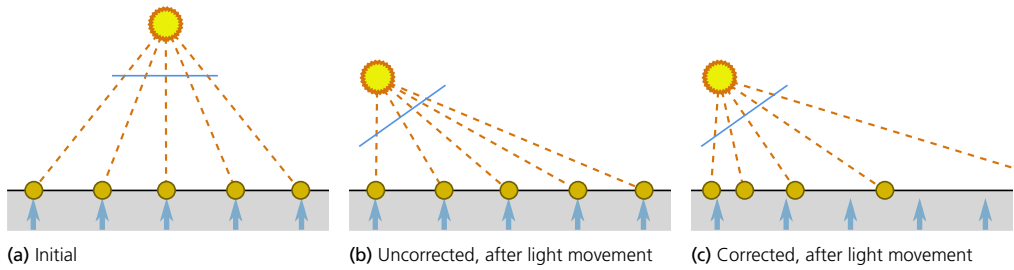


Figure 16: Visualising the distribution of photons as the light moves. The photons, represented as orange spheres, are traced uniformly from the light source, and their initial position is marked by a blue arrow. If the light paths are kept as-is as the light rotates (middle), the uniform distribution is lost and the lighting will not reflect the current orientation of the light source. The right version shows how a uniform distribution from the new orientation would look like, for comparison.

sources moved and changed orientation, resulting in corresponding reductions in tracing time. When a light was shining directly onto a diffuse moving object, our error-based measure allowed for most intersecting light paths to be preserved rather than invalidated. A side benefit from this reuse, is the improved temporal coherency which is especially important when the photon density is not high enough to provide a decent coverage. Diffuse global illumination with up to six bounces, in a dynamic scene, was possible at interactive frame rate. This algorithm will work with glossy surfaces, though the smoother the surface is, the fewer segments can be reused thanks to the error-based validation as even small changes on the inputs will result in moderate to high changes on the outputs. The tracking done on a per-light basis to maintain their distribution works well for a few light sources, but does not scale to a hundred light sources, let alone the tens of thousands from the many-light work (Section 6.1).

Paper v — Real-Time Rendering of Indirectly Visible Caustics

The rendering of caustics in a real-time context was severely constrained due to being expensive to compute. Indeed, it is a multi-bounce phenomenon, and is most efficiently evaluated starting from the light source. As a result, there is no guarantee that only results useful for the camera will be generated, nor that the work can be reused easily by other parts. Some of the first interactive renderings of caustics did not use ray tracing, relying instead on other approximations, and were limited to single-reflection caustics [53] (i.e. light is emitted, then reflected off a specular surface, and finally creates a caustic at the very next intersection), or caustics from entering and exiting a single refractive object [56]. Ray-traced approaches were used, lifting some of the restrictions, such as McGuire and Luebke [33] ray tracing on the CPU, or more recently, with the recent release of consumer GPU hardware accelerating ray tracing, the work from Kim [27] and Yang and Ouyang [58]. However, the trade-off for achieving real-time performance has been to accumulate the photons in screen

space thereby foregoing caustics that are not *directly* visible from the camera. As mentioned in Section 2.4, screen space corresponds to the 2-d grid of pixels, and for each pixel, the data available corresponds to the closest surface to the camera. If the camera captures a mirror in which the reflection of a caustic can be seen, accumulating photons in screen space means accumulating on the mirror itself, which is problematic for two different reasons. The first one, is that the photon needs to be reflected towards the mirror right after generating the caustic, in order for the caustic to be visible via the mirror. Unfortunately, the surface onto which a caustic is formed, is a diffuse surface, so the photon only has a small chance of being reflected towards the mirror and not somewhere else. The second reason, is that even if the photon reached the mirror, it would have to come from a specific direction (or a limited set of directions) for it to be reflected towards the camera, as a mirror is a specular surface (or highly glossy, if slightly rough).

In order to render caustics visible via mirrors or behind transparent objects, a different accumulation approach is required. *Progressive photon mapping* (PPM) [19], which is an evolution of photon mapping, starts by tracing rays from the camera and into the scene, until a diffuse surface is hit, marking the location where photons will be accumulated, and avoiding the same issues as screen-space accumulation. To allow for fast queries of those accumulation locations, which we call *collection points*, Hachisuka et al. used an acceleration that does not translate well to GPUs. Instead, we rely on a BVH built around those areas, as proposed by Evangelou et al. [15] which has the advantage of using GPUs.

Our approach can be summarised as follows. Similar to PPM, the first *diffuse* surface encountered when tracing camera paths is queried, and marks where photons will be accumulated over an area dependent on the properties of surfaces hit along the way. The collection points are then placed inside a BVH, to accelerate their lookup. Finally, paths are traced from the light sources to simulate the caustics, and each time a caustic-generating photon hits a surface, it will contribute its energy to all collection point containing that hit point. The accumulated contributions can be reused from frame to frame by averaging the values stored in nearby collection points of the previous frame.

The proposed method supports rendering caustics, regardless of how many specular bounces are found between the camera and the caustic, be it one for a mirror, two for simple transparent objects, or more. This is achieved in real-time, thanks to the hardware acceleration, even when taking multiple reflections into account. As the accumulated data are stored in world space, their reuse from one frame to the next does not require re-projecting results using motion vectors, providing several benefits:

- no reliance on motion vectors especially as those are not readily available for specular surfaces such as the mirrors used in our work;
- a caustic can be reused even as the camera path leading to it changes.

While the approach renders in real-time, it remains too expensive for applications such as computer games. One of the reasons is the computational cost of rebuilding from scratch the BVH around collection points every frame, while another is the overhead of querying and accumulating at the collection points while tracing the photons.

7 Contributions

In the previous section, the context in which the different projects took place as well as the algorithms themselves were described. As a follow-up, the different contributions made in those projects will now be summarised.

By decreasing the number of photons considered during shading, the frame cost could be lowered in Paper I without degrading the image quality. This was made possible by using a different acceleration data structure, which is more efficient at discarding photons not contributing to the final image. We also showed how that acceleration structure could be easily extended for higher culling efficiency, or for improved performance by introducing some approximations. In the latter case, an algorithm which trades off some image quality for lower frame time was presented, with the added benefit of its computational cost being less sensitive to the amount of photons processed, and therefore providing a nearly constant frame rate experience.

In Paper II, a validation scheme for reusing light paths from a previous frame in the current one was introduced. This scheme has several benefits: among others, it minimises the number of rays being traced, and supports fully dynamic scenes. Additionally, we demonstrated how the validation scheme can exchange some image quality for increased reuse of light segments interacting with dynamic objects; the amount of error tolerated is configurable by the user. The end result is the ability to render dynamic global illumination at interactive frame rates, with longer light paths than would be supported by existing methods.

To improve the rendering in real time of scenes with many light sources, an acceleration data structure and sampling scheme used in offline rendering was successfully adapted to real-time in Paper III. Thanks to that structure, not only does it provide unbiased evaluation of direct illumination at primary hits, but it also does so at secondary ones which ameliorates the quality of indirect lighting. Since many different parameters can be tuned during the construction and traversal of the acceleration data structure, an evaluation of their impact on performance cost and image quality was performed, to help adopters in deciding which settings to use based on their needs.

The work presented in Paper III was extended to dynamic scenes in Paper IV. We first showed that the acceleration data structure could be refitted on the GPU at a low cost to accommodate moderate amounts of light motion in a scene. Then, by breaking up that

single acceleration structure into multiple ones that are local to groups of light sources, only the relevant subsets would be updated. A two-level hierarchy is introduced, with a top-level acceleration structure built atop the aforementioned local acceleration structures. The top-level acceleration structure can be asynchronously updated, allowing for a large range of movements by light sources, and so at a low computation cost and with limited sampling quality degradation.

In Paper v, two existing algorithms were combined and adapted to enable the rendering of indirectly visible caustics in real time. Thanks to the acceleration data structure used, built around collection locations unlike the ones used in Paper III and IV which were built around light sources, collection points can have their size dynamically set based on their ray differentials for higher image quality. To improve the temporal stability and the image quality of each rendered image, a method reusing previous computations was proposed that does not require motion vectors (which are usually unavailable in the areas of interest) and adds almost no computational overhead to the main algorithm.

In summary, we introduced algorithms for handling direct lighting from millions of potentially dynamic light sources in an unbiased way, giving artists additional freedom when lighting a scene. Caustics can now be rendered even when indirectly visible, allowing for caustics to be more present in scenes without risking to being displayed inconsistently. Since enabling new effects is not enough for reaching parity in real-time rendering with offline rendering, optimisations for current effects were also looked into, in order to lower their computational cost and make them more accessible. We hope to see those effects being included in real-time applications in the near future, and that our different algorithms will inspire new research in the community.

8 Conclusion and Looking Forward

In order to further improve the rendering quality and the realism in real-time applications, the work presented in this thesis covers both algorithmic optimisations and new approaches for improving the visual fidelity. For optimisations, we proposed a more computationally efficient way for associating a photon to a pixel as well as for processing their contribution to said pixel. We also looked into which light rays can be kept from one frame to the next with minimal impact on the image quality, freeing some resources from having to re-compute them every frame. As for the simulation of previously ignored visual effects, a different caching mechanism allows for caustics that are indirectly visible to be rendered in real time. Thanks to a careful selection of the lights used for evaluating direct lighting, dynamic scenes lit by many light sources can be enjoyed at 30 FPS or more.

In light of recent advances and changes in the area, it is interesting to reflect back on the

different research projects described in this thesis. Looking first at the papers tackling direct lighting, there have been some major developments since those papers were published. Lin and Yuksel [30] achieved higher image quality thanks to a faster method that allows them to consider more samples in a given time budget. However, they do not take the BSDF into account, resulting in poor results for glossy surfaces. The introduction of ReSTIR [9], and its refinement [57], further improved the quality while also reducing its computational cost, thanks in part to not having to maintain an acceleration data structure. The price paid is being restricted to evaluating direct lighting at only the first bounce (partly alleviated by Boksansky et al. [11]), and having difficulties with very localised light sources. A lower cost version of the approach presented in Paper IV could complement the work of Wyman and Pantelev [57] to lift those restrictions, while still taking the BSDF into account that Lin and Yuksel [30] were missing. The algorithms presented in Paper I were fast at processing the photons, but were limited to accumulating at the very first hit from the camera. With the advent of techniques relying on the ray tracing acceleration hardware [15], it seems unlikely our photon accumulation will be useful outside of running on older GPUs that do not have that custom hardware. Regarding photon-tracing efficiency, most of the recent work [27, 58] have focused on guiding the light paths towards where they will contribute to the final image. As a result, the validation scheme proposed in Paper II is still interesting, though a re-evaluation on newer hardware will be necessary to measure if the overhead for reusing segments is still worth it when ray tracing has become cheaper. That scheme is incompatible with many-light rendering, but so are current real-time caustics rendering algorithms as they perform most of their tracking on a per-light source basis. To be adopted by real-time applications, a lower computational cost will be necessary for our approach to rendering indirectly-visible caustics (Paper V). To achieve that, the amount of overlap between different collection points would have to be reduced, and one should be able to update the BVH around those collection points more efficiently. It would also be interesting to apply that approach to other effects, like direct lighting on a surface visible via a mirror.

During the time frame of my Ph.D. studies, the field of real-time rendering has seen some major changes. As in many other fields, machine learning has been expanding rapidly and been applied to many different aspects of rendering. One of its first applications (in rendering) has been denoising, where machine learning-based methods quickly became among the best and widely used, even in computer games, as exemplified by NVIDIA's *Deep Learning Super Sampling* (DLSS) [34]. It has now been applied to many other problems, like rendering simplification [20], or even whole scene rendering based on just the scene graph [18]. This push resulted in GPUs from some vendors containing dedicated hardware to accelerate those types of workflows. Another major change, which had a direct impact on our work, was the release by NVIDIA of consumer-grade GPUs with dedicated hardware for accelerating ray tracing, providing up to an order of magnitude speed-up when tracing rays. As a result, there was a renewed focus into using ray tracing for real-time applications, as well as hybrid approaches to keep rasterisation where it performs best and use ray tracing

where it does not. Several computer games started using ray tracing to enhance the rendering realism with physically-correct reflections (*Battlefield v*), soft shadows (*Shadow of the Tomb Raider*), global illumination (*Metro Exodus*, *Control*). Since the initial release, additional GPU manufacturers have shipped or announced GPUs with hardware support for accelerating ray tracing, such as AMD (already shipping), Intel (shipping soon), and ARM (announced).

Even if some recently released updates or games were fully path-traced, such as *Q2VKPT* or *Minecraft*, there are still many challenges left before seeing it applied to more complex environments of AAA computer games. For example, large open games need to stream models in and out depending on where the player is, which can be problematic for ray tracing acceleration structures. Also, the large openness will affect image quality as the amount of encountered objects will increase with the distance, leading to higher variance. Quality metrics are currently missing to help decide between updating and rebuilding an acceleration structure. On the rendering side, direct lighting from many light sources has improved considerably recently, however very localised light sources are problematic to the current state of the art [57]. Participating media (such as smoke) and other volumetric effects are commonly used in some computer games, but ray-traced support remains too expensive for such applications even if some progress was recently made [22, 21, 29]. Perfectly reflective and refractive objects can be easily handled by ray tracing and quickly produce noise-free images, but as the materials become rougher it takes longer for results to converge. Besides those challenges, I think research in real-time rendering will slowly increase its targeted frame rate to 60 FPS and adopt *high dynamic range* (HDR). It will be fascinating to see the evolution to a more ray tracing-based workflow for real-time rendering, as new algorithms are developed and hardware-accelerated ray tracing GPUs become more widespread.

9 References

- [1] ABRASH, M. *Quake's Lighting Model*. Coriolis Group Books, Scottsdale, AZ, USA, 1997, ch. 68, p. 1244–1256.
- [2] AMAZON LUMBERYARD. Amazon Lumberyard bistro, open research content archive (ORCA). <http://developer.nvidia.com/orca/amazon-lumberyard-bistro>, Jul. 2017. Accessed: 2021-11-03.
- [3] ANDERSSON, P., NILSSON, J., AKENINE-MÖLLER, T., OSKARSSON, M., ÅSTRÖM, K., AND FAIRCHILD, M. D. FLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 2 (Aug. 2020), 15:1–15:23.

- [4] APPEL, A. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring)* (New York, NY, USA, 1968), Association for Computing Machinery, p. 37–45.
- [5] ARVO, J. Backward ray tracing. In *Developments in Ray Tracing, Computer Graphics, Proceedings of the ACM SIGGRAPH 86 Course Notes* (New York, NY, USA, 1986), vol. 12, Association for Computing Machinery, p. 259–263.
- [6] BARTEKS2. Car low poly concept 3D. <https://www.turbosquid.com/3d-models/concept-car-3d-1177980>, Jul. 2017. Accessed: 2021-11-03.
- [7] BEKAERT, P., SBERT, M., AND HALTON, J. Accelerating path tracing by re-using paths. In *Proceedings of the 13th Eurographics Workshop on Rendering* (Goslar, DEU, 2002), P. Debevec and S. Gibson, Eds., EGRW '02, The Eurographics Association, p. 125–134.
- [8] BIKKER, J. *Ray Tracing in Real-time Games*. PhD thesis, Delft University, 2012.
- [9] BITTERLI, B., WYMAN, C., PHARR, M., SHIRLEY, P., LEFOHN, A., AND JAROSZ, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics* 39, 4 (2020), 148:1–148:17.
- [10] BLENDER FOUNDATION. Blender. <https://www.blender.org/>. Accessed: 2021-12-10.
- [11] BOKSANSKY, J., JUKARAINEN, P., AND WYMAN, C. Rendering many lights with grid-based reservoirs. In *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*, A. Marrs, P. Shirley, and I. Wald, Eds. Apress, Berkeley, CA, USA, 2021, p. 351–365.
- [12] CATMULL, E. E. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Dec. 1974.
- [13] CONTY ESTÉVEZ, A., AND KULLA, C. Importance sampling of many lights with adaptive tree splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018), 25:1–25:17.
- [14] DMITRIEV, K., BRABEC, S., MYZKOWSKI, K., AND SEIDEL, H.-P. Interactive global illumination using selective photon tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering* (Goslar, DEU, 2002), P. Debevec and S. Gibson, Eds., EGRW '02, The Eurographics Association, p. 25–36.
- [15] EVANGELOU, I., PAPAIOANNOU, G., VARDIS, K., AND VASILAKIS, A. A. Fast radius search exploiting ray tracing frameworks. *Journal of Computer Graphics Techniques (JCGT)* 10, 1 (2021), 25–48.

- [16] F3NIX. Helicopter concept low-poly. <https://sketchfab.com/3d-models/helicopter-concept-low-poly-359bd29b2a074562865c4acb953385f6>, Feb. 2018. Accessed: 2021-11-03.
- [17] GOLDSMITH, J., AND SALMON, J. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7, 5 (1987), 14–20.
- [18] GRANSKOG, J., SCHNABEL, T. N., ROUSSELLE, F., AND NOVÁK, J. Neural scene graph rendering. *ACM Transactions on Graphics* 10, 4 (2021), 1–11.
- [19] HACHISUKA, T., OGAKI, S., AND JENSEN, H. W. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 Papers* (New York, NY, USA, 2008), J. C. Hart, Ed., SIGGRAPH Asia '08, Association for Computing Machinery.
- [20] HASSELGREN, J., MUNKBERG, J., LEHTINEN, J., AITTALA, M., AND LAINE, S. Appearance-driven automatic 3D model simplification. *Computing Research Repository (CoRR) abs/2104.03989* (2021).
- [21] HOFMANN, N., AND EVANS, A. Efficient unbiased volume path tracing on the GPU. In *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*, A. Marrs, P. Shirley, and I. Wald, Eds. Apress, Berkeley, CA, USA, 2021, p. 699–711.
- [22] HOFMANN, N., HASSELGREN, J., CLARBERG, P., AND MUNKBERG, J. Interactive path tracing and reconstruction of sparse volumes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 4, 1 (2021), 1–19.
- [23] JENSEN, H. W. Global illumination using photon maps. In *Rendering Techniques '96* (Vienna, AUT, 1996), X. Pueyo and P. Schröder, Eds., Eurographics, Springer, Vienna, p. 21–30.
- [24] KAJIYA, J. T. The rendering equation. *SIGGRAPH Computer Graphics* 20, 4 (1986), 143–150.
- [25] KALWEIT, S., CLARBERG, P., KOLB, C., YAO, K.-H., FOLEY, T., WU, L., CHEN, L., AKENINE-MOLLER, T., WYMAN, C., CRASSIN, C., AND BENTY, N. The Falcor rendering framework. <https://github.com/NVIDIAGameWorks/Falcor>, Aug. 2021. Accessed: 2021-12-10.
- [26] KELLER, A. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., p. 49–56.
- [27] KIM, H. Caustics using screen-space photon mapping. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, E. Haines and T. Akenine-Möller, Eds. Apress, Berkeley, CA, USA, 2019, p. 543–555.

- [28] LAUTERBACH, C., YOON, S.-E., MANOCHA, D., AND TUFT, D. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *2006 IEEE Symposium on Interactive Ray Tracing* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, p. 39–46.
- [29] LIN, D., WYMAN, C., AND YUKSEL, C. Fast volume rendering with spatiotemporal reservoir resampling. *ACM Transactions on Graphics* 40, 6 (2021), 278:1–278:18.
- [30] LIN, D., AND YUKSEL, C. Real-time stochastic lightcuts. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 1 (2020).
- [31] MARA, M., LUEBKE, D., AND MCGUIRE, M. Toward practical real-time photon mapping: Efficient GPU density estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2013), M. Garland and R. Wang, Eds., I3D '13, Association for Computing Machinery, p. 71–78.
- [32] MCGUIRE, M. Computer graphics archive. <https://casual-effects.com/data>, Jul. 2017. Accessed: 2021-11-12.
- [33] MCGUIRE, M., AND LUEBKE, D. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), D. McAllister, M. Pharr, and I. Wald, Eds., HPG '09, Association for Computing Machinery, p. 77–89.
- [34] NVIDIA. Deep learning super sampling (DLSS). <https://developer.nvidia.com/dlss>, 2018. Accessed: 2021-11-03.
- [35] NVIDIA. RTX direct illumination (RTXDI). <https://developer.nvidia.com/rtxdi>, 2020. Accessed: 2021-10-31.
- [36] OLSSON, O., BILLETER, M., AND ASSARSSON, U. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Goslar, DEU, 2012), C. Dachsbacher, J. Munkberg, and J. Pantaleoni, Eds., EGGH-HPG'12, The Eurographics Association, p. 87–96.
- [37] PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., McALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics* 29, 4 (2010), 66:1–66:13.
- [38] PHARR, M., JAKOB, W., AND HUMPHREYS, G. *Physically Based Rendering*, third ed. Morgan Kaufmann, Boston, MA, USA, 2016.
- [39] SAITO, T., AND TAKAHASHI, T. Comprehensive rendering of 3-D shapes. *SIGGRAPH Computer Graphics* 24, 4 (1990), 197–206.

- [40] SCHIED, C., KAPLANYAN, A., WYMAN, C., PATNEY, A., CHAITANYA, C. R. A., BURGESS, J., LIU, S., DACHSBACHER, C., LEFOHN, A., AND SALVI, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High-Performance Graphics* (New York, NY, USA, 2017), V. Havran and K. Vaidyanathan, Eds., HPG '17, Association for Computing Machinery, p. 2:1–2:12.
- [41] SINTORN, E., KÄMPE, V., OLSSON, O., AND ASSARSSON, U. Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2014), M. Olano and M. A. Otaduy, Eds., I3D '14, Association for Computing Machinery, p. 111–118.
- [42] STONE, L. D. *Theory of Optimal Search*, vol. 118 of *Mathematics in Science and Engineering*. Academic Press, Inc., New York, NY, USA, 1975.
- [43] STÜRZLINGER, W., AND BASTOS, R. Interactive rendering of globally illuminated glossy scenes. In *Rendering Techniques '97* (Vienna, AUT, 1997), J. Dorsey and P. Slusallek, Eds., Eurographics, Springer, Vienna, p. 93–102.
- [44] SUTHERLAND, I. E. Sketchpad: A man-machine graphical communication system. In *AFIPS '63 (Spring)* (New York, NY, USA, 1963), E. C. Johnson, Ed., Association for Computing Machinery, p. 329–346.
- [45] TOKUYOSHI, Y., AND HARADA, T. Stochastic light culling. *Journal of Computer Graphics Techniques (JCGT)* 5, 1 (2016), 35–60.
- [46] TOKUYOSHI, Y., AND HARADA, T. Stochastic light culling for VPLs on GGX microsurfaces. *Computer Graphics Forum* 36, 4 (2017), 55–63.
- [47] VORBA, J., KARLÍK, O., ŠIK, M., RITSCHER, T., AND KŘIVÁNEK, J. On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics* 33, 4 (2014), 101:1–101:11.
- [48] WALD, I. On fast construction of SAH-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing* (NW Washington, DC, USA, 2007), RT '07, IEEE Computer Society, p. 33–40.
- [49] WALD, I., BENTHIN, C., AND SLUSALLEK, P. Distributed interactive ray tracing of dynamic scenes. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003*. (NW Washington, DC, USA, 2003), PVG '03, IEEE Computer Society, p. 77–85.
- [50] WALD, I., BOULOS, S., AND SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics* 26, 1 (2007), 6–es.

- [51] WALTER, B., ARBREE, A., BALA, K., AND GREENBERG, D. P. Multidimensional lightcuts. *ACM Transactions on Graphics* 25, 3 (2006), 1081–1088.
- [52] WALTER, B., FERNANDEZ, S., ARBREE, A., BALA, K., DONIKIAN, M., AND GREENBERG, D. P. Lightcuts: A scalable approach to illumination. *ACM Transactions on Graphics* 24, 3 (2005), 1098–1107.
- [53] WAND, M., AND STRAßER, W. Real-time caustics. *Computer Graphics Forum* 22, 3 (2003), 611–620.
- [54] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (Apr. 2004), 600–612.
- [55] WHITTED, T. An improved illumination model for shaded display. *Communications of the ACM* 23, 6 (Jun. 1980), 343–349.
- [56] WYMAN, C., AND DAVIS, S. Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2006), C. Sequin and M. Olano, Eds., I3D '06, Association for Computing Machinery, p. 153–160.
- [57] WYMAN, C., AND PANTELEEV, A. Rearchitecting spatiotemporal resampling for production. In *High-Performance Graphics - Symposium Papers* (Goslar, DEU, 2021), N. Binder and T. Ritschel, Eds., The Eurographics Association.
- [58] YANG, X., AND OUYANG, Y. Real-time ray traced caustics. In *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*, A. Marrs, P. Shirley, and I. Wald, Eds. Apress, Berkeley, CA, USA, 2021, p. 469–497.

Scientific publications

Author contributions

A visual representation of the author's contributions can be found in the table below. For all papers, there always was an ongoing discussion between all authors at every stage but especially while defining the concept(s).

Paper	Concept	Implementation	Evaluation	Writing
Paper I (2015–2016)	☐	◐	●	◐
Paper II (2017–2018)	◐	◐	●	◐
Paper III (2018–2019)	☐	●	●	◐
Paper IV (2019)	◐	●	●	◐
Paper V (2021)	◐	●	●	◐

- Lead and did almost all the work
- ◐ Lead and did a majority of the work
- ◑ Contributed to a majority of the work
- ☐ Contributed to a minority of the work

Concept Coming up with the ideas of the paper

Implementation Implementing the software described in the paper

Evaluation Conducting the evaluation described in the paper

Writing Drafting and editing the paper

Paper I



Photon Splatting Using a View-Sample Cluster Hierarchy

P. Moreau¹, E. Sintorn², V. Kämpe², U. Assarsson² and M. Doggett¹

¹Lund University, Sweden

²Chalmers University of Technology, Sweden

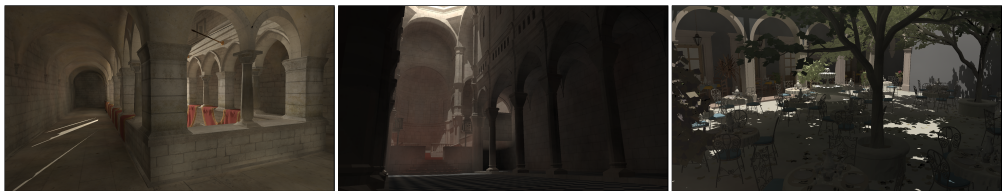


Figure 1: Views from Sponza, Sibenik, and San Miguel, rendered using our method with 200k photons and radius set up to produce a smooth image. The time taken to splat the photons is (left to right): 14 ms, 16 ms and 17 ms; full frame time is: 35 ms, 33 ms and 48 ms. The scenes were rendered at 1080p on an NVIDIA Titan X.

Abstract

Splatting photons onto primary view samples, rather than gathering from a photon acceleration structure, can be a more efficient approach to evaluating the photon-density estimate in interactive applications, where the number of photons is often low compared to the number of view samples. Most photon splatting approaches struggle with large photon radii or high resolutions due to overflow and insufficient culling. In this paper, we show how dynamic real-time diffuse interreflection can be achieved by using a full 3D acceleration structure built over the view samples and then splatting photons onto the view samples by traversing this data structure. Full dynamic lighting and scenes are possible by tracing and splatting photons, and rebuilding the acceleration structure every frame. We show that the number of view-sample/photon tests can be significantly reduced and suggest further culling techniques based on the normal cone of each node in the hierarchy. Finally, we present an approximate variant of our algorithm where photon traversal is stopped at a fixed level of our hierarchy, and the incoming radiance is accumulated per node and direction, rather than per view sample. This improves performance significantly with little visible degradation of quality.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

Global illumination algorithms attempt to simulate, in a physically based manner, how light is transported through a virtual scene. The goal is to estimate the radiance that is incident to each pixel of the image, which will be an aggregate of all possible light-transport paths that end up intersecting that pixel. Light-transport paths originate from virtual light sources and will undergo any number of reflections (where energy may be splatted or absorbed). There are several textbooks that discuss the common theory of global-illumination algorithms and the many different algorithms that numerically solve the underlying equations to produce photo-realistic images (see, e.g.,

[PH10, DBBS06]). These algorithms typically favour correctness over computation speed and can take minutes to hours to compute an image.

On the other end of the spectra are real-time, or interactive, global illumination algorithms. Depending on the use case, these algorithms have between one and a few hundred milliseconds to produce a plausible image. There exist a very large number of such algorithms, each with its own limitations and benefits. In this paper, we will explore one class of algorithms, commonly referred to as *photon splatting*, and suggest several novel improvements.

Photon splatting is a variant of the *photon-mapping* [Jen01] class

of algorithms. In the photon-splatting variant of algorithms, photons are traced from the emitters and directly visualized, by accumulating the contribution to an outgoing radiance of each photon against the *view samples* (i.e. primary ray hit points) that it affects. Thus, a photon is given some artificial *region of influence*, usually a sphere or an ellipsoid, and the main difference between splatting algorithms is how this geometric shape is intersected with all view samples.

We propose a novel approach to real-time photon splatting for global illumination. For each frame, a 3D acceleration structure is built over the current view samples. We use the *view-sample cluster hierarchy* [SKOA14], which builds a hierarchy of clusters of view samples based on screen tiles. We show that this technique can be used to efficiently cull large clusters of view samples that lie outside of a photon's region of influence before considering individual view samples. Additionally, we show that having the view samples arranged in groups of clusters (that all lie close to each other), improves performance by stopping traversal as soon as a photon contains an entire node.

Another benefit is that under the assumption that nearby view samples are likely to have similar normals, we can utilize optimizations that cull photons intersecting a node, but originating from a direction such that they will not affect any of the contained view samples. If we accept the limitation that a photon affects all view samples equally (i.e. the surfaces have a constant BRDF and no distance-based smoothing kernel is used), we can also stop traversal as soon as all view samples contained in a node have normals such that they will be affected by the photon.

In all, these improvements result in an algorithm that can render a large number of photons with sufficiently large radii to produce smooth results at high frame rates. To further push performance in the direction where it could be used as a global-illumination solution for, e.g., video games, we propose a more approximate solution. Instead of testing each photon against each individual view sample, we accumulate *directional* flux for the hierarchy nodes. In the fastest version of this algorithm, where a leaf node at most contains 32×32 view samples, we obtain very high frame rates without significant loss in quality.

2. Related Work

There is a vast body of work concerned with interactive global illumination, and we refer the reader to an excellent recent survey [RDGK12] for a more complete introduction. In this section, we will briefly discuss only the previous work that is most relevant to our proposed method.

Traditional photon mapping, where a kd-tree is built over the photons to accelerate the *gathering* of nearby photons, has been accelerated on the GPU [ZHWG08, LSP*12], but has not been shown to be practical for real-time scenarios where the light source moves. Instead, it is common to *splat* photons onto the view samples. McGuire and Luebke [ML09] find the first bounce from the light by rasterization and trace the remainder of the path on the CPU. Photons are splatted by rendering spheres that enclose the photons' influence regions. Mara et al. [MML13] explore a number of faster approaches to splatting, which will be detailed in Section 3.2.

A large portion of recent work on interactive global illumination stems from the concept of Instant Radiosity [Ke197], which is similar to photon splatting in that particles are traced from the light source and stored at each bounce. Unlike photon splatting, these particles are then treated as *Virtual Point Lights* and store the outgoing radiosity (rather than incoming flux). The scene is then lit from all such VPLs. Reflective Shadow Maps [DS05] is a GPU based variant of this, in which the first bounce from the light source is calculated using rasterization. For each view sample, a stochastic subset of the generated VPLs is gathered for shading. Nichols and Wyman [NW09] suggest an alternative approach where the VPLs are instead scattered onto a multi-resolution buffer. These techniques do not consider visibility between VPL and view sample. This is addressed in a method called Imperfect Shadow Maps [RGK*08, REH*11], where highly approximative shadow maps are calculated in real time for each VPL.

We are only aware of a few real-time global-illumination methods that are production proven. One notable example is Voxel Cone Tracing [CNS*11] in which the scene is voxelized to a sparse voxel octree and a cone-marching algorithm gathers approximate incoming radiance. Another is Cascaded Light Propagation Volumes [KD10], where the light field is stored using spherical harmonics in a coarse discretization of the view frustum. Mara et al. [MMNL14] present a fast approximation to Global Illumination using deep G-Buffers, but since the illumination is view frustum based, it does not produce a photon tracing of the scene every frame, leading to issues with off screen illumination.

The view-sample cluster hierarchies that we use in this paper were first suggested by Sintorn et al. [SKOA14], who use it to splat per-triangle shadow volumes. That paper, in turn, was inspired by the work of Olsson et al. [OBA12], who first suggested arranging view samples in clusters to speed up shading with many bounded light sources. In their work, the acceleration structure was built over the light sources, instead of the clusters.

3. Background

In this section, we will give an overview of the previous work that this paper directly builds upon.

3.1. Photon Mapping

Photon mapping, introduced by Jensen [Jen96], is a well established technique for global illumination. Photons are traced from the light source into the scene and are gathered from an acceleration structure to perform a radiance estimate when raytracing from the camera in a second pass. Usually, the photon map is not queried for the first vertex of a camera path, but secondary reflection rays are traced and where *they* hit, the photon map is used. This is called the *final-gather* step. Photon mapping has been extended in a large number of ways, and we refer the reader to [HJB*12] for a thorough overview.

3.2. Photon Splatting

For real-time global illumination, a final-gather pass is usually too costly, but tracing a number of photons through the scene to capture some indirect-illumination effects can be achieved at real-time frame

rates. Direct lighting can be efficiently computed with the standard rendering pipeline (with light visibility handled by, e.g., shadow maps). By not storing photons at the first bounce from the light source, the remaining photons can be used to directly estimate indirect illumination.

Even so, the traditional photon-map density estimate is often too costly. An alternative approach is to give each photon an a priori region of influence (or *radius*) and to *splat* the photons onto the view samples. This is often referred to as *photon splatting* [SB97] and can be implemented on a GPU by rendering the photons as geometric objects onto the current depth buffer, calculating the photons' influence on each affected view sample and accumulating the results [LP03, ML09].

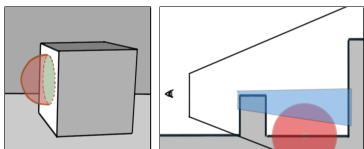


Figure 2: Photon Splatting. Left: when splatting photons using the rasterizer, view samples that lie behind the photon (red region) will not be culled. Right: A tiled renderer alleviates this, but tiles with depth discontinuities will still cause sub-optimal culling.

Mara et al. [MML13] evaluate a number of photon-splatting algorithms and find two algorithms to perform better than the rest. We will briefly explain these next. The first is 2.5D Photon Volumes that render 2D screen-aligned polygons [MM13] that represent the photons onto all the view samples that the polygon covers. The main problem with this method is that this can give rise to many unnecessary photon/view sample tests (see Figure 2).

The second method is Tiled Photon Splatting. This method associates a photon with a tile based on the tile's closest and furthest depth, reducing incorrect photon associations but still resulting in photons being incorrectly associated with tiles that have large depth ranges (see Figure 2). Also, the tile division is typically rather coarse, so that each tile will contain a long list of photons of which only a few affect each individual view sample.

4. Algorithm

In this section, we will describe our new photon-splatting algorithm in detail, beginning with the basic algorithm and then introducing some optimizations that are made possible by arranging the view samples in cluster hierarchies. Finally, we will discuss an approximate algorithm that is much faster, at the cost of a slight decrease in quality.

The basic algorithm consists of six passes:

1. **Render G-Buffer.** Using the standard pipeline, render the view sample positions, normals, and material properties to a G-Buffer texture.
2. **Generate Cluster Hierarchies.** Using the view-sample positions from the previous step, generate the cluster hierarchy (see Section 4.1).

3. **Photon Tracing.** A number of paths are traced from the light source, and at each bounce (except the first), a photon is stored to a list (see Section 4.2).
4. **Photon Splatting to Clusters.** Each photon is traversed through the cluster hierarchy, and when a node is enclosed, or a leaf node is found to be intersected, the photon ID is stored in a list unique to that node. (see Section 4.3)
5. **Radiance Estimate.** For each view sample, the lists of photons of the containing nodes are traversed and the contribution of each photon is accumulated. (see Section 4.4)
6. **Final Shading.** Direct lighting is computed in a full-screen pass, and the indirect lighting from the previous pass is added to obtain the final pixel color.

4.1. Generate Cluster Hierarchies

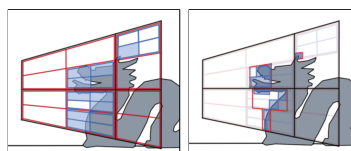


Figure 3: Left: Each cluster (cell in the 3D grid) is marked as occupied if it contains view samples. Then, parent nodes are marked as occupied recursively. Right: The bounding box of each cluster is calculated and propagated upwards in the tree.

We build a hierarchical view space 3D acceleration structure around the view samples using the technique presented by Sintorn et al. [SKOA14] and extending it for Photon Splatting. This addresses the issues of tiled photon splatting by creating much tighter bounds around view samples and allowing smaller groupings of view samples.

The cluster hierarchy divides the view frustum into a 3D grid with roughly cubical boxes. Each box is called a *cluster* and view samples are attached to one cluster. The next level of the hierarchy contains 32 clusters, a 32-bit word that indicates the occupancy of the child clusters, and a 32-bit node key (see Figure 3). Bounding boxes for each node are calculated and propagated up the tree.

4.2. Photon Tracing

This paper focuses on efficiently performing the radiance estimate, and our photon tracing is a straightforward GPU implementation using NVIDIA's Optix [PBD*10] framework. Our light sources are diffuse emitters with an angular cut-off. At each bounce, a new direction is chosen by importance sampling the BRDF (or the cosine term, for diffuse surfaces), and the new flux is calculated as:

$$\Phi' = \frac{f(\omega, \omega')}{p(\omega, \omega')} \cos(\mathbf{n}, \omega') \Phi, \quad (1)$$

where f is the BRDF, p is the Probability Density Function (PDF), and Φ is the current flux of the photon. In order to maintain a roughly similar flux among stored photons, we then terminate the path with probability $R = \min(1, \Phi'/\Phi)$ and set $\Phi = (1/R)\Phi'$ if the photon

was not terminated. We keep tracing paths until we have reached some predefined number of photons.

In previous work [ML09, MML13], it has been suggested that the photon radius could be varied according to the path probability. This is important to avoid overblurring caustics, but we are mostly interested in scenarios where very few photons are stored to obtain real-time global lighting effects (avoiding caustics) and can generally keep the radius fixed.

4.3. Photon Splatting to Clusters

This step of the algorithm is actually broken into several passes to improve overall performance. Ultimately, each photon is intersected with the view-sample cluster hierarchy and inserted into a list at every node that is completely inside the photon, or leaf node that is intersected.

Since some photons will intersect many more nodes than others, there can be load-balancing issues. As suggested in [SKOA14], we therefore implement a pass that traverses down 3 levels and pushes the current photon ID and node key to a global list. To ensure the global list has good memory coherence, it is rearranged into local groups that are used in the following insertion passes to find the leaf intersections. When running the leaf intersection passes, a sufficient number of threads are started to fully utilize the GPU and each thread fetches jobs from the global list until it is empty.

At each cluster where photons intersect, an array is allocated to store the intersection photons. We calculate the required array size for each node in a first pass (traversing the hierarchy exactly as we do when inserting photons), and store the result in an array. We then use the prefix sum (calculated using cudpp [SHG011]) of that array as a per-node index where the node's first photon shall be stored. An alternative approach to using arrays would be per-node linked lists. We have found, however, that while building these linked lists is not very expensive, iterating through them in the radiance estimation pass (Section 4.4) is very inefficient.

One thread per photon is launched on the GPU, and that thread will recursively test the photon against the nodes' bounding boxes. The traversal is implemented in an iterative fashion with a small stack in shared memory, which is initialized with the child mask from the root node. The main traversal loop then starts by looking at the top mask on the stack, and checking the first existing child node for intersection. That child node is cleared from the child mask on the stack, so as not to be tested again. When a photon is found to completely enclose a node, or the traversal reaches an intersected leaf node (cluster), the photon is appended to a list for that node and its child-nodes are not traversed.

When traversing the cluster hierarchy to find the sphere intersection we maintain a stack of child masks, and the current *node index* or *node key*. As each node is visited, the first bit of the 32-bit child mask becomes the active node, and the bit is removed from the mask and the remaining mask is pushed to the stack to track the remaining child nodes that still have to be processed. At a new node, the node key is used to fetch the child mask and bounding box of the current node from the corresponding global lists. Instead of storing the node key on the stack it can be stored in a single integer. When traversing

to the i :th child of a node, the node key is simply shifted five bits to the left, and i is appended in the lower bits. When a node has been fully processed and the stack is popped, the node key is shifted back five bits to the right. For all but the final level, the node key is used as the immediate index in the list of bounding boxes. At the final level, to reduce the memory footprint, the node key is instead used to find an index to where the corresponding bounding box is in a compact list.

The algorithm differs from the method described by Sintorn et al. [SKOA14]. First, in their algorithm, a *warp* (32 threads) is started per primitive, and the intersection tests are done in parallel. We found that, because only a few of the subnodes are likely to be occupied, letting a single thread do all intersections for the occupied nodes is more efficient, at least in our case where the intersection tests are simple sphere/bounding box tests. Secondly, in their algorithm the bounding boxes were defined in *Normalized Device Coordinates* (NDC), and the hierarchy could be tested against each per-triangle shadow volume by, starting from the root, seeing which subnodes were occupied and testing the bounding boxes of each against the shadow-volume planes (also in NDC). In our case, the primitives to be tested are spheres, which are not simple geometric shapes in NDC, so we instead store the bounding boxes in view-space coordinates, which will be overly conservative at higher levels of the hierarchy.

Note that we consider photons to have a spherical influence region, rather than a squashed sphere as has been proposed in previous work [ML09]. Clipping the sphere by one or two planes is a simple modification to the algorithm but causes unwanted artifacts as we want to avoid a smoothing kernel in our radiance estimate (as explained below).

4.4. Radiance Estimation

When all photons have been traversed through the hierarchy, we have a list of photons per node that must be considered for all contained view samples. We start one thread per view sample and loop over the photons in each containing node's list. For each photon, we check if the view sample is actually within and, if so, we accumulate the reflected radiant intensity: $I = I + f(\omega, \omega_p)\Phi_p$ (where f is the BRDF, ω is the view direction, ω_p is the incoming direction of the photon and Φ_p is the flux carried by the photon). When all photons have been processed, we calculate the outgoing radiance as $L = I/\pi r^2$, where r is the photon radius (or, if we have photons of varying radii, the distance to the furthest photon).

When an insufficient number of photons are available in an area, the distinct photon splats will become visible on the surface. It is common practice to attempt to hide this by multiplying each photon's contribution with a distance-dependent smoothing kernel. In the scenarios we are focusing on, with smoothly varying diffuse inter-reflections, we found that it was more likely to reduce the image quality.

4.5. Normal Cones

If the view samples in a node all have normals such that the photons direction is not incident to any of their tangent planes, the node can be rejected immediately.

The view samples that fall into one cluster are guaranteed to lie close to each other. Therefore, we make the assumption that they are also likely to have similar normals, and upon that assumption we attempt an optimization. When building the cluster hierarchy, starting with the leaf level, we compute the average normal and the maximum angle that a view sample’s normal deviates from this average. We call this the *normal cone* of the cluster. We later refer to this optimisation as *cluster-cone*. This normal cone is similar to that used by Sederberg and Meyers [SM88] to bound Bézier normal vectors. At the next level in the hierarchy, we generate the average of all the subnode normal-cone directions and the maximum of the subnodes’ deviation from that average *plus* the subnodes’ normal-cone angle. In this way, we propagate the normal cone upwards in the hierarchy (see Figure 4).

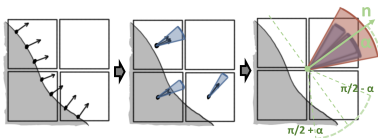


Figure 4: The normals of all view samples in a cluster are aggregated into a normal cone. The normal cones of all clusters are aggregated into normal cones for their parents.

With the normal cones in place, we can add a new simple rejection test to the photon traversal. Whenever a photon is found to intersect a node, we also check if the angle between the photon direction and the normal-cone direction is greater than $\pi/2 + \alpha$, where α is the normal-cone angle. If so, the photon will not affect any of the contained view samples and does not need to be traversed further.

4.6. Trivially Accepting Photons

For diffuse surfaces, where the BRDF is constant, the contribution of a photon to a view sample within a node is either zero (if the photon is incoming from below the surface), or constant (for all other directions). Thus, when a photon encloses a node, and we know that all view samples within that node have normals on the same hemisphere as the incoming direction, the contribution for all view samples will be the same and we can accumulate this in the node instead.

Thus, if the angle between the photon’s direction and the normal-cone direction is less than $\pi/2 - \alpha$ (i.e. the photon is incident on all contained view samples’ tangent planes), we simply add the photons flux to a per-node value. In the next pass, when estimating radiance, we add this flux multiplied by the BRDF to the view sample’s accumulated intensity. This can greatly increase the performance of the radiance estimate pass, while the traversal performance remains nearly the same. This optimisation is referenced as *cluster-trivial*.

4.7. Directional Approximation

A further performance optimization, later referenced as *directional*, can be achieved by avoiding each view sample having to loop

through all intersecting photons and instead accumulating a *directional* flux per intersected node. For each node, we store the incoming flux into a small number of buckets corresponding to discrete directions. When a photon is found to enclose a node, or intersect a leaf node, it adds its contribution to the bucket with the closest associated direction. When estimating radiance, we no longer have to process a list but simply evaluate the incoming flux from each direction, for all nodes that contain the view sample.

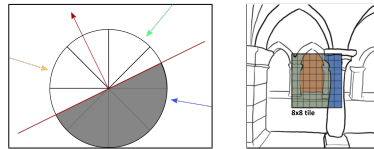


Figure 5: Left: a 2D representation of flux being stored per octant based on its incoming direction (yellow, green and blue arrows); the stored flux is later weighted to account for certain directions being below (greyed area) the view-sample’s tangent-plane (the red plane, with its normal). Right: For each tile, there can be several clusters at different depths (uniquely colored in the image), and our method evaluates illumination per such cluster.

In our implementation, we store the flux incoming from each octant of the sphere (the number of directions used has a direct impact on memory consumption, as shown in Section 5). When shading, we only have the sum of the incoming flux from each octant. Parts of, or all of, the octant might lie below the view sample’s tangent plane, and photons from those directions should not contribute. To remedy this overestimation of incoming flux, we could, for each direction, clip the corresponding octant of the unit sphere against the view sample’s tangent plane. The ratio of the area that is below the tangent plane to the area that is above would give us a reasonable weight for this direction (see Figure 5). This calculation would be too expensive to perform for every view sample and direction, but the weights depend only on the view-space normal, so we could pre-compute it and store it in a small cubemap. In practice, however, we have found that a much simpler heuristic gives acceptable results. The contribution from each direction is simply weighted with $2 \cos(n, d)$, where n is the view-space normal, and d is the direction in view space. We multiply by 2 to account for that the integral of the cosine is smaller (π) than the integral of the hemisphere (2π).

The proposed method only gives us an estimation of the incoming flux per cluster. If we use that result to shade each view sample, the result can be visibly blocky when the light-field changes quickly. To alleviate this, for diffuse materials, we store only the irradiance in a texture, and apply a depth- and normal-aware blur filter to that texture. The blurred irradiance is then multiplied by the diffuse BRDF in the final shading stage.

It is important to note that the proposed approximate method gives much better results than simply rendering the indirect illumination at a coarser resolution. E.g. a 32×32 tile can contain view samples from several distinct surfaces at different depths which will fall into different clusters. Our method will sample the irradiance for each of these, whereas a simple upsampling would pick only the one in the middle of the tile (see Figure 5).

5. Results

All measurements have been made at a 1920x1080 resolution on a NVIDIA Titan X GPU with 12 GB VRAM using OptiX [PBD*10] 3.9.0 for tracing the photons in real-time and CUDA [NBGS08] 7.0 for every step of the algorithm, apart from generating the G-buffers, and the shadowmaps, as well as the blur pass, which are done using OpenGL. We limit the photon tracing to four bounces and do not store photons on the first bounce but instead use standard deferred shading to compute the direct lighting.

Three different scenes are used throughout this paper: Sponza, Sibenik and San Miguel (see Figure 1). For each of them, we have created a fly-through animation (see the accompanying videos). These fly-throughs do not include moving lights or geometry, but we have supplied additional videos showing dynamic scenes. The total frame time, including the time taken to trace photons is given in Figure 1. Updating the acceleration structures when an object moves adds less than a millisecond to these times.

5.1. Performance

We compare the splatting and shading performance of our method, with and without optimizations, to our own implementation of the tiled photon splatting presented by Mara et al. [MML13]. We have not implemented the stochastic selection of photons proposed by Mara et al., and therefore, we do not preload the shared memory with photons but interleave the loading of photons with the arithmetic operations of computing their contribution. This also simplifies the case when list sizes do not fit in shared memory. For the tiled algorithm we use a tile size of 32×32 .

Figure 6 presents a comparison of each method’s execution time (photon tracing and G-buffer generation is not included, as the time is similar for all methods) for a fly-through of each scene. The cluster-trivial method is on average about two times faster than the tiled method, and up to three at some peaks. The tiled method has a lower overhead compared to our methods, and will therefore be faster when few photons are being splatted (see Figure 6a, around frame 150). With the directional algorithm, we achieve another 2-3x speedup and the difference in image quality is minimal (see Figure 10, and Table 2). The directional algorithm also shows little variance in execution times, due to better load balancing. The directional algorithm stores radiance data starting at one level above the leaf nodes in the cluster hierarchy. The performance difference between the different optimizations added to our cluster version are presented in Figure 9.

When breaking down the execution times of the different methods into passes (see Figure 8), the tiled method is dominated by the final shading pass. The total frame time is dominated by the photon tracing, but it should be noted that the photon tracing code, which relies on OptiX, has not been optimized, nor has the deferred rendering pass; better performance results could therefore be expected for these steps.

5.2. Memory Consumption

Our cluster method statically allocates memory for a dense hierarchy, even though we only build and use a sparse hierarchy per

	Cluster	Directional	Tiled
Tiles z-Bounds	-	-	0.016
Cluster Hierarchy	97	97	-
Final Bounds	256	256	-
Normal Cone	24	-	-
Accum. Flux	73	582	-
<i>Sub-Total</i>	450	935	0.016
Jobs	2.4	2.4	-
Photons Array	60	-	28
Photon Map	0.16	0.16	0.16
<i>Sub-Total</i>	63	7.2	28
Total	513	942	28

Table 1: Memory-consumption (in MiB) breakdown for our cluster-trivial version, a directional version using eight regions and a tiled version using 32×32 tiles, all of them at a resolution of 1920×1080 and using 10k photons of radius 4 in Sponza.

frame. We use a similar hierarchy to the 1080p hierarchy of Sintorn et al. [SKOA14] with six half-floats for each AABB and 32-bit childmask per node. In addition, we store one 32-bit word per node for the normal cone (theta and the normal cone angle are compacted to 8-bit values, whereas phi requires 16 bits), and 3 floats per node for the per-node flux, effectively doubling the memory consumption.

As explained in Section 4.3, we split the splatting pass into two sub-passes. The first sub-pass splats the photons down to a certain pre-defined level, from which the second sub-pass starts and continues splatting further down. This means that we need to store additional data, the *jobs* mentioned in Table 1, which is an array of pairs (a 32-bit value for the photon id, and another 32-bit value for the cluster key); as we group the jobs per cluster to improve data locality, two lists are needed in practice. The largest size required for this list during our experiments was 30M elements (requiring 458 MiB for both lists) for 50M photons of radius 0.2 in Sponza, and we simply pre-allocate a list of sufficient size. In order to reduce its memory footprint, the different sub-passes can be run several times on a smaller sized list.

The different methods presented here generate arrays of photons per cluster/tile. Those photons are stored in a compact format, amounting to 16 bytes per photon. Position and flux are both stored as three half-floats, the orientation as two 8-bit values (theta and phi), and the radius as a 16-bit value. We pre-allocate a sufficiently large array to hold these photons, and in our experiments it has never exceeded 102M elements (requiring 1.5 GiB when storing the full photon, rather than just its id) for 50M photons of radius 0.2 in Sponza.

In the directional method, we do not need to store individual photons per cluster, but we instead need to allocate memory for storing the accumulated flux for each direction of each node, starting at the level where the first splat sub-pass stops, down to the leaves. This amounts, for eight regions per node, to 24 floats per node, which translates to a total of 582 MiB for a 1080p resolution and starting the accumulation from level 4.

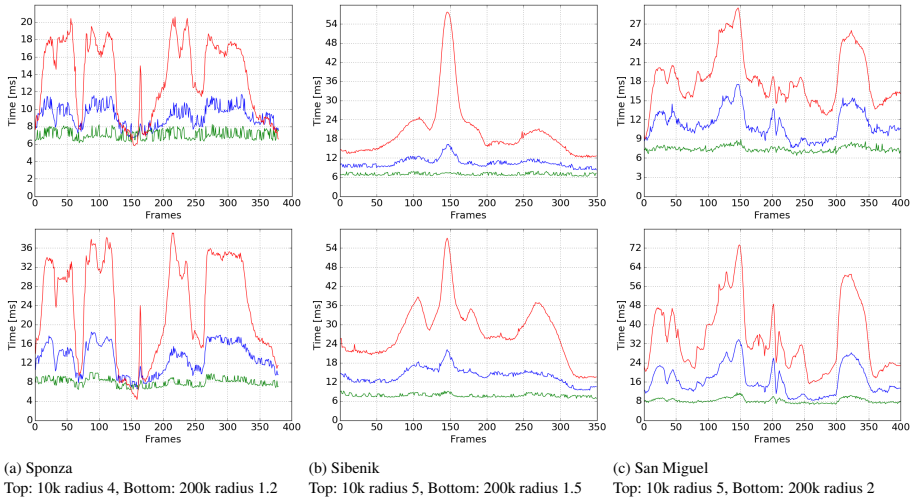


Figure 6: Comparison of cluster, directional and tiled methods splatting time for the San Miguel, Sibenik and Sponza scenes using different radii and photons numbers. The red, blue and green curves correspond respectively to tiled, cluster-trivial and directional.

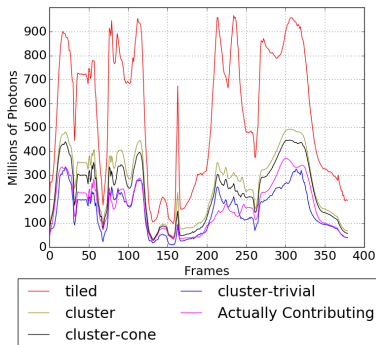


Figure 7: Sum of photons read in total during the shading pass for tiled shading and our method compared against the number of actually contributing photons. (case: 10k photons of radius 4 in Sponza)

5.3. Photon Splatting Efficiency

In Figure 7, we see that the tiled splatting reads many more photons during the shading pass than are actually contributing to the shading. Our method results in much fewer reads due to the much better spatial bounds. With our method, on average, less than 3% of the photons read from the lists are rejected by testing the view sample position against the photon influence sphere.

With the cluster-cone optimization (see Section 4.5) we discard even more photons and obtain more relevant lists (see Figure 7) and with the cluster-trivial optimization (see Section 4.5), we replace some of the insertions with accumulation of flux, which decreases the number of list reads to be even fewer than the number of photons that are actually contributing to the radiance.

5.4. Quality Evaluation

In order to assess the quality of our algorithms resulting images, we compute their SSIM [WBS04] and PSNR mean score against a path traced reference image generated using Embree [WWB⁺14]. The results are summarised in Table 2. The cluster and tiled methods have almost identical SSIM mean score, which is expected as they end up shading view samples with the same list of photons: only the way those lists are computed changes. Even though the directional method is an approximation, its SSIM mean score remains only slightly below the cluster score. For a visual comparison, Figure 10 presents the final image for the different methods in Sponza with 200k photons of radius 2.

By using more photons and of smaller radius, our experiments show that the final image quality improves, as regular photon mapping would. This is supported by the SSIM and PSNR scores listed in 2, at least for the cluster and tiled methods. Combined with the scaling of our algorithm, performance-wise (see Figure 8), for "higher quality" setups, we expect the cluster-trivial method to perform favorably on newer hardware, without any additional modifications. Figure 11 compares the final image for our cluster-trivial optimisation in Sponza between a low-quality setting (10k photons of radius 4) and a high-quality one (50M photons of radius 0.2).

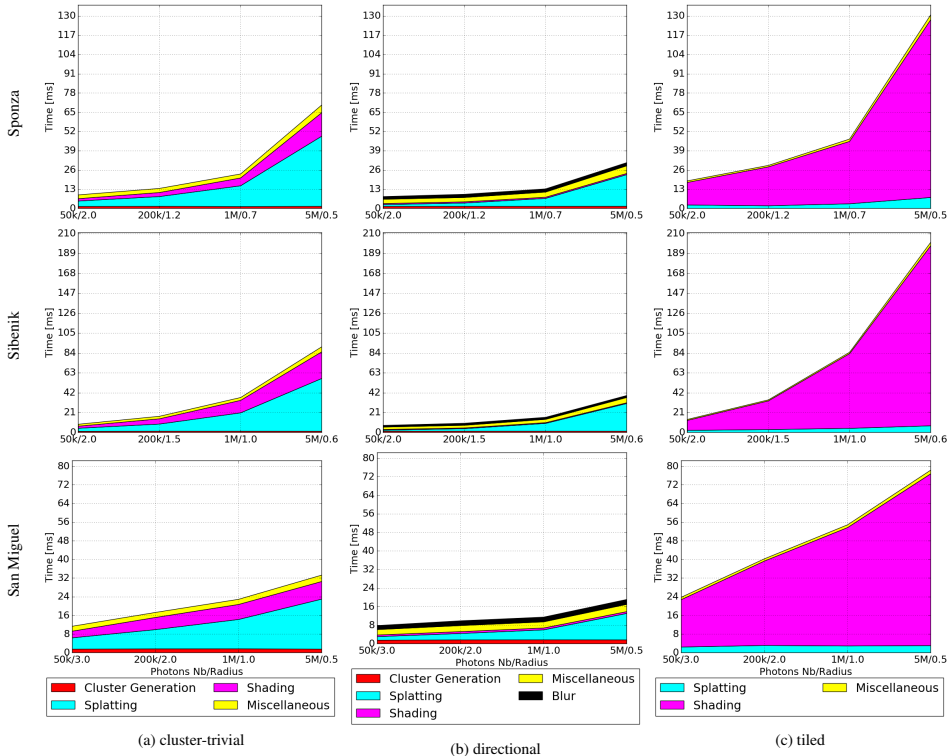


Figure 8: Breakdown of total frame time into its various components, for our cluster-trivial and directional methods, as well as Mara et al. tiled, inside the Sponza scene using the same viewpoint as in Figure 1. *Miscellaneous* groups buffer clearing, texture mapping and unmapping.

5.5. Reflections

Our method can be used to render non-diffuse BRDFs, but we cannot use the cluster-trivial optimization (as the incoming direction of each photon is important for view-dependent BRDFs). Figure 12 shows an example of glossy reflections. Note that, while curved objects look convincing, we do not shoot enough photons (nor small enough photons) to capture glossy reflections from e.g. a flat floor.

6. Discussion and Limitations

We have shown that view-sample cluster hierarchies can be used to perform fast radiance estimates in interactive settings where photons are few enough to be traced per frame, and large enough to provide a smooth result. Additionally, we have shown that an approximate version of our method can produce convincing diffuse inter-reflection images at around 10 ms per frame, bringing global illumination closer to use in real-time applications like video games.

Like most realtime GI algorithms (including production proven algorithms like Voxel Cone Tracing and Light Propagation Volumes), the most obvious drawback of Photon Splatting is the existence of light leakage. While clearly visible in all of our renderings, we have not found these artifacts too disturbing, however, and are convinced that photon splatting (at the low cost we obtain) is a usable solution for global lighting phenomena. As we focus on a small number of large photons, high frequency phenomena, like caustics, are poorly represented, but still supported by our method. More convincing results might be attained by a higher number of smaller photons. Similarly, glossy surfaces are supported by our method, but results remain poor, even with a higher number of small photons, as it does for traditional photon mapping.

We have illustrated efficient splatting in the context of photon splatting, but the same method could be applied to VPL based algorithms, and we are eager to explore if there may be other uses (e.g. ambient occlusion).

Scene	Photons Nb	Radius	Cluster		Directional		Tiled	
			SSIM	PSNR	SSIM	PSNR	SSIM	PSNR
Sponza	10k	4.0	90	25	91	27	90	24
	50k	2.0	91	26	90	25	92	27
	200k	1.2	93	30	92	27	93	30
	1M	0.7	94	32	92	27	94	32
	5M	0.5	94	32	92	27	94	32
	50M	0.2	95	33	89	25	95	33
San Miguel	10k	5.0	75	22	81	25	76	22
	50k	3.0	82	26	83	26	81	25
	200k	2.0	84	26	83	26	83	26
	1M	1.0	85	27	82	26	84	27
	5M	0.5	85	28	82	26	85	27
	50M	0.2	86	28	83	27	85	28
Sibenik	10k	5.0	91	28	87	27	92	27
	50k	2.0	88	31	85	28	89	31
	200k	1.5	94	32	89	28	93	32
	1M	1.0	95	34	90	28	94	34
	5M	0.6	96	36	94	30	95	36
	50M	0.2	96	38	93	30	96	38

Table 2: SSIM (in %) and PSNR (in dB) results for various setups across the three test scenes using the cluster-trivial and the directional methods against a path traced reference image generated using Embree.

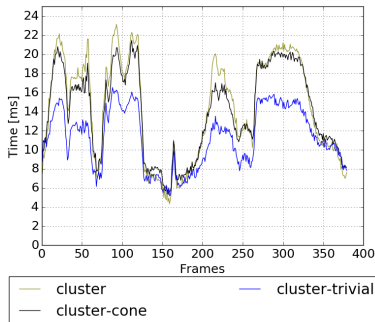


Figure 9: Total execution time for our method with and without optimizations. Included timings are splatting, shading and other essential steps, e.g., building the cluster hierarchy. Tracing of photons and deferred shading are not included. (case: 200k photons of radius 1.2 in Sponza)

Acknowledgements We would like to thank Jacob Munkberg and Jon Hasselgren for their critical and helpful comments about the paper. We use Mehdi Rabah’s SSIM implementation [Rab]. The Sponza scene is created by Frank Meinel, Sibenik is by Marko Dabrovic, and San Miguel is by Guillermo M. Leal Llaguno. All scenes are freely available at Morgan McGuire’s Computer Graphics Archive [McG]. Pierre and Michael thank the Swedish Research Council under grant 2014-5191 and ELLIIT for funding. Erik, Viktor

and Ulf are supported in part by the Swedish Research Council under grant 2014-4559.

References

- [CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing: A preview. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D ’11, ACM, pp. 207–207. 2
- [DBBS06] DUTRE P., BALA K., BEKAERT P., SHIRLEY P.: *Advanced Global Illumination*. AK Peters Ltd, 2006. 1
- [DS05] DACHSBACHER C., STAMMINGER M.: Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), I3D ’05, ACM, pp. 203–231. 2
- [HJB*12] HACHISUKA T., JAROSZ W., BOUCHARD G., CHRISTENSEN P., FRISVAD J. R., JAKOB W., JENSEN H. W., KASCHALK M., KNAUS C., SELLE A., SPENCER B.: State of the art in photon density estimation. In *ACM SIGGRAPH 2012 Courses* (New York, NY, USA, 2012), SIGGRAPH ’12, ACM, pp. 6:1–6:469. 2
- [Jen96] JENSEN H. W.: Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques ’96* (London, UK, UK, 1996), Springer-Verlag, pp. 21–30. 2
- [Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001. 1
- [KD10] KAPLAYAN A., DACHSBACHER C.: Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D ’10, ACM, pp. 99–107. 2
- [Kel97] KELLER A.: Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH ’97, ACM Press/Addison-Wesley Publishing Co., pp. 49–56. 2
- [LP03] LAVIGNOTTE F., PAULIN M.: Scalable photon splatting for global illumination. In *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South*



(a) cluster-trivial



(b) directional

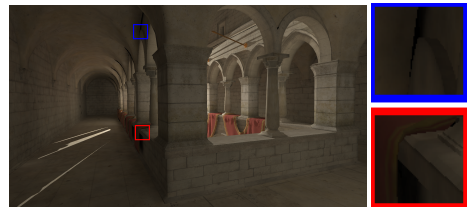


(c) tiled

Figure 10: View from Sponza rendered using our cluster-trivial method, our directional method, and Mara et al. tiled method with 200k photons of radius 1.2.



(a) 10k photons of radius 4 in 10 ms



(b) 50M photons of radius 0.2 in 290 ms

Figure 11: Comparison between a "low-quality" setting and a "high-quality" setting in Sponza; the time are those from cluster-trivial.



Figure 12: Glossy materials of varying roughness rendered with our method. 50k photons with radius 1. Total time per frame: 34 ms.

East Asia (New York, NY, USA, 2003), GRAPHITE '03, ACM, pp. 203–ff. 3

[LSP*12] LI S., SIMONS L., PAKARAVOOR J. B., ABBASINEJAD F., OWENS J. D., AMENTA N.: kann on the gpu with shifted sorting. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2012), EGGH-HPG'12, Eurographics Association, pp. 39–47. 2

[McG] MCGUIRE M.: Computer graphics archive. <http://graphics.cs.williams.edu/data> Accessed on 2016/03/29. 9

[ML09] MCGUIRE M., LUEBKE D.: Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics* (New York, NY, USA, August 2009), ACM. 2, 3, 4

[MM13] MARA M., MCGUIRE M.: 2d polyhedral bounds of a clipped, perspective-projected 3d sphere. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (August 2013), 70–83. 3

[MML13] MARA M., MCGUIRE M., LUEBKE D.: Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation. In *Interactive 3D Graphics and Games 2013* (March 2013), 2, 3, 4, 6

[MMNL14] MARA M., MCGUIRE M., NOWROUZSAHRAI D., LUEBKE D.: *Fast Global Illumination Approximations on Deep G-Buffers*. Tech. Rep. NVR-2014-001, NVIDIA Corporation, June 2014. 2

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with cuda. *Queue* 6, 2 (Mar. 2008), 40–53. 6

[NW09] NICHOLS G., WYMAN C.: Multiresolution splatting for indirect illumination. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 83–90. 2

[OBA12] OLSSON O., BILLETER M., ASSARSSON U.: Clustered deferred and forward shading. In *HPG '12: Proceedings of the Conference on High Performance Graphics 2012* (2012). 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics* (August 2010). 3, 6

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. 1

- [Rab] RABAH M.: C++ implementation of SSIM. <http://mehdi.rabah.free.fr/SSIM/> Accessed on 2016/03/29. 9
- [RDGK12] RITSCHEL T., DACHSBACHER C., GROSCH T., KAUTZ J.: The state of the art in interactive global illumination. *Comput. Graph. Forum* 31, 1 (Feb. 2012), 160–188. 2
- [REH*11] RITSCHEL T., EISEMANN E., HA I., KIM J. D., SEIDEL H.-P.: Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes. *Computer Graphics Forum (presented at EGSR 2011)* (2011). 2
- [RGK*08] RITSCHEL T., GROSCH T., KIM M. H., SEIDEL H.-P., DACHSBACHER C., KAUTZ J.: Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph.* 27, 5 (Dec. 2008), 129:1–129:8. 2
- [SB97] STÜRLINGER W., BASTOS R.: Interactive rendering of globally illuminated glossy scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97* (London, UK, UK, 1997), Springer-Verlag, pp. 93–102. 3
- [SHG011] SENGUPTA S., HARRIS M., GARLAND M., OWENS J. D.: Efficient parallel scan algorithms for many-core gpus. In *Scientific Computing with Multicore and Accelerators*, Kurzak J., Bader D. A., Dongarra J., (Eds.), Chapman & Hall/CRC Computational Science. Taylor & Francis, Jan. 2011, ch. 19, pp. 413–442. 4
- [SKOA14] SINTORN E., KÄMPE V., OLSSON O., ASSARSSON U.: Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2014), I3D '14, ACM. 2, 3, 4, 6
- [SM88] SEDERBERG T. W., MEYERS R. J.: Loop detection in surface patch intersections. *Computer Aided Geometric Design* 5, 2 (1988), 161 – 171. 5
- [WBSS04] WANG Z., BOVIK A. C., SHEIKH H. R., SIMONCELLI E. P.: Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (April 2004), 600–612. 7
- [WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.* 33, 4 (July 2014), 143:1–143:8. 7
- [ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.* 27, 5 (Dec. 2008), 126:1–126:11. 2

Paper II



Path Verification for Dynamic Indirect Illumination

PIERRE MOREAU, Lund University, Sweden

MICHAEL DOGGETT, Lund University, Sweden

ERIK SINTORN, Chalmers University of Technology, Sweden

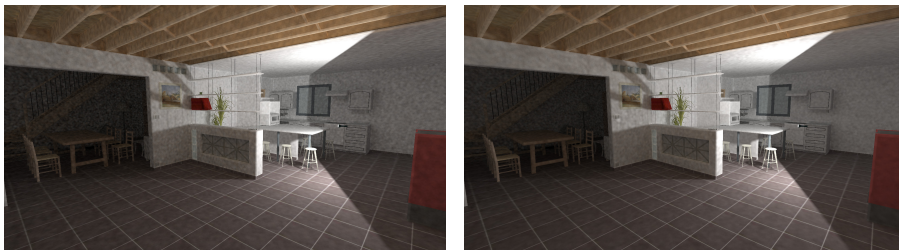


Fig. 1. Rendering in the Villa scene for the baseline on the left, and our error-based method on the right at roughly equal frame time (341 ms, and resp. 333 ms); in both cases, the number of bounces is limited to 7. The baseline traces 3 million paths in about 178 ms, and the splatting of the photons takes 148 ms. On the other hand, our method traces and reuses 5 million paths in about 80 ms, and the splatting of the photon takes 236 ms.

In this paper we present a technique that improves rendering performance for real-time scenes with ray traced lighting in the presence of dynamic lights and objects. In particular we verify photon paths from the previous frame against dynamic objects in the current frame, and show how most photon paths are still valid. When using area lights, we use a data structure to store light distribution that tracks light paths allowing photons to be reused when the light source is moving in the scene. We also show that by reusing paths when the error in the reflected energy is below a threshold value, even more paths can be reused. We apply this technique to Indirect Illumination using a screen space photon splatting rendering engine. By reusing photon paths and applying our error threshold, our method can reduce the number of rays traced by up to 5 \times , and improve performance by up to 2 \times .

CCS Concepts: • **Computing methodologies** \rightarrow *Ray tracing; Rendering.*

Additional Key Words and Phrases: Photon Mapping, Global Illumination

1 INTRODUCTION

Indirect illumination is an important cue for the perceived realism of computer generated imagery, but its accurate computation can be computationally expensive. A recent survey by Ritschel et al. [2012] covers many algorithms that approximate indirect illumination for real-time applications. Since the general problem is complex, and cannot be easily solved even in offline rendering, where hundreds of cores can spend hours on a single frame, most real-time algorithms are specifically designed to generate a good estimation under very specific assumptions about lighting and materials.

Computing indirect illumination and soft shadows, while considering animated objects, leads to a more accurate representation

of the lighting in a scene, than can usually be achieved with pre-computed techniques. In this paper we build upon the photon splatting technique by Moreau et al. [2016] to enable indirect lighting with multiple dynamic light sources. Unlike Moreau et al. [2016], where the photon map is recomputed every frame, we propose to opportunistically reuse as many photon paths as possible, including those from moving light sources. We achieve this by reusing lighting from previous frames, and within area lights. In this paper we present a technique that uses a photon map with several bounces, many more than previous techniques, enabling subtle lighting effects in neighboring areas which have no direct path to the light source.

Photon mapping traces light paths from the light, and deposits photon energy onto diffuse surfaces. To create the final image, camera rays are traced to gather the light deposited on surfaces in the scene. Path tracing instead traces rays from the camera through the scene until they find a light source. This means that for traditional path tracing, all segments of a path must be verified if lights or objects move in the scene. While for photon mapping, only the light paths from the light to the surface need to be verified, as camera rays will be traced regardless.

In scenes with moving objects and dynamic light sources, we present techniques for path verification. If a path from a previous frame is classified as still valid, we reuse the photon path in the current frame. By reusing photon paths we are able to achieve interactive frame rates in scenes with indirect illumination. Unlike previous methods with geometric approximations or sparse samples sets, we use a dense photon map, and instead of recomputing all light per frame, including lighting that is still valid, we carefully reuse the light transport from the previous frame.

Using photon maps allows indirect illumination to be computed in world space without the limitations of screen space methods, and

Authors' addresses: Pierre Moreau, Lund University, Sweden; Michael Doggett, Lund University, Sweden; Erik Sintorn, Chalmers University of Technology, Sweden.

© 2021 Copyright held by the authors, published under Creative Commons CC-BY-SA-4.0 License.

enables the possibility to temporally reuse light transport computations from previous frames.

2 RELATED WORK

Early attempts to achieve precomputed light transport stored it in textures [McTaggart 2004], or used spherical harmonics to store precomputed light transport [Sloan et al. 2002], or, more recently, in precomputed light field probes [McGuire et al. 2017]. These algorithms can be very efficient to query, but require significant computational resources for offline precomputations, and large memory buffers for high quality results. More importantly, they do not allow for dynamic lights. On the other end of the spectra are screen space reflection algorithms [Sousa et al. 2011] that can very quickly estimate the radiance reflected from a glossy material, but only when the reflected surfaces are directly visible to the user. Other screen space algorithms [Ritschel et al. 2009] are only effective for local reflections when the material is lambertian and do not take into account more complex materials or light sources outside of the viewing volume.

Dmitriev et al. [2002] used two different types of photons to detect areas where lighting changed between frames, and focused updates to the lighting in those regions using *corrective photons*. This allows them to support dynamic scenes and prioritise the updates to perform, however each corrective photon needs to be traced twice, with one of the tracings taking place against an earlier version of the scene.

To reduce the number of paths needed per pixel, Bekaert et al. [2002] proposed creating path segments in neighboring pixels and sharing those paths to increase the number of paths traced in each individual pixel. While our work also attempts to reduce computation per frame by reusing paths, we reuse the photon paths from the previous frame, not neighboring pixels.

Voxel Cone Tracing [Crassin et al. 2011] alleviates both of these problems by voxelizing a rough representation of the scene around the user’s position, and can be used for diffuse indirect illumination, but is much too expensive in terms of memory to allow for large scenes. Also this algorithm does not allow for many bounces of light.

Also dynamic scenes have been mixed with Stochastic Progressive Photon Mapping [Weiss and Grosch 2012], but this is a much more complex technique and requires much longer frame times than our technique.

More recently, several denoising algorithms have been suggested, that allow for fast denoising of extremely noisy path-traced images [Chaitanya et al. 2017; Mara et al. 2017; Schied et al. 2017]. By filtering samples both spatially and temporally, the reflected radiance of each pixel can be estimated almost as well as if hundreds of indirect illumination rays had been shot per pixel and can handle both glossy and diffuse surfaces. While these approaches can generate path traced images at real-time rates, they use short path lengths to ensure performance.

Another recent method is that of Silvennoinen et al. [2017] where a very sparse set of light probes are updated every frame by shooting a single ray per direction and looking up the intersected surface’s direct-lighting response in a texture. Our technique robustly checks

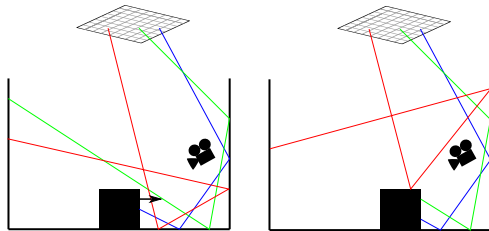


Fig. 2. A simple scene showing an area light source at the top, shown as a grid, from which several photon paths are traced. A small cube moves in the bottom of the scene, shown in its starting position on the left, and its new position on the right. In the right image the cube intersects the existing segments of all paths. The grid at the top represents what we call a distribution map covering the two dimensional area light. For the green and blue paths, as only their last segment is being intersected, we only need to compute the new intersection point. On the other hand, the red path’s first segment is intersected also triggering a computation of the new intersection point, but also a resampling of the BRDF there, due to the second photon having become invisible from the first one. However, since the new second photon can see the old third one, and the energy between the two is similar to before, the third photon is kept as-is.

all dynamic objects, unlike Silvennoinen et al. [2017], which only has support for approximated dynamic objects and so will fail to correctly capture the illumination when all light has undergone several bounces before reaching the camera.

Corso et al. [2017] recently looked into reusing shading information at primary view samples from previous frames. This is done by reprojecting the view sample locations into the current frame and validating their visibility. They also maintain a uniform distribution of outgoing rays to avoid having too many or too few paths at a given pixel. Our approach extends the validation to consider the whole path rather than just the first segment, and we modified how the uniform distribution is maintained to apply to light sources and support area lights.

3 ALGORITHM

Given a scene made only of static objects and lights, the tracing of the light paths only needs to be done once and can be reused for all frames. In this paper we focus on the reuse of light paths from previous frames in the presence of dynamic objects and lights, and are not concerned with static scenes. To verify that a light path is still valid in the current frame, it must be checked against moving objects and light sources. Figure 2 illustrates how the algorithm handles a single moving object intersecting three photon paths. In this section we outline how this verification of light paths is performed first for dynamic lights, and then for dynamic objects.

Our algorithm is made of 5 main steps, that process all light paths from previous frames and tell a slightly modified photon mapper/splatter which paths should be retraced; details about the modifications done to the photon mapper/splatter can be found in Section 4. In the following we give a brief introduction to the five

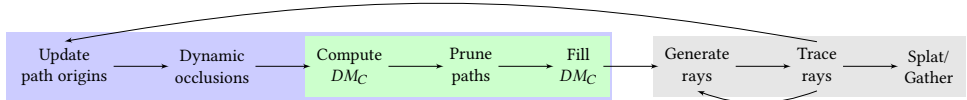


Fig. 3. A diagram showing where our algorithm sits in a regular photon renderer, visualised as a grey box, as well as its different main steps. The green box groups steps involving distribution maps together, whereas the blue box groups all steps needed for reusing photons from frame to frame. The “generate rays” and “trace rays” steps bare a few differences in our algorithm, compared to the classic version. However, as those are not significant, they are represented as the same steps in this diagram; the differences will be presented in Section 4. DM_C represents the *distribution map* of the current frame, computed from the existing light paths.

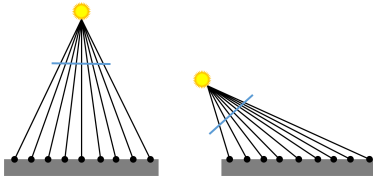


Fig. 4. On the left, a spotlight is lighting a certain area on the ground, with the primary photons represented as black spheres and its near plane as a blue line. If one was to keep the same photons as the light moves (while still illuminating the same area), it would result in a distribution of light across the near plane that is different from the initial one, as seen on the right.

main stages of our algorithm, and then explain them in more detail later in this section.

Update path origins to match the current position and orientation of light sources.

Dynamic occlusions will detect and schedule for re-tracing paths intersected by dynamic objects.

Compute DM_C to know how many light paths are emitted from each cell.

Prune paths to decrease the number of emitted paths for cells with too many light paths.

Fill DM_C to increase the number of light paths in cells below the required amount.

Apart from “dynamic occlusions”, which will be presented in Section 3.2 as it is unnecessary for dynamic lights, the remaining four main steps will be presented in Section 3.1. Figure 3 shows the structure of the five steps and where they sit in relation to a regular photon splatting architecture.

3.1 Supporting dynamic lights

As lights move in a scene, some surfaces that were previously unlit become now visible to a light source and receive light, while others fade in the shadows. To validate light paths against such behaviours, a first approach would be to test whether the primary segment of each path is still within the light’s field of view, and if not, replace the whole path by a new one. However this can lead to changes in the light’s distribution, as showcased in Figure 4.

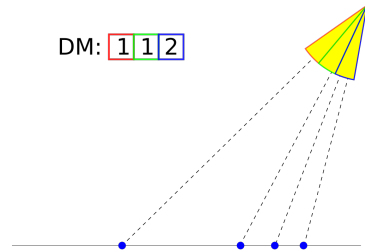


Fig. 5. A simple example of a distribution map for a spotlight in 2D. Here the distribution map is composed of only three cells, and is represented in the top-left corner as an array. Each cell keeps track of how many paths originated from a specific region on the light; the cells form a partition of all possible origin configurations. The mapping between a cell and its corresponding region on the light is colour-coded and can be visualised directly on the figure.

Even a light that moves parallel to the plane it is illuminating, will have issues if photons that are now no longer visible from the light are randomly re-traced over the whole volume visible from the light. This would result in very few photons in the newly visible areas, as many of the new photons would end up in the already visible areas. To avoid those issues, we propose to maintain the distribution of photons from the light source between frames.

To achieve equal distribution across the light we partition its surface, and its set of outgoing directions into cells, and ensure that each cell maintains a given amount of primary paths emitted from that cell. Those cells form an n -D array which we call a *Distribution Map* (DM). The parametrisation of this array is not constrained and can be different for different light types. For example, a point light could have a 2-D parametrisation (θ, ϕ) whereas a rectangular area light could use a 4-D parametrisation (x, y, θ, ϕ) ; an example of parametrisation for a 2-D spotlight can be seen in Figure 5.

The distribution map is initialised with a user-defined distribution for the light source. This initial set of values is denoted as DM_I , or the *targeted* distribution of the light. This target distribution could be updated every frame to allow for textured light sources. A second distribution map, noted DM_C , is computed each frame

using all existing paths at the end of the previous frame. Each frame, we apply a set of operations to make DM_C converge towards DM_T ; those operations might update the values stored in DM_C to ensure that it counts only valid paths.

Update path origins. As the lights move, we need to update the position on the light from which the paths are emitted. For a point light, this simply means setting the light sample to the new position of the light and recomputing the outgoing direction based on this new position and the existing primary photon. We also check that the primary photon is still visible from the light, by making use of the attached shadow map. This will however not work for area lights, so in those cases, to avoid tracing visibility rays towards each primary photon, we keep the existing outgoing direction and compute its intersection with the plane of the light to get the new origin of the path. Some of the light paths can already be invalidated during this step.

Compute DM_C . As the light path origins have been updated to reflect the current position and orientation of the lights, we can now compute how many light paths are emitted from each cell; this is done for all paths that were successfully updated in the previous step. If the parametrisation function of the distribution map returns a correct value given the position on the light and outgoing direction of a light path, the cell found to have emitted this path is atomically increased. Otherwise, the path is marked as invalid and will be re-traced in the “fill DM_C ” step.

Prune paths. Thanks to the previous step, we now know how many paths lie in each cell. Some of them might contain more paths than they should, if the light moved. In order to converge back to DM_T , for each cell where $DM_C > DM_T$, every path emitted from that cell will be pruned with the following probability:

$$\frac{DM_C - DM_T}{DM_C} \quad (1)$$

note that this does not ensure that DM_C will be equal to DM_T , but ensures that DM_C will converge towards the target over a number of frames. Also, all paths pruned by this pass are valid paths: we could keep them and reduce their energies, however that could result over time in paths with low energy, so we prune them instead.

Fill DM_C . For the same reasons that some cells will contain more paths, others will be lacking some paths. For each cell to reach its expected amount of paths, we sample the light to obtain a new position on the light and outgoing direction. The sampling of the light is restricted to the domain contained within the cell. Those inputs will later be used to trace new light paths in the “trace rays” step.

3.2 Supporting dynamic objects

To handle dynamic objects, the “dynamic occlusions” step of the algorithm adds visibility rays to compute whether the visibility between two vertices of the path changed. These rays test the current segment against the bounding box of every dynamic object in the scene, and kill the segment if any of the tests fail. This is a conservative approach and might return false positives.

In order to avoid unnecessary tests, we test the segments of a path in order, starting from the segment leaving the light. If the i th segment of the path is intersected by a dynamic object, all segments after it will be different. After a segment is found to be intersected by a dynamic object, we schedule that segment and all the following ones to be re-traced.

3.3 Error-based threshold for path reuse

While the solution presented in Section 3.2 is straightforward, some paths propagate very similar energy from frame to frame and could be reused. Instead of killing the intersected segment i , we trace a visibility ray from the segment’s origin to its destination, and compute the new end of that segment, which is also the origin of the next segment j . As we updated the origin of j , we may break the visibility between both ends of that segment. We can trace a new visibility ray along j to compute its new end point, and continue similarly until we reach the end of the path. The error-based threshold algorithm is shown in Algorithm 1.

We can however avoid the visibility ray under certain circumstances: if the segment is not intersected by any dynamic objects, neither its origin nor end are located on dynamic objects, and its newly computed origin is located at the same position as its old origin. This situation can occur when segment i is intersected by the bounding box of a dynamic object, but in practice is not intersected by any of the object’s triangles.

When reusing path vertices, sometimes the energy reflected E_N , at the new intersection point is very similar to how much energy E_O was being reflected at the old intersection point. We detect these cases by using a user specified energy threshold T , and if Equation 2 is satisfied, we don’t update or propagate the new energy value saving valuable computation time. Otherwise, all the segments – starting from the current vertex – are re-traced. By always comparing E_N to the original reflected energy E_O , we ensure that we do not accumulate errors for the energy over multiple frames.

$$(-T \times E_O \leq E_N - E_O) \wedge (E_N - E_O \leq T \times E_O) \quad (2)$$

This technique will never trace more rays than if the whole path had been invalidated, and can improve the temporal coherency by reusing some of the segments.

4 IMPLEMENTATION DETAILS

Each path is made up of several photons, and in order to keep track of the paths’ structure, including the photon order, we store the photons in a 2-D array where the i -th row contains the i -th photon of a path, and each column is a different path. This memory layout, rather than its transpose, allows for better memory access patterns, as all threads loading their i -th photon will result in consecutive memory accesses. Photons use a total of 32 bytes:

- Incoming direction (as XYZ): 3 floats;
- ID of object hit: 32-bit integer;
- Energy (as RGB): 3 floats;
- Radius: float.

To help with the current status of a path, we store separately a small data structure (a single 32-bit word) containing the following:

- The ID of the DM_C ’s cell in which this path lies. (22 bits);

Algorithm 1: Error-based threshold approach to dynamic occlusions handling

```

// For each path, iterate over its segments,
// starting from the first one.
1 foreach Segment ∈ Path do
2   if not IntersectedByObjects(Segment) then
3     continue
4     // Compute intersection along segment
5     Hit ← TraceRay(Segment.origin, Segment.dir)
6     HitPos ← ComputeHitPos(Segment, Hit)
7     NextSeg ← Next(Segment, Path)
8     // Compute new outgoing direction
9     NextSeg.origin ← HitPos
10    NextSeg.dir ← NextSeg.dest − NextSeg.origin
11    Energy ← BRDF(Segment, NextSeg)
12    if not AreEnergiesClose(NextSeg.energy, Energy) then
13      Segment.dest ← HitPos
14      // Sample BRDF to generate new ray
15      return
16    else if AreClose(HitPos, Segment.dest) ∧
17      not IntersectedByObjects(NextSeg) ∧
18      not HitMovingObject(NextSeg.origin) ∧
19      not HitMovingObject(NextSeg.dest) then
20      // Skip visibility check between
21      // NextSeg.origin and NextSeg.dest
22      Segment ← NextSeg
23      continue
24    Segment.dest ← HitPos
25    // Visibility check for NextSeg will occur on
26    // the next iteration

```

- The number of segments in the path. (4 bits);
- Starting segment to retrace path from. (4 bits);
- Replace path. The path is retraced if the bit is set. (1 bit);
- Reuse light keeping light position and direction. (1 bit).

The representation of the different steps as seen in Figure 3 does not match 1:1 to our implementation. For example, we actually update the path origins and compute the DM_C in the same kernel, while the dynamic occlusions are tested right after that. The merging of the two kernels was done for performance reasons, in order to avoid reading from memory data that was recently written, and the two kernels were relatively small. Since the dynamic occlusions testing can not be done before the path origins are updated, it had to be moved after the computation of the DM_C .

For simplicity reasons, we generate new rays as soon as it has been decided we need to replace an existing ray. This means that ray generation is effectively done in multiple places: during the dynamic occlusions testing, when filling the DM_C and when processing the results from the tracing pass, if the maximum depth has not been reached yet.

Finally, when pruning extra paths, we end up modifying the number of paths found in the distribution map, while needing to

use the initial amount in the pruning probability (see Equation (1)). This can be achieved by modifying a copy of the distribution, thus using more memory, or by doing the update in two passes by first marking the pruned paths, and then editing the distribution map values. We are using the second approach in our implementation.

5 RESULTS

All presented results were rendered at a resolution of 1920×1080 on an NVIDIA Titan X (Pascal architecture, 12 GB of VRAM). The tracing of the photons was done using OptiX Prime 5.0.0 [Parker et al. 2010], whereas the path-reuse computations were implemented using CUDA 9.1 [Nickolls et al. 2008]. We compare our “naive” approach, presented in Section 3.2, to our error-based method, presented in Section 3.3 and to a baseline, which consists in not reusing any information from previous frames and re-tracing every single path each frame.

We tested our methods on different scenes:

Merry-go-round Conference, with a disc area light placed above the centre of the conference table, 3 scaling and rotating teapots placed on that table, around which 8 bunnies move as shown in Figure 6a);

Armadillo Conference scene with an armadillo moving from one door to the presenter stand, waiting there for a few seconds, then proceeding to the other door as shown in Figure 6b.

Villa a small torchlight, made of a disc-shaped area light, is moving within the kitchen of a house, indirectly lighting the living room as shown in Figure 6c.

We recorded the first 30 seconds of the rendering of each scene, for the baseline and our two methods; those videos can be found in the supplemental materials. The configurations used (number of paths, resolution of the DM, etc.) are the same as the ones mentioned in Figure 7. Note that the time displayed in the top-right corner in the videos corresponds to the *total frame time*, while Figure 7 and 8 both focus on only a few steps of the process, ignoring for example the time taken for splatting the photons (≥ 130 ms) as orthogonal to the reuse.

5.1 Performance

The breakdowns presented in Figure 8 uses the different categories presented in Figure 3, but with the modifications described in Section 4. So, for example, the “update path origins” time is included within the “compute DM_C ” time, as they are implemented within the same kernel.

Our two methods only differ in how they handle moving objects, but their handling of moving lights is the same. This explains why there is no differences between our two methods, neither in number of rays reused nor in tracing time, in Figure 7c.

Even our naive method for dynamic objects already significantly reduces the number of rays traced each frame, for example for the armadillo scene, it is reduced by $5\times$, as can be seen in Figure 7b. This does not translate into a $5\times$ decrease in the time taken by OptiX prime for tracing those rays, but into a $3\times$ decrease instead. This could come from more primary rays, proportionally, not being retraced, compared to secondary rays, which are more expensive, as well as not taking special care to maximise ray locality and

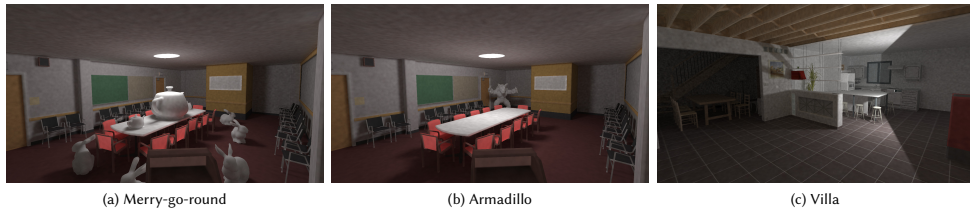


Fig. 6. Images of the scenes used in this paper.

Table 1. Memory-consumption (in MiB) breakdown when not reusing photons, reusing photons with moving lights and reusing photons with moving objects. In all scenarios, 5 millions paths containing each at most 7 photons were considered; those paths were traced from a single disc-shaped area light, which was associated to a 32^4 distribution map.

	No reuse	Reuse lights	Reuse obj.
Path information	-	19.07	19.07
Path origin pos.	-	57.22	57.22
Distribution maps	-	8.000	-
Pruned paths array	-	19.07	-
Sub-total	-	103.36	76.29
Photon map		1068	
Total	1068	1171	1144

coherency. Overall, our error-based method only slightly improves the number of rays reused, except when the armadillo gets close to the light source (around frame 250), where it retraces only half the number of rays compared to our naive method.

The merry-go-round scene reduces the effectiveness of ray reuse, as many primary rays will be hitting a moving object, instantly invalidating the whole path. Despite that, our naive method queries almost half as many rays as the baseline. Furthermore, our error-based approach reuses close to $1.5\times$ as many rays as our naive approach, as seen in Figure 7a.

Our different methods do add a small overhead compared to just re-tracing the paths every frame. This overhead includes updating the path’s origin, computing the DM_C and optimising it. On average the overhead is about 2.5 ms, compared to the average baseline time of 60 ms, as shown in Figure 8, and even including this overhead our method still leads to an average $4\times$ increase in performance.

5.2 Memory Consumption

In this section we present the amount of memory being used for reusing photons from previous frames. As reusing photons can be decoupled from the method used for rendering using the photon map, we do not discuss the memory used for the rendering method.

Path information is stored in a single 32-bit word, per path, as described in Section 4. This compactness does introduce some limitations, like being limited to at most 16 bounces, or to having at

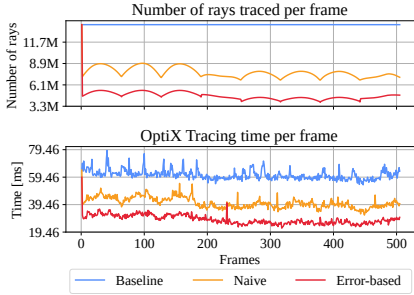
most 4 million cells in a distribution map, but those are not scenarios presented in this paper and were done in order to improve performance and reduce memory consumption. Those restrictions could be lifted by using more memory instead, without needing to change the algorithm.

For each path, we also store the position on the light from which it was emitted; this is only needed for area lights, as for point lights, it will always be the same position as the light itself. One could avoid having to store that information separately, by instead storing for each photon its incoming direction, scaled by the distance between it and its predecessor, and its position, allowing to recompute the origin point. However, this will make all photons larger, resulting in an increased memory consumption.

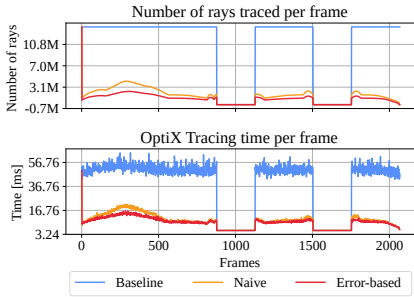
A single 32^4 DM is 4 MiB, but as each light gets two of them (the current one and the expected one), the number reported is 8 MiB. Note that DM_T could be compressed if memory consumption is an issue, as, depending on the representation used, multiple symmetries can be exploited. For example for a diffuse rectangular area light, all points on its surface will have the same outgoing directions profile, so only one set could be stored, bringing down the distribution map size from 4-D to 2-D. Also, if using an angular representation for the directions, the values obtained for the partitioning along θ are the same for all ϕ partitions, bringing the dimensionality further down to 1. DM_T can also be computed as needed, to avoid having to store it.

When we need to process all pruned paths, i.e. paths that were marked during the “prune paths” step (see Section 3.1), we could go over the path information attached to each path, and only process the ones marked. However this could result in blocks with only a couple of active threads using the GPU resources and preventing other blocks from running, whereas if combining all active threads into as few blocks as possible, they could all run simultaneously. So in order to achieve the latter, we maintain an array containing all pruned paths, and process from the start only those paths, at the cost of using more memory (a single 32-bit word per path).

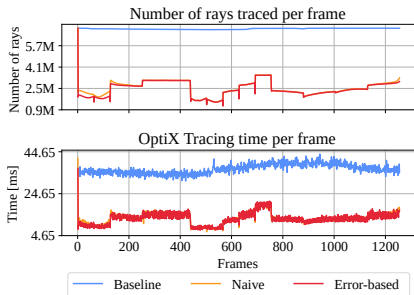
In cases where paths do not bounce up to the limit, our photon map design (described in Section 4) will be wasting some memory space. It is however quite simple and allows straightforward accesses to any photon of any path, and is quite efficient when processing all paths, at the same i th bounce, simultaneously.



(a) Merry-go-round

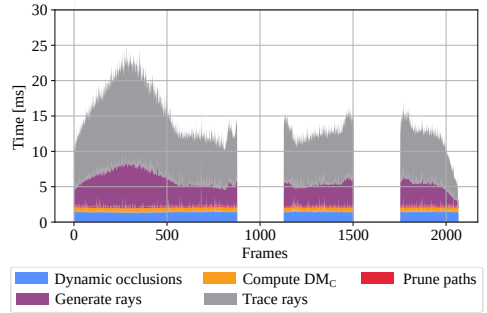


(b) Armadillo

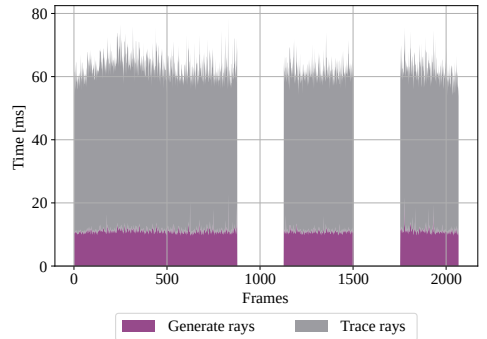


(c) Villa

Fig. 7. Tracing time and number of rays compared to the baseline, for the different scenes. The merry-go-round and armadillo scenes both used 2 million paths, whereas the villa has 1 million paths, but for all of them the paths contained at most 7 photons and the distribution map had a resolution of $8 \times 8 \times 64 \times 64$. For our error-based method, the energy threshold was set to 0.1%.



(a) Our error-based method



(b) Baseline

Fig. 8. Breakdowns of our method for reusing photons (top) and of the baseline (bottom), for the armadillo scene. In both cases, 2m paths were traced with a maximum of 7 bounces, and for our method, the distribution map had a resolution of $8 \times 8 \times 64 \times 64$ while a 0.1% error on the outgoing radiance was allowed on reused segments.

6 LIMITATIONS

Glossy surfaces If an intersection on a glossy surface is located on a static object and neither the incoming nor outgoing directions have changed, our method will be able to reuse those segments. However, if the above condition does not hold, then we might have to re-trace the outgoing ray, as even a small change in direction can lead to a large change in reflected energy.

Motion Blur For this to be correct we would need to detect occlusions in between frames.

7 CONCLUSION

Path tracing for indirect illumination requires a substantial amount of computation and in this paper we have shown how light transport paths can be reused temporally by verifying the path segments. In particular for moving lights we demonstrate that even though the light source moves, we can still reuse photon paths coming from area light sources. Furthermore when moving objects are present in the scene we demonstrate how paths can be brute force tested against dynamic objects in a relatively short amount of time compared to overall frame time. By using an error threshold for path verification we further demonstrate that path reuse can be improved and the number of retraced rays per frame can be significantly reduced. Path verification is particularly important for scenes with long paths where reuse has an even greater impact on frame time.

Since our technique is focused on verifying the validity of paths, it would also be applicable to camera paths for path tracing methods. For path tracing the distribution map would be located on the near plane of the camera and the 2D distribution map should behave similarly to that of a spotlight.

ACKNOWLEDGMENTS

In the villa scene, the “flash light” [naves 2017] model is courtesy of naves and the “maison à ossature bois” [Envisioneer 2015] model is courtesy of ADoc Envisioneer, both under CC Attribution 4.0. Conference, the bunny, the teapot and Armadillo are taken from Morgan McGuire’s Computer Graphics Archive [2017]. Pierre and Michael are sponsored by the Swedish Research Council under grant № 2014-5191.

REFERENCES

- Philippe Bekaert, Mateu Sbert, and John Halton. 2002. Accelerating Path Tracing by Re-using Paths. In *Proceedings of the 13th Eurographics Workshop on Rendering (EGRW)*. 125–134.
- Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Transactions on Graphics* 36, 4, Article 98 (July 2017), 12 pages. <https://doi.org/10.1145/3072959.3073601>
- Alessandro Dal Corso, Marco Salvi, Craig Kolb, Jeppe Revall Firsvad, Aaron Lefohn, and David Luebke. 2017. Interactive Stable Ray Tracing. In *Proceedings of High Performance Graphics (HPG '17)*, Vlastimil Havran and Karthik Vaidyanathan (Eds.). Association for Computing Machinery, New York, NY, USA, 1–20. <https://doi.org/10.1145/3105762.3105769>
- Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. 2011. Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)* 30, 7 (Sept. 2011), 207–207.
- Kirill Dmitriev, Stefan Bräbe, Karol Myszkowski, and Hans-Peter Seidel. 2002. Interactive Global Illumination Using Selective Photon Tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering (EGRW '02)*, Paul Debevec and Simon Gibson (Eds.). The Eurographics Association, Goslar, DEU, 25–36.
- ADoc Envisioneer. 2015. Maison à Ossature Bois. <https://sketchfab.com/models/67e4fb7f01942e0a162be0173bb72b>
- Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. 2017. An Efficient Denoising Algorithm for Global Illumination. In *Proceedings of High Performance Graphics (Los Angeles, California, USA)*. ACM, New York, NY, USA. <https://doi.org/10.1145/3105762.3105774>
- Morgan McGuire. 2017. Computer Graphics Archive. <https://casual-effects.com/data>
- Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. 2017. Real-time Global Illumination Using Precomputed Light Field Probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '17)*. ACM, New York, NY, USA, Article 2, 11 pages. <https://doi.org/10.1145/3023368.3023378>
- Gary McTaggart. 2004. Half-Life® 2/Valve Source™ Shading. In *Direct3D Tutorial (GDC)*.
- Pierre Moreau, Erik Sintorn, Viktor Kämpe, Ulf Assarsson, and Michael Doggett. 2016. Photon Splatting Using a View-Sample Cluster Hierarchy. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, Ulf Assarsson and Warren Hunt (Eds.). The Eurographics Association. <https://doi.org/10.2312/hpg.20161194>
- naves. 2017. Flash Light. <https://sketchfab.com/models/123a79642c2646d8b31557682fea84a>
- John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. <https://doi.org/10.1145/1365490.1365500>
- Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics* 29, 4, Article 66 (July 2010), 13 pages.
- Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. 2012. The State of the Art in Interactive Global Illumination. *Comput. Graph. Forum* 31, 1, Article 1 (Feb. 2012), 29 pages. <https://doi.org/10.1111/j.1467-8659.2012.02093.x>
- Tobias Ritschel, Thorsten Grosch, and Seidel Hans-Peter. 2009. Approximating dynamic global illumination in image space. In *Proc. ACM 13D*.
- Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination. In *Proceedings of High Performance Graphics (Los Angeles, California) (HPG '17)*. ACM, New York, NY, USA, Article 2, 12 pages. <https://doi.org/10.1145/3105762.3105770>
- Ari Silvenoinen and Jaakko Lehtinen. 2017. Real-time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 36, 6 (Nov. 2017), 230:1–230:13. <https://doi.org/10.1145/3130800.3130852>
- Peter-Pike Sloan, Jan Kautz, and John Snyder. 2002. Precomputed Radiance Transfer for Real-time Rendering in Dynamic, Low-frequency Lighting Environments. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 21, 3 (July 2002), 527–536. <https://doi.org/10.1145/566654.566612>
- Tiago Sousa, Nikolay Kasyan, and Nicolas Schulz. 2011. Secrets of CryENGINE 3 Graphics Technology. In *Advances in Real-Time Rendering in 3D Graphics and Games, SIGGRAPH Tutorial*. <http://www.crytek.com/cryengine/presentations/secrets-of-cryengine-3-graphics-technology>
- Maayan Weiss and Thorsten Grosch. 2012. Stochastic Progressive Photon Mapping for Dynamic Scenes. *Computer Graphics Forum* 31, 2pt4, Article 1 (May 2012), 8 pages.

Paper III



Importance Sampling of Many Lights on the GPU

Pierre Moreau^{1,2} and Petrik Clarberg¹

¹NVIDIA

²Lund University

ABSTRACT

The introduction of standardized APIs for ray tracing, together with hardware acceleration, opens up possibilities for physically based lighting in real-time rendering. Light importance sampling is one of the fundamental operations in light transport simulations, applicable to both direct and indirect illumination. This chapter describes a bounding volume hierarchy data structure and associated sampling methods to accelerate importance sampling of local light sources. The work is based on recently published methods for light sampling in production rendering, but it is evaluated in a real-time implementation using Microsoft DirectX Raytracing.

18.1 INTRODUCTION

A realistic scene may contain hundreds of thousands of light sources. The accurate simulation of the light and shadows that they cast is one of the most important factors for realism in computer graphics. Traditional real-time applications with rasterized shadow maps have been practically limited to use a handful of carefully selected dynamic lights. Ray tracing allows more flexibility, as we can trace shadow rays to different sampled lights at each pixel.

Mathematically speaking, the best way to select those samples is to pick lights with a probability in proportion to each light's contribution. However, the contribution varies spatially and depends on the local surface properties and visibility. Hence, it is challenging to find a single global probability density function (PDF) that works well everywhere.

The solution that we explore in this chapter is to use a hierarchical acceleration structure built over the light sources to guide the sampling [11, 22]. Each node in the data structure represents a cluster of lights. The idea is to traverse the tree from top to bottom, at each level estimating how much each cluster

contributes, and to choose which path through the tree to take based on random decisions at each level. Figure 18-1 illustrates these concepts. This means that lights are chosen approximately proportional to their contributions, but without having to explicitly compute and store the PDF at each shading point. The performance of the technique ultimately depends on how accurately we manage to estimate the contributions. In practice, the pertinence of a light or a cluster of lights, depends on its:

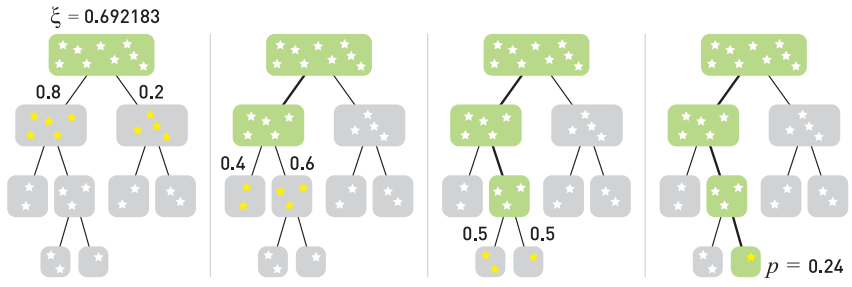


Figure 18-1. All the light sources in the scene are organized in a hierarchy. Given a shading point X , we start at the root and proceed down the hierarchy. At each level, the importance of each immediate child with respect to X is estimated by a probability. Then, a uniform random number ξ decides the path through the tree, and at the leaf we find which light to sample. In the end, more important lights have a higher probability of being sampled.

- > *Flux:* The more powerful a light is, the more it will contribute.
- > *Distance to the shading point:* The further away a light lies, the smaller the solid angle it subtends, resulting in less energy reaching the shading point.
- > *Orientation:* A light source may not emit in all directions, nor do so uniformly.
- > *Visibility:* Fully occluded light sources do not contribute.
- > *BRDF at the shading point:* Lights located in the direction of the BRDF's main peaks will have a larger fraction of their energy reflected.

A key advantage of light importance sampling is that it is independent of the number and type of lights, and hence scenes can have many more lights than we can afford to trace shadow rays to and large textured area lights can be seamlessly supported. Since the probability distributions are computed at runtime, scenes can be fully dynamic and have complex lighting setups. With recent advances in denoising, this holds promise to reduce rendering time, while allowing more artistic freedom and more realistic results.

In the following, we discuss light importance sampling in more detail and present a real-time implementation that uses a bounding volume hierarchy (BVH) over the lights. The method is implemented using the Microsoft DirectX Raytracing (DXR) API, and source code is available.

18.2 REVIEW OF PREVIOUS ALGORITHMS

With the transition to path tracing in production rendering [21, 31], the visibility sampling is solved by tracing shadow rays toward sampled points on the light sources. When a shadow ray does not hit anything on its way from a shading point to the light, the point is deemed to be lit. By averaging over many such samples over the surfaces of the lights, a good approximation of the lighting is achieved. The approximation converges to ground truth as more samples are taken. However, with more than a handful of light sources, exhaustive sampling is not a viable strategy, not even in production rendering.

To handle the complexity of dynamic lighting with many lights, most techniques generally rely on building some form of spatial acceleration structure over the lights, which is then used to accelerate rendering by either culling, approximating, or importance-sampling the lights.

18.2.1 REAL-TIME LIGHT CULLING

Game engines have transitioned to use mostly physically based materials and light sources specified in physical units [19, 23]. However, for performance reasons and due to the limitations of the rasterization pipeline, only a few point-like light sources can be rendered in real time with shadow maps. The cost per light is high and the performance scales linearly with the number of lights. For area lights, the unshadowed contribution can be computed using linearly transformed cosines [17], but the problem of evaluating visibility remains.

To reduce the number of lights that need to be considered, it is common to artificially limit the influence region of individual lights, for example, by using an approximate quadratic falloff that goes to zero at some distance. By careful placement and tweaking of the light parameters, the number of lights that affect any given point can be limited.

Tiled shading [2, 28] works by binning such lights into screen-space tiles, where the depth bounds of the tiles effectively reduce the number of lights that need to be processed when shading each tile. Modern variants improve culling rates by splitting frusta in depth (2.5D culling) [15], by clustering shading points or lights [29, 30], or by using per-tile light trees [27].

A drawback of these culling methods is that the acceleration structure is in screen space. Another drawback is that the required clamped light ranges can introduce noticeable darkening. This is particularly noticeable in cases where many dim lights add up to a significant contribution, such as Christmas tree lights or indoor office illumination. To address this, Tokuyoshi and Harada [40] propose using stochastic light ranges to randomly reject unimportant lights rather than assigning fixed ranges. They also show a proof-of-concept of the technique applied to path tracing using a bounding sphere hierarchy over the light sources.

18.2.2 MANY-LIGHT ALGORITHMS

Virtual point lights (VPLs) [20] have long been used to approximate global illumination. The idea is to trace photons from the light sources and deposit VPLs at path vertices, which are then used to approximate the indirect illumination. VPL methods are conceptually similar to importance sampling methods for many lights. The lights are clustered into nodes in a tree, and during traversal estimated contributions are computed. The main difference is that, for importance sampling, the estimations are used to compute light selection probabilities rather than directly to approximate the lighting.

For example, *lightcuts* [44, 45] accelerate the rendering with millions of VPLs by traversing the tree per shading point and computing error bounds on the estimated contributions. The algorithm chooses to use a cluster of VPLs directly as a light source, avoiding subdivision to finer clusters or individual VPLs, when the error is sufficiently small. We refer to the survey by Dachsbacher et al. [12] for a good overview of these and other many-light techniques. See also the overview of global illumination algorithms by Christensen and Jarosz [8].

18.2.3 LIGHT IMPORTANCE SAMPLING

In early work on accelerating ray tracing with many lights, the lights are sorted according to contribution and only the ones above a threshold are shadow tested [46]. The contribution of the remaining lights is then added based on a statistical estimate of their visibility.

Shirley et al. [37] describe importance sampling for various types of light sources. They classify lights as bright or dim by comparing their estimated contributions to a user-defined threshold. To sample from multiple lights, they use an octree that is hierarchically subdivided until the number of bright lights is sufficiently small. The contribution of an octree cell is estimated by evaluating the contribution at a large number of points on the cell's boundary. Zimmerman and Shirley [47] use a uniform spatial subdivision instead and include an estimated visibility in the cells.

For real-time ray tracing with many lights, Schmittler et al. [36] restrict the influence region of lights and use a k -d tree to quickly locate the lights that affect each point. Bikker takes a similar approach in the Arauna ray tracer [5, 6], but it uses a BVH with spherical nodes to more tightly bound the light volumes. Shading is done Whitted-style by evaluating all contributing lights. These methods suffer from bias as the light contributions are cut off, but that may potentially be alleviated with stochastic light ranges as mentioned earlier [40].

In the Brigade real-time path tracer, Bikker [6] uses *resampled importance sampling* [39]. A first set of lights is selected based on a location-invariant probability density function, and then this set is resampled by more accurately estimating the contributions using the BRDF and distances to pick one important light. In this approach, there is no hierarchical data structure.

The Iray rendering system [22] uses a hierarchical light importance sampling scheme. Iray works with triangles exclusively and assigns a single flux (power) value per triangle. A BVH is built over the triangular lights and traversed probabilistically, at each node computing the estimated contribution of each subtree. The system encodes directional information at each node by dividing the unit sphere into a small number of regions and storing one representative flux value per region. Estimated flux from BVH nodes is computed based on the distance to the center of the node.

Conty Estevez and Kulla [11] take a similar approach for cinematic rendering. They use a 4-wide BVH that also includes analytic light shapes, and the lights are clustered in world space including orientation by using bounding cones. In the traversal, they probabilistically select which branch to traverse based on a single uniform random number. The number is rescaled to the unit range at each step, which preserves stratification properties (the same technique is used in hierarchical sample warping [9]). To reduce the problem of poor estimations for large nodes, they use a metric for adaptively splitting such nodes during traversal. Our real-time implementation is based on their technique, with some simplifications.

18.3 FOUNDATIONS

In this section, we will first review the foundations of physically based lighting and importance sampling, before diving into the technical details of our real-time implementation.

18.3.1 LIGHTING INTEGRALS

The radiance L_o leaving a point X on a surface in viewing direction \mathbf{v} is the sum of emitted radiance L_e and reflected radiance L_r , under the geometric optics approximation described by [18]:

$$L_o(X, \mathbf{v}) = L_e(X, \mathbf{v}) + L_r(X, \mathbf{v}), \tag{1}$$

$$\text{where } L_r(X, \mathbf{v}) = \int_{\Omega} f(X, \mathbf{v}, \mathbf{l}) L_i(X, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l}) d\omega \tag{2}$$

and where f is the BRDF and L_i is the incident radiance arriving from a direction \mathbf{l} . In the following, we will drop the X from the notation when we speak about a specific point. Also, let the notation $L[X \leftarrow Y]$ denote the radiance emitted from a point Y in the direction toward a point X .

In this chapter, we are primarily interested in the case where L_i comes from a potentially large set of local light sources placed within the scene. The algorithm can, however, be combined with other sampling strategies for handling distant light sources, such as the sun and sky.

The integral over the hemisphere can be rewritten as an integral over all the surfaces of the light sources. The relationship between solid angle and surface area is illustrated in Figure 18-2. In fact, a small patch dA at a point Y on a light source covers a solid angle

$$d\omega = \frac{|\mathbf{n}_y \cdot -\mathbf{l}|}{\|X - Y\|^2} dA, \tag{3}$$

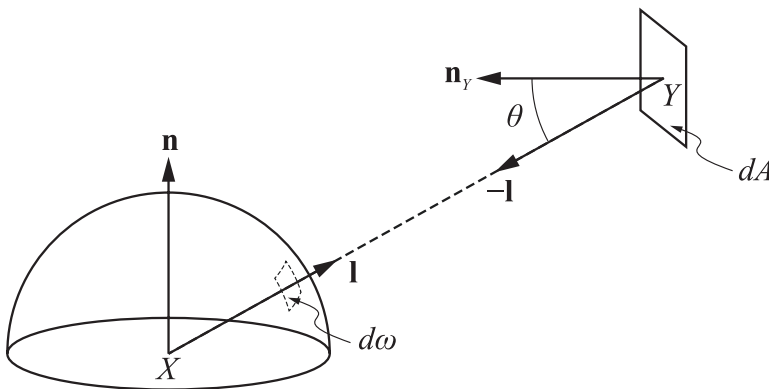


Figure 18-2. The differential solid angle $d\omega$ of a surface patch dA at a point Y on a light source is a function of its distance $\|X - Y\|$ and the angle $\cos\theta = |\mathbf{n}_y \cdot -\mathbf{l}|$ at which it is viewed.

i.e., there is an inverse square falloff by distance and a dot product between the light's normal \mathbf{n}_y and the emitted light direction $-\mathbf{l}$. Note that in our implementation, light sources may be single-sided or double-sided emitters. For single-sided lights, we set the emitted radiance $L(X \leftarrow Y) = 0$ if $(\mathbf{n}_y \cdot -\mathbf{l}) \leq 0$.

We also need to know the visibility between our shading point X and the point Y on the light source, formally expressed as

$$v(X \leftrightarrow Y) = \begin{cases} 1 & \text{if } X \text{ and } Y \text{ are mutually visible,} \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

In practice, we evaluate v by tracing shadow rays from X in direction \mathbf{l} , with the ray's maximum distance $t_{\max} = \|X - Y\|$. Note that to avoid self-intersections due to numerical issues, the ray origin needs to be offset and the ray shortened slightly using epsilons. See Chapter 6 for details.

Now, assuming that there are m light sources in the scene, the reflected radiance in Equation 2 can be written as

$$L_r(X, \mathbf{v}) = \sum_{i=1}^m L_{r,i}(X, \mathbf{v}), \quad \text{where} \quad (5)$$

$$L_{r,i}(X, \mathbf{v}) = \int_{\Omega} f(X, \mathbf{v}, \mathbf{l}) L_i(X \leftarrow Y) v(X \leftrightarrow Y) \max(\mathbf{n} \cdot \mathbf{l}, 0) \frac{|\mathbf{n}_y \cdot -\mathbf{l}|}{\|X - Y\|^2} dA_i. \quad (6)$$

That is, L_r is the sum of the reflected light from each individual light $i = \{1, \dots, m\}$. Note that we clamp $\mathbf{n} \cdot \mathbf{l}$ because light from points backfacing to the shading point cannot contribute. The complexity is linear in the number of lights m , which may become expensive when m is large. This leads us to the next topic.

18.3.2 IMPORTANCE SAMPLING

As discussed in Section 18.2, there are two fundamentally different ways to reduce the cost of Equation 5. One method is to limit the influence regions of lights, and thereby reduce m . The other method is to sample a small subset of lights $n \ll m$. This can be done in such a way that the result is *consistent*, i.e., it converges to the ground truth as n grows.

18.3.2.1 MONTE CARLO METHOD

Let Z be a discrete random variable with values $z \in \{1, \dots, m\}$. The probability that Z is equal to some value z is described by the discrete PDF $p(z) = P(Z = z)$, where

$\sum p(z) = 1$. For example, if all values are equally probable, then $p(z) = \frac{1}{m}$. If we

have a function $g(Z)$ of a random variable, its expected value is

$$\mathbb{E}[g(Z)] = \sum_{z \in Z} g(z)p(z), \quad (7)$$

i.e., each possible outcome is weighted by how probable it is. Now, if we take n random samples $\{z_1, \dots, z_n\}$ from Z , we get the n -sample *Monte Carlo estimate* $\tilde{g}_n(z)$ of $\mathbb{E}[g(Z)]$ as follows:

$$\tilde{g}_n(z) = \frac{1}{n} \sum_{j=1}^n g(z_j). \quad (8)$$

In other words, the expectation can be estimated by taking the mean of random samples of the function. We can also speak of the corresponding Monte Carlo *estimator* $\tilde{g}_n(Z)$, which is the mean of the function of the n independent and identically distributed random variables $\{Z_1, \dots, Z_n\}$. It is easy to show that

$\mathbb{E}[\tilde{g}_n(Z)] = \mathbb{E}[g(Z)]$, i.e., the estimator gives us the correct value.

Since we are taking random samples, the estimator $\tilde{g}_n(Z)$ will have some variance. As discussed in Chapter 15, the variance decreases linearly with n :

$$\text{Var}[\tilde{g}_n(Z)] = \frac{1}{n} \text{Var}[g(Z)]. \quad (9)$$

These properties show that the Monte Carlo estimator is consistent. In the limit, when we have infinitely many samples, the variance is zero and it has converged to the correct expected value.

To make this useful, note that almost any problem can be recast as an expectation. We thus have a consistent way of estimating the solution based on random samples of the function.

18.3.2.2 LIGHT SELECTION IMPORTANCE SAMPLING

In our case, we are interested in evaluating the sum of light reflected from all the light sources (Equation 5). This sum can be expressed as an expectation (cf., Equation 7) as follows:

$$L_r(X, \mathbf{v}) = \sum_{i=1}^m L_{r,i}(X, \mathbf{v}) = \sum_{i=1}^m \frac{L_{r,i}(X, \mathbf{v})}{P(Z=i)} P(Z=i) = \mathbb{E} \left[\frac{L_{r,Z}(X, \mathbf{v})}{p(Z)} \right]. \quad (10)$$

Following Equation 8, the Monte Carlo estimate \tilde{L}_r of the reflected light from all light sources is therefore

$$\tilde{L}_r(X, \mathbf{v}) = \frac{1}{n} \sum_{j=1}^n \frac{L_{r,z_j}(X, \mathbf{v})}{p(z_j)}, \quad (11)$$

that is, we sum the contribution from a randomly selected set of lights $\{z_1, \dots, z_n\}$, divided by the probability of selecting each light. This estimator is always consistent, independent of how few samples n we take. However, the more samples we take, the smaller the variance of the estimator will be.

Note that nothing discussed so far makes any assumptions on the distribution of the random variable Z . The only requirement is that $p(z) > 0$ for all lights where $L_{r,z} > 0$, otherwise we would risk ignoring the contribution from some lights. It can be shown that the variance is minimized when $p(z) \propto L_{r,z}(X, \mathbf{v})$ [32, 38]. We will not go into the details here, but when the probability density function is exactly proportional to the function that we are sampling, the summation in the Monte Carlo estimator reduces to a sum of constant terms. In that case the estimator has zero variance.

In practice, this is not achievable because $L_{r,z}$ is unknown for a given shading point, but we should aim for selecting lights with a probability as close as possible to their relative contribution to the shading point. In Section 18.4, we will look at how $p(z)$ is computed.

18.3.2.3 LIGHT SOURCE SAMPLING

To estimate the reflected radiance using Equation 11, we also need to evaluate the integral $L_{r,z_j}(X, \mathbf{v})$ for the randomly selected set of lights. The expression in Equation 6 is an integral over the surface of the light that involves both BRDF and visibility terms. In graphics, this is not practical to evaluate analytically. Therefore, we again resort to Monte Carlo integration.

The surface of the light source is sampled uniformly with s samples $\{Y_1, \dots, Y_s\}$. For triangle mesh lights, each light is a triangle, which means that we pick points uniformly on the triangle using standard techniques [32] (see Chapter 16). The probability density function for the samples on a triangle i is $p(Y) = \frac{1}{A_i}$, where A_i is the area of the triangle. The integral over the light is then evaluated using the Monte Carlo estimate

$$\tilde{L}_{r,i}(X, \mathbf{v}) = \frac{A_i}{s} \sum_{k=1}^s f(X, \mathbf{v}, \mathbf{l}_k) L_i(X \leftarrow Y_k) v(X \leftrightarrow Y_k) \max(\mathbf{n} \cdot \mathbf{l}_k, 0) \frac{|\mathbf{n}_{Y_k} \cdot -\mathbf{l}_k|}{\|X - Y_k\|^2}. \quad (12)$$

In the current implementation, $s = 1$ as we trace a single shadow ray for each of the n sampled light sources, and $\mathbf{n}_{Y_k} = \mathbf{n}_i$ since we use the geometric normal of the light source when evaluating its emitted radiance. Smooth normals and normal mapping are disabled by default for performance reasons, because they often have negligible impact on the light distribution.

18.3.3 RAY TRACING OF LIGHTS

In real-time applications, a common rendering optimization is to separate the geometric representation from the actual light-emitting shape. For example, a light bulb can be represented by a point light or small analytic sphere light, while the visible light bulb is drawn as a more complex triangle mesh.

In this case, it is important that the emissive property of the light geometry matches the intensity of the actual emitter. Otherwise, there will be a perceptual difference between how bright a light appears in direct view and how much light it casts into the scene. Note that a light source is often specified in photometric units in terms of its luminous flux (lumen), while the emissive intensity of an area light is given in luminance (cd/m^2). Accurate conversion from flux to luminance therefore needs to take the surface area of the light's geometry into account. Before rendering, these photometric units are finally converted to the radiometric quantities that we use (flux and radiance).

Another consideration is that when tracing shadow rays toward an emitter, we do not want to inadvertently hit the mesh representing the light source and count the emitter as occluded. The geometric representation must therefore be invisible to shadow rays, but visible for other rays. The Microsoft DirectX Raytracing API allows control of this behavior via the `InstanceMask` attribute on the acceleration structure and by the `InstanceInclusionMask` parameter to `TraceRay`.

For *multiple importance sampling* (MIS) [41], which is an important variance reduction technique, we must be able to evaluate light sampling probabilities given samples generated by other sampling strategies. For example, if we draw a sample over the hemisphere using BRDF importance sampling that hits a light source after traversal, we compute its probability had the sample been generated with light importance sampling. Based on this probability together with the BRDF sampling probability, a new weight for the sample can be computed using, for example, the power heuristic [41] to reduce the overall variance.

A practical consideration for MIS is that if the emitters are represented by analytic shapes, we cannot use hardware-accelerated triangle tests to search for the light source in a given direction. An alternative is to use custom intersection shaders to compute the intersections between rays and emitter shapes. This has not yet been implemented in our sample code. Instead, we always use the mesh itself as the light emitter, i.e., each emissive triangle is treated as a light source.

18.4 ALGORITHM

In the following, we describe the main steps of our implementation of light importance sampling. The description is organized by the frequency at which operations occur. We start with the preprocessing step that can happen at asset-creation time, which is followed by the construction and updating of the light data structure that runs once per frame. Then, the sampling is described, which is executed once per light sample.

18.4.1 LIGHT PREPROCESSING

For mesh lights, we precompute a single flux value Φ_i per triangle i as a preprocess, similar to Iray [22]. The flux is the total radiant power emitted by the triangle. For diffuse emitters, the flux is

$$\Phi_i = \iint_{\Omega} L_i(X)(\mathbf{n}_i \cdot \omega) d\omega dA_i, \quad (13)$$

where $L_i(X)$ is the emitted radiance at position X on the light's surface. For non-textured emitters, the flux is thus simply $\Phi_i = \pi A_i L_i$, where L_i is the constant radiance of the material and A_i is the triangle's area. The factor π comes from the integral of the cosine term over the hemisphere. To handle textured emitters, which in our experience are far more common than untextured ones, we evaluate Equation 13 as a preprocess at load time.

To integrate the radiance, we rasterize all emissive triangles in *texture space*. The triangles are scaled and rotated so that each pixel represents exactly one texel at

the largest mip level. The integral is then computed by loading the radiance for the corresponding texel in the pixel shader and by accumulating its value atomically. We also count the number of texels and divide by that number at the end.

The only side effect of the pixel shader is atomic additions to a buffer of per-triangle values. Due to the current lack of floating-point atomics in DirectX 12, we use an NVIDIA extension via NVAPI [26] to do floating-point atomic addition.

Since the pixel shader has no render target bound (i.e., it is a `void` pixel shader), we can make the viewport arbitrarily large within the API limits, without worrying about memory consumption. The vertex shader loads the UV texture coordinates from memory and places the triangle at an appropriate coordinate in texture space so that it is always within the viewport. For example, if texture wrapping is enabled, the triangle is rasterized at pixel coordinates

$$(x, y) = (u - \lfloor u \rfloor, v - \lfloor v \rfloor) \cdot (w, h), \quad (14)$$

where w, h are the dimensions of the largest mip level of the emissive texture. With this transform, the triangle is always in view, independent of the magnitude of its (pre-wrapped) UV coordinates.

We currently rasterize the triangle using one sample per pixel, and hence only accumulate texels whose centers are covered. Tiny triangles that do not cover any texels are assigned a default nonzero flux to ensure convergence. Multisampling, or conservative rasterization with analytic coverage computations in the pixel shader, can be used to improve accuracy of the computed flux values.

All triangles with $\Phi_e = 0$ are excluded from further processing. Culling of zero flux triangles is an important practical optimization. In several example scenes, the majority of the emissive triangles lie in black regions of the emissive textures. This is not surprising, as often the emissiveness is painted into larger textures, rather than splitting the mesh into emissive and non-emissive meshes with separate materials.

18.4.2 ACCELERATION STRUCTURE

We are using a similar acceleration structure as Conty Estevez and Kulla [11], that is, a bounding volume hierarchy [10, 33] built from top to bottom using binning [43]. Our implementation uses a binary BVH, meaning that each node has two children. In some cases, a wider branching factor may be beneficial.

We will briefly introduce how binning works, before presenting different existing heuristics used during the building process, as well as minor variants thereof.

18.4.2.1 BUILDING THE BVH

When building a binary BVH from top to bottom, the quality and speed at which the tree is built depends on how the triangles are split between the left and right children at each node. Analyzing all the potential split locations will yield the best results, but this will also be slow and is not suitable for real-time applications.

The approach taken by Wald [43] consists of uniformly partitioning the space at each node into bins and then running the split analysis on those bins only. This implies that the more bins one has, the higher the quality of the generated tree will be, but the tree will also be more costly to build.

18.4.2.2 LIGHT ORIENTATION CONE

To help take into account the orientation of the different light sources, Conty Estevez and Kulla [11] store a light orientation cone in each node. This cone is made of an axis and two angles, θ_o and θ_e : the former bounds the normals of all emitters found within the node, whereas the latter bounds the set of directions in which light gets emitted (around each normal).

For example, a single-sided emissive triangle would have $\theta_o = 0$ (there is only one normal) and $\theta_e = \frac{\pi}{2}$ (it emits light over the whole hemisphere). Alternatively, an emissive sphere would have $\theta_o = \pi$ (it has normals pointing in all directions) and $\theta_e = \frac{\pi}{2}$, as around each normal, light is still only emitted over the whole hemisphere; θ_e will often be $\frac{\pi}{2}$, except for lights with a directional emission profile or for spotlights, where it will be equal to the spotlight's cone angle.

When computing the cone for a parent node, its θ_o will be computed such that it encompasses all the normals found in its children, whereas θ_e is simply computed as the maximum of each child's θ_e .

18.4.2.3 DEFINING THE SPLIT PLANE

As mentioned earlier, an axis-aligned split plane has to be computed to split the set of lights into two subsets, one for each child. This is usually achieved by computing a cost metric for each possible split and picking the one with the lowest cost. In the context of a binned BVH, we tested the *surface area heuristic* (SAH) (introduced by Goldsmith and Salmon [14] and formalized by MacDonald and Booth [24]) and the *surface area orientation heuristic* (SAOH) [11], as well as different variants of those two methods.

For all the variants presented below, the binning performed while building the BVH can be done either on the largest axis only (of a node's axis-aligned bounding box (AABB)) or on all three axes and the split with the lowest cost is selected. Only considering the largest axis will result in lower build time but also lower tree quality, especially for the variants taking the light orientations into account. More details on those trade-offs can be found in Section 18.5.

SAH The SAH focuses on the surface area of the AABB of the resulting children as well as on the number of lights that they contain. If we define the left child as $L = \cup_{j=0}^i \text{bin}_j$ and the right child as $R = \cup_{j=i+1}^k \text{bin}_j$, where k is the number of bins and $i \in [0, k - 1]$, the cost for the split creating L and R as children is

$$\text{cost}(L, R) = \frac{n(L)a(L) + n(R)a(R)}{n(L \cup R)a(L \cup R)}, \quad (15)$$

where $n(C)$ and $a(C)$ return the number of lights and the surface area of a potential child node C , respectively.

SAOH The SAOH is based on the SAH and includes two additional weights: one based on the bounding cone around the directions in which the lights emit light, and another based on the flux emitted by the resulting clusters. The cost metric is

$$\text{cost}(L, R, s) = k_r(s) \frac{\Phi(L)a(L)M_\Omega(L) + \Phi(R)a(R)M_\Omega(R)}{a(L \cup R)M_\Omega(L \cup R)}, \quad (16)$$

where s is the axis on which the split is occurring, $k_r(s) = \text{length}_{\max} / \text{length}_s$ is used to prevent thin boxes, and M_Ω is an orientation measure [11].

VH The *volume heuristic* (VH) is based on the SAH and replaces the surface area measure $a(C)$ in Equation 15 by the volume $v(C)$ of a node C 's AABB.

VOH The *volume orientation heuristic* (VOH) similarly replaces the surface area measure in the SAOH (Equation 16) by the volume measure.

18.4.3 LIGHT IMPORTANCE SAMPLING

We now look at how the lights are actually sampled based on the acceleration structure described in the previous section. First, the light BVH is probabilistically traversed in order to select a single light source, and then a light sample is generated on the surface of that light (if it is an area light). See Figure 18-1.

18.4.3.1 PROBABILISTIC BVH TRAVERSAL

When traversing the acceleration data structure, we want to select the node that will lead us to the lights that contribute the most to the current shading point, with a probability for each light that is proportional to its contribution. As mentioned in Section 18.4.2, the contribution depends on many parameters. We will use either approximations or the exact value for each parameter, and we will try different combinations to optimize quality versus performance.

Distance This parameter is computed as the distance between the shading point and the center of the AABB of the node being considered. This favors nodes that are close to the shading point (and by extension lights that are close), if the node has a small AABB. However, in the first levels of the BVH, the nodes have large AABBs that contain most of the scene, giving a poor approximation of the actual distance between the shading point and some of the lights contained within that node.

Light Flux The flux of a node is computed as the sum of the flux emitted by all light sources contained within that node. This is actually precomputed when building the BVH for performance reasons; if some light sources have changing flux values over time, the precomputation will not be an issue because the BVH will have to be rebuilt anyway since the flux is also used for guiding the building step.

Light Orientation The selection so far does not take into consideration the orientation of the light source, which could give as much weight to a light source that is shining directly upon the shading point as to another light source that is backfacing. To that end, Conty Estevez and Kulla [11] introduced an additional term to a node's importance function that conservatively estimates the angle between the light normal and direction from the node's AABB center to the shading point.

Light Visibility To avoid considering lights that are located below the horizon of a shading point, we use the clamped $\mathbf{n} \cdot \mathbf{l}$ term in the importance function of each node. Note that Conty Estevez and Kulla [11] use this clamped term, multiplied by the surface's albedo, as an approximation to the diffuse BRDF, which will achieve the same effect of discarding lights that are beneath the horizon of the shading point.

Node Importance Using the different parameters just defined, the importance function given a shading point X and a child node C is defined as

$$\text{importance}(X, C) = \frac{\Phi(C) |\cos \theta'_r|}{\|X - C\|^2} \times \begin{cases} \cos \theta' & \text{if } \theta' < \theta_e, \\ 0 & \text{otherwise,} \end{cases} \quad (17)$$

where $\|X - C\|$ is the distance between shading point X and the center of the AABB of C , $\theta'_i = \max(0, \theta_i - \theta_u)$, and $\theta' = \max(0, \theta - \theta_o - \theta_u)$. The angles θ_e and θ_o come from the light orientation cone of node C . The angle θ is measured between the light orientation cone's axis and the vector from the center of C to X . Finally, θ_i is the incident angle and θ_u the uncertainty angle; these can all be found in Figure 18-3.

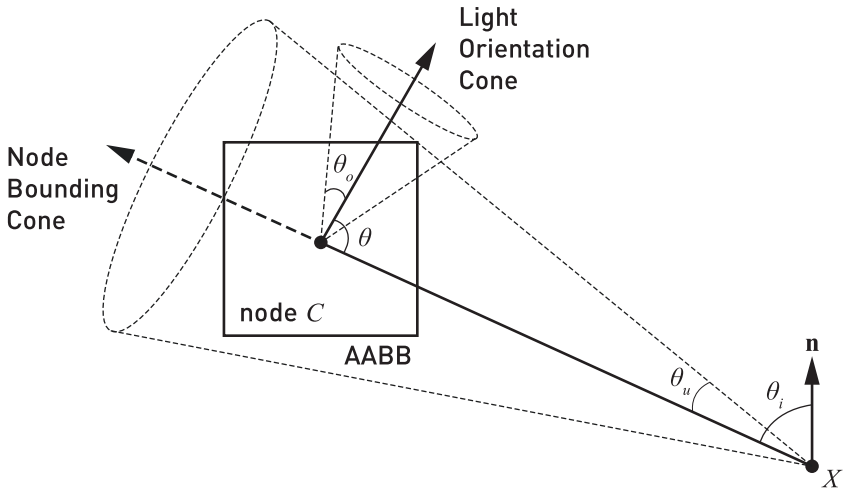


Figure 18-3. Description of the geometry used for computing the importance of a child node C as seen from a shading point X . In Figure 18-1, the importance is computed twice at each step in the traversal, once for each child. The angle θ_u , and the axis from X to the center of the AABB represent the smallest bounding cone containing the whole node and are used to compute conservative lower bounds on θ_i and θ .

18.4.3.2 RANDOM NUMBER USAGE

A single uniform random number is used to decide whether to take the left or the right branch. The number is then rescaled and used for the next level. This technique preserves stratification (cf., hierarchical sample warping [9]) while also avoiding the cost of generating new random numbers at every level of the hierarchy. The rescaling of a random number ξ to find a new random number ξ' is done as follows:

$$\xi' = \begin{cases} \frac{\xi}{p(L)} & \text{if } \xi < p(L), \\ \frac{\xi - p(L)}{p(R)} & \text{otherwise,} \end{cases} \quad (18)$$

where $p(C)$ is the probability of selecting node C , computed as the importance of that node divided by the total importance:

$$p(L) = \frac{\text{importance}(L)}{\text{importance}(L) + \text{importance}(R)}. \quad (19)$$

Care must be taken to ensure enough random bits are available due to the limits of floating-point precision. For scenarios with huge numbers of lights, two or more random numbers may be alternated or higher precision used.

18.4.3.3 SAMPLING THE LEAF NODE

At the end of the traversal, a leaf node containing a certain number of light sources has been selected. To decide which triangle to sample, we can either uniformly pick one of the triangles stored in the leaf node or use an importance method similar to the one used for computing the node's importance during the traversal. For importance sampling, we consider the closest distance to the triangle and the largest $\mathbf{n} \cdot \mathbf{l}$ bound of the triangle; including the triangle's flux and its orientation to the shading point could further improve the results. Currently, up to 10 triangles are stored per leaf node.

18.4.3.4 SAMPLING THE LIGHT SOURCE

After a light source has been selected through the tree traversal, a light sample needs to be generated on that light source. We use the sampling techniques presented by Shirley et al. [37] for generating the light samples uniformly over the surfaces of different types of lights.

18.5 RESULTS

We demonstrate the algorithm for multiple scenes with various numbers of lights, where we measure the rate at which the error decreases, the time taken for building the BVH, and the rendering time.

The rendering is accomplished by first rasterizing the scene in a G-buffer using DirectX 12, followed by light sampling in a full-screen ray tracing pass using a single shadow ray per pixel, and finally temporally accumulating the frames if no movements occurred. All numbers are measured on an NVIDIA GeForce RTX 2080 Ti and an Intel Xeon E5-1650 at 3.60 GHz, with the scenes being rendered at a resolution of 1920×1080 pixels. For all the results shown in this chapter, the indirect lighting is never evaluated and we instead use the algorithm to improve the computation of direct lighting.

We use the following scenes, as depicted in Figure 18-4, in our testing:

- > *Sun Temple*: This scene features 606,376 triangles, out of which 67,374 are textured emissive; however, after the texture pre-integration described in Section 18.4.1, only 1,095 emissive triangles are left. The whole scene is lit by textured fire pits; the part of the scene shown in Figure 18-4 is only lit by two fire pits located far on the right, as well as two other small ones located behind the camera. The scene is entirely diffuse.



Sun Temple



Bistro (view 1)



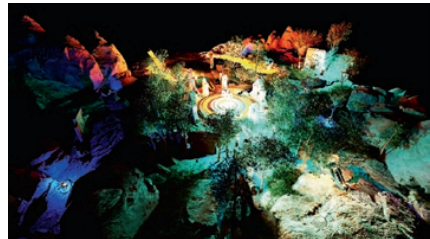
Bistro (view 2)



Bistro (view 3)



Paragon Battlegrounds: Dawn (PBG-D)



Paragon Battlegrounds: Ruins (PBG-R)

Figure 18-4. Views of all the different scenes that were used for testing.

- > *Bistro*: The Bistro scene has been modified to make the meshes of many of the different light sources actually emissive. In total, there are 20,638 textured emissive triangles, out of 2,829,226 total triangles. Overall, the light sources mainly consist of small light bulbs, with the addition of a few dozen small spotlights and a few emissive shop signs. The scene is mostly diffuse, with the exception of the bistro's windows and the Vespa.
- > *Paragon Battlegrounds*: This scene is made of three different parts, of which we only use two: Dawn (PBG-D) and Ruins (PBG-R). Both consist of a mix of large emissive area lights located in the ground, as well as small ones such as runes engraved in rocks or small lights on the turrets; most of the materials are specular, with the exception of the trees. PBG-D features 90,535 textured emissive triangles, of which 53,210 are left after the texture integration; the whole scene is made of 2,467,759 triangles (emissive ones included). In comparison, PBG-R features 389,708 textured emissive triangles, of which 199,830 are left after the texture integration; the whole scene is made of 5,672,788 triangles (emissive ones included).

Note that although all these scenes are currently static, dynamic scenes are supported in our method by rebuilding the light acceleration structure per frame. Similar to how DXR allows refitting of the acceleration structure, without changing its topology, we could choose to update only the nodes in a pre-built tree if lights have not moved significantly between frames.

We use different abbreviations for some of the methods used in this section. Methods starting with "BVH_" will traverse the BVH hierarchy in order to select a triangle. The suffix after "BVH_" refers to which information is being used during the traversal: "D" for the distance between the viewpoint and a node's center, "F" for the flux contained in a node, "B" for the $\mathbf{n} \cdot \mathbf{l}$ bound, and finally "O" for the node orientation cone. The method Uniform uses MIS [41] to combine samples obtained by sampling the BRDF with samples obtained by randomly selecting an emissive triangle among all emissive triangles present in the scene with a uniform probability.

When MIS [41] is employed, we use the power heuristic with an exponent of 2. The sample budget is shared equally between sampling the BRDF and sampling the light source.

18.5.1 PERFORMANCE

18.5.1.1 ACCELERATION STRUCTURE CONSTRUCTION

Building the BVH using the SAH, with 16 bins on only the largest axis, takes about 2.3 ms on Sun Temple, 26 ms on Bistro, and 280 ms on Paragon Battlegrounds. Note that the current implementation of the BVH builder is CPU-based, is single-threaded, and does not make use of vector operations.

Binning along all three axes at each step is roughly 2× slower due to having three times more split candidates, but the resulting tree may not perform better at runtime. The timings presented here use the default setting of 16 bins per axis. Decreasing that number makes the build faster, e.g., 4 bins is roughly 2× faster, but again quality suffers. For the remaining measurements, we have used the highest-quality settings, as we expect that the tree build will not be an issue once the code is ported to the GPU and used for game-like scenes with tens of thousands of lights.

The build time with SAOH is about 3× longer than with SAH. The difference is mainly due to the extra lighting cone computations. We iterate once over all lights to compute the cone direction and a second time to compute the angular bounds. Using an approximate method or computing bounds bottom-up could speed this up.

Using the volume instead of the surface area did not result in any performance change for building.

18.5.1.2 RENDER TIME PER FRAME

We measured the rendering times with trees built using different heuristics and with all the sampling options turned on. See Table 18-1. Similarly to the build performance, using the volume-based metrics instead of surface area did not significantly impact the rendering time (usually within 0.2 ms of the surface area-based metric). Binning along all three axes or only the largest axis also has no significant impact on the rendering time (within 0.5 ms of each other).

Table 18-1. Rendering times in milliseconds per frame with four shadow rays per pixel, measured over 1,000 frames and using the SAH and SAOH heuristics with different build parameters. The BVH_DFBO method was used with MIS, 16 bins were used for the binning, and at most one triangle was stored per leaf node.

	SAH		SAOH	
	Largest Axis	All Axes	Largest Axis	All Axes
Sun Temple	16.9 ± 0.27	17.5 ± 0.10	17.3 ± 0.47	16.2 ± 0.30
Bistro (view 1)	30.3 ± 0.18	30.3 ± 0.61	31.8 ± 0.26	30.4 ± 0.20
Bistro (view 2)	38.8 ± 0.43	36.9 ± 0.30	39.6 ± 0.31	38.3 ± 1.12
Bistro (view 3)	31.2 ± 0.60	32.3 ± 0.19	33.0 ± 0.17	32.7 ± 0.20
PBG-D	23.6 ± 0.22	23.6 ± 0.19	23.7 ± 0.59	23.3 ± 0.20
PBG-R	40.5 ± 0.14	39.8 ± 0.15	41.9 ± 0.57	41.0 ± 0.16

When testing different maximum amounts of triangles per leaf node (1, 2, 4, 8, and 10), the rendering times were found to be within 5 % of each other with 1 and 10 being the fastest. Results for two of the scenes can be found in Figure 18-5, with similar behavior observed in the other scenes. The computation of the importance of each triangle adds a noticeable overhead. Conversely, storing more triangles per leaf node will result in shallower trees and therefore quicker traversal. It should be noted that the physical size of the leaf nodes was not changed (i.e., it was always set to accept up to 10 triangle IDs), only the amount that the BVH builder was allowed to put in a leaf node. Also for these tests, leaf nodes were created as soon as possible rather than relying on a leaf node creation cost.

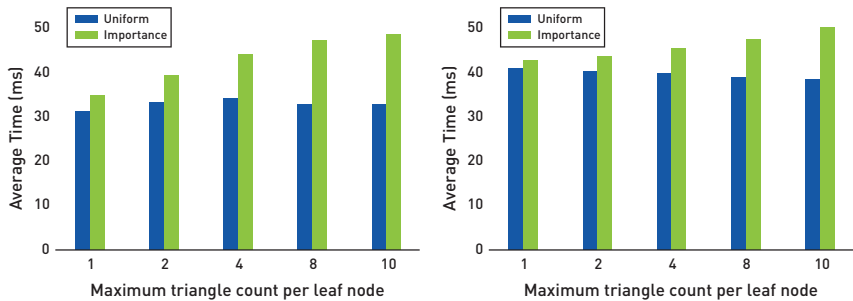


Figure 18-5. Rendering times in milliseconds per frame for various maximum numbers of triangles per leaf node for Bistro (view 1) (left) and PBG-R (right), with and without importance sampling for triangle selection within the leaves. In all cases the BVH was built with 16 bins along all three axes using SAOH, and BVH_DFBO was used for the traversal.

The use of SAOH over SAH results in similar rendering times overall, but the use of a BVH over the lights as well as which terms are considered for each node's importance do have an important impact, with BVH_DFB0 being between 2× and 3× slower than Uniform. This is shown in Figure 18-6. This boils down to the additional bandwidth required for fetching the nodes from the BVH as well as the additional instructions for computing the $\mathbf{n} \cdot \mathbf{l}$ bound and the weight based on the orientation cone. This extra cost could be reduced by compressing the BVH nodes (using 16-bit floats instead of 32-bit floats, for example); the current nodes are 64 bytes for the internal nodes and 96 bytes for the external ones.

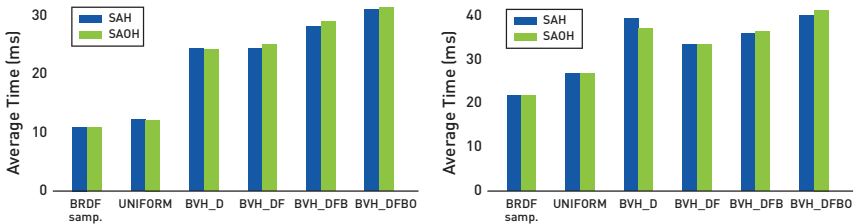


Figure 18-6. Comparisons in *Bistro* (view 1) (left) and *PBG-R* (right) of rendering times in milliseconds per frame using the different traversal methods, compared to sampling the BRDF to get the light sample direction. All methods use 4 samples per pixel, and BVH-based methods use 16 bins along all three axes.

18.5.2 IMAGE QUALITY

18.5.2.1 BUILD OPTIONS

Overall, the volume variants perform worse than their surface-area equivalents, and methods using 16 bins perform better than their counterparts only using 4 bins. As for how many axes should be considered for defining the best split, considering all three axes leads to lower mean squared error results in most cases compared to only using the largest axis, but not always. Finally, SAOH variants are usually better than or at least on par with their SAH equivalents. This can be highly dependent on how they formed their nodes at the top of the BVH: as those nodes contain most of the lights in the scene, they represent a poor spatial and directional approximation of the emissive surfaces that they contain.

This can be seen in Figure 18-7 in the area around the pharmacy shop sign (pointed at by the red arrow), for example at point A (pointed at by the white arrow). When using SAH, point A is closer to the green node than the magenta one, resulting in a higher chance of choosing the green node and therefore missing the green light emitted by the cross sign even though that green light is important, as can be seen in Figure 18-4 for *Bistro* (view 3). Conversely, with SAOH the point A has a high

chance of selecting the node containing the green light, improving the convergence in that region. However, it is possible to find regions where SAH will give better results than SAOH for similar reasons.

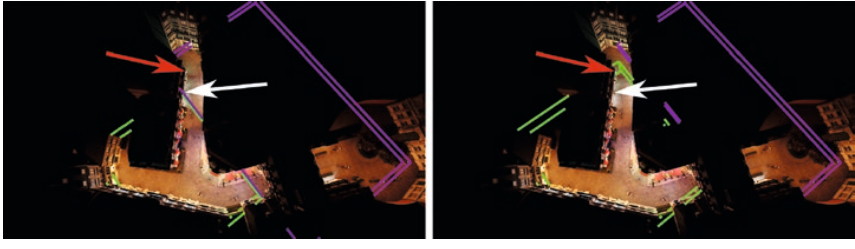


Figure 18-7. Visualization of the second level of the BVH when built using SAH (left) and SAOH (right); the ABB of the left child is colored in green whereas the one of the right child is in magenta. In both cases, 16 bins were used and all three axes were considered.

18.5.2.2 TRIANGLE AMOUNT PER LEAF NODE

As more triangles are stored in leaf nodes, the quality will degrade when using a uniform selection of the triangles because it will do a poorer job than the tree traversal. Using importance selection reduces the quality degradation compared to uniform selection, but it still performs worse than using only the tree. The results for Bistro (view 3) can be seen on the right in Figure 18-8.

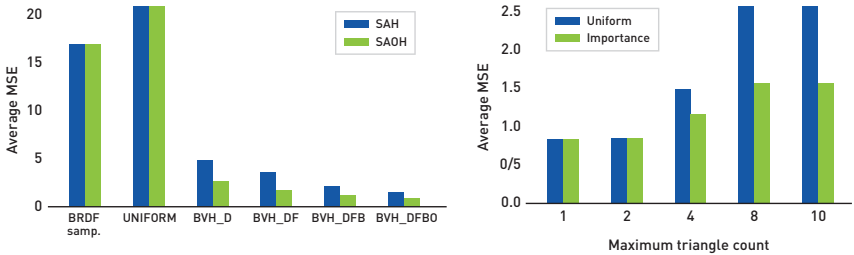


Figure 18-8. Comparisons in Bistro (view 3) of mean squared error (MSE) results for the different traversal methods, compared to sampling the BRDF to get the light sample direction (left) and various maximum amounts of triangles for BVH_DFB0 (right). All methods use 4 samples per pixel, and BVH-based methods use 16 bins along all three axes.

18.5.2.3 SAMPLING METHODS

In Figure 18-9 we can see the resulting images when using and combining different sampling strategies for the Bistro (view 2) scene.



Figure 18-9. Visual results at 4 samples per pixel (SPP) (left) and 16 SPP (right), using the different sampling strategies defined in Section 18.4.3. All BVH-based methods use a BVH built with SAOH evaluated for 16 bins along all axes. The BVH techniques use MIS: half their samples sample the BRDF and half traverse the light acceleration structure.

As expected, using light sampling greatly outperforms the BRDF sampling approach, by being able to find some valid light paths at each pixel. Using the BVH with the distance as an importance value allows picking up of the contributions from nearby light sources, as can be seen for the two white light sources placed on each side of the door of the bistro, the different lights placed on its facade, or its windows.

When also considering the flux of the light sources during the traversal, we can observe a shift from a mostly blueish color (from the hanging small blue light sources closest to the ground) to a more yellowish tone coming from the different street lights, which might be located farther away but are more powerful.

Using the $\mathbf{n} \cdot \mathbf{l}$ bounds does not make much of a difference in this scene, except for the reflections on the Vespa (mostly visible on the 16 SPP images), but the effects can be way more pronounced in other scenes. Figure 18-10 shows an example from Sun Temple. There, using the bounds on $\mathbf{n} \cdot \mathbf{l}$ results in the back of the column on the right receiving more light and being distinguishable from the shadow it casts on the nearby wall, as well as the architectural details of the ceiling of the enclave in which the statue is located becoming visible.



Figure 18-10. Visual results when not using the $\mathbf{n} \cdot \mathbf{l}$ bounds (left) compared to using it (right). Both images use 8 SPP (4 BRDF samples and 4 light samples) and a BVH binned along all three axes with 16 bins using SAH, and both take into account the distance and flux of the light.

Even without SAOH, the orientation cone still has a small impact on the final image; for example, the facades in Figure 18-9 (at the end of the street and in the right-hand corner of the image) are less noisy compared to not using the orientation cones.

The use of an acceleration structure significantly improves the quality of the rendering, as seen in Figure 18-8, with between 4× and 6× improved average MSE score over the Uniform method even when only considering the distance to a node for that node's importance function. Incorporating the flux, the $\mathbf{n} \cdot \mathbf{l}$ bound and the orientation cone give a further 2× improvement.

18.6 CONCLUSION

We have presented a hierarchical data structure and sampling methods to accelerate light importance sampling in real-time ray tracing, similar to what is used in offline rendering [11, 22]. We have explored sampling performance on the GPU, taking advantage of hardware-accelerated ray tracing. We have also presented results using different build heuristics. We hope this work will inspire future work in game engines and research to incorporate better sampling strategies.

While the focus of this chapter has been on the sampling problem, it should be noted that any sample-based method usually needs to be paired with some form of denoising filter to remove the residual noise, and we refer the reader to recent real-time methods based on advanced bilateral kernels [25, 34, 35] as a suitable place to start. Deep learning-based methods [3, 7, 42] also show great promise. For an overview of traditional techniques, refer to the survey by Zwicker et al. [48].

For the sampling, there are a number of worthwhile avenues for improvement. In the current implementation, we bound $\mathbf{n} \cdot \mathbf{l}$ to cull lights under the horizon. It would be helpful to also incorporate BRDF and visibility information to refine the sampling probabilities during tree traversal. On the practical side, we want to move the BVH building code to the GPU for performance reasons. That will also be important for supporting lights on dynamic or skinned geometry.

ACKNOWLEDGEMENTS

Thanks to Nicholas Hull and Kate Anderson for creating the test scenes. The Sun Temple [13] and Paragon Battlegrounds scenes are based on assets kindly donated by Epic Games. The Bistro scene is based on assets kindly donated by Amazon Lumberyard [1]. Thanks to Benty et al. [4] for creating the Falcor rendering research framework, and to He et al. [16] and Jonathan Small for the Slang shader compiler that Falcor uses. We would also like to thank Pierre Moreau's advisor Michael Doggett at Lund University. Lastly, thanks to Aaron Lefohn and NVIDIA Research for supporting this work.

REFERENCES

- [1] Amazon Lumberyard. Amazon Lumberyard Bistro, Open Research Content Archive (ORCA). <http://developer.nvidia.com/orca/amazon-lumberyard-bistro>, July 2017.
- [2] Andersson, J. Parallel Graphics in Frostbite—Current & Future. Beyond Programmable Shading, SIGGRAPH Courses, 2009.

- [3] Bako, S., Vogels, T., McWilliams, B., Meyer, M., Novák, J., Harvill, A., Sen, P., DeRose, T., and Rousselle, F. Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings. *ACM Transactions on Graphics* 36, 4 (2017), 97:1–97:14.
- [4] Benty, N., Yao, K.-H., Foley, T., Kaplanyan, A. S., Lavelle, C., Wyman, C., and Vijay, A. The Falcor Rendering Framework. <https://github.com/NVIDIAGameworks/Falcor>, July 2017.
- [5] Bikker, J. Real-Time Ray Tracing Through the Eyes of a Game Developer. In *IEEE Symposium on Interactive Ray Tracing* (2007), pp. 1–10.
- [6] Bikker, J. *Ray Tracing in Real-Time Games*. PhD thesis, Delft University, 2012.
- [7] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Transactions on Graphics* 36, 4 (2017), 98:1–98:12.
- [8] Christensen, P. H., and Jarosz, W. The Path to Path-Traced Movies. *Foundations and Trends in Computer Graphics and Vision* 10, 2 (2016), 103–175.
- [9] Clarberg, P., Jarosz, W., Akenine-Möller, T., and Jensen, H. W. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Transactions on Graphics* 24, 3 (2005), 1166–1175.
- [10] Clark, J. H. Hierarchical Geometric Models for Visibility Surface Algorithms. *Communications of the ACM* 19, 10 (1976), 547–554.
- [11] Conty Estevez, A., and Kulla, C. Importance Sampling of Many Lights with Adaptive Tree Splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018), 25:1–25:17.
- [12] Dachsbacher, C., Křivánek, J., Hašan, M., Arbree, A., Walter, B., and Novák, J. Scalable Realistic Rendering with Many-Light Methods. *Computer Graphics Forum* 33, 1 (2014), 88–104.
- [13] Epic Games. Unreal Engine Sun Temple, Open Research Content Archive (ORCA). <http://developer.nvidia.com/orca/epic-games-sun-temple>, October 2017.
- [14] Goldsmith, J., and Salmon, J. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications* 7, 5 (1987), 14–20.
- [15] Harada, T. A 2.5D Culling for Forward+. In *SIGGRAPH Asia 2012 Technical Briefs* (2012), pp. 18:1–18:4.
- [16] He, Y., Fatahalian, K., and Foley, T. Slang: Language Mechanisms for Extensible Real-Time Shading Systems. *ACM Transactions on Graphics* 37, 4 (2018), 141:1–141:13.
- [17] Heitz, E., Dupuy, J., Hill, S., and Neubelt, D. Real-Time Polygonal-Light Shading with Linearly Transformed Cosines. *ACM Transactions on Graphics* 35, 4 (2016), 41:1–41:8.
- [18] Kajiya, J. T. The Rendering Equation. *Computer Graphics (SIGGRAPH)* (1986), 143–150.
- [19] Karis, B. Real Shading in Unreal Engine 4. Physically Based Shading in Theory and Practice, SIGGRAPH Courses, August 2013.
- [20] Keller, A. Instant Radiosity. In *Proceedings of SIGGRAPH* (1997), pp. 49–56.
- [21] Keller, A., Fascione, L., Fajardo, M., Georgiev, I., Christensen, P., Hanika, J., Eisenacher, C., and Nichols, G. The Path Tracing Revolution in the Movie Industry. In *ACM SIGGRAPH Courses* (2015), pp. 24:1–24:7.

- [22] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The Iray Light Transport Simulation and Rendering System. arXiv, <https://arxiv.org/abs/1705.01263>, 2017.
- [23] Lagarde, S., and de Rousiers, C. Moving Frostbite to Physically Based Rendering 3.0. Physically Based Shading in Theory and Practice, SIGGRAPH Courses, 2014.
- [24] MacDonald, J. D., and Booth, K. S. Heuristics for Ray Tracing Using Space Subdivision. *The Visual Computer* 6, 3 (1990), 153–166.
- [25] Mara, M., McGuire, M., Bitterli, B., and Jarosz, W. An Efficient Denoising Algorithm for Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 3:1–3:7.
- [26] NVIDIA. NVAPI, 2018. <https://developer.nvidia.com/nvapi>.
- [27] O'Donnell, Y., and Chajdas, M. G. Tiled Light Trees. In *Symposium on Interactive 3D Graphics and Games* (2017), pp. 1:1–1:7.
- [28] Olsson, O., and Assarsson, U. Tiled Shading. *Journal of Graphics, GPU, and Game Tools* 15, 4 (2011), 235–251.
- [29] Olsson, O., Billeter, M., and Assarsson, U. Clustered Deferred and Forward Shading. In *Proceedings of High-Performance Graphics* (2012), pp. 87–96.
- [30] Persson, E., and Olsson, O. Practical Clustered Deferred and Forward Shading. *Advances in Real-Time Rendering in Games*, SIGGRAPH Courses, 2013.
- [31] Pharr, M. Guest Editor's Introduction: Special Issue on Production Rendering. *ACM Transactions on Graphics* 37, 3 (2018), 28:1–28:4.
- [32] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.
- [33] Rubin, S. M., and Whitted, T. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics [SIGGRAPH]* 14, 3 (1980), 110–116.
- [34] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 2:1–2:12.
- [35] Schied, C., Peters, C., and Dachsbacher, C. Gradient Estimation for Real-Time Adaptive Temporal Filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018), 24:1–24:16.
- [36] Schmittler, J., Pohl, D., Dahmen, T., Vogelgesang, C., and Slusallek, P. Realtime Ray Tracing for Current and Future Games. In *ACM SIGGRAPH Courses* (2005), pp. 23:1–23:5.
- [37] Shirley, P., Wang, C., and Zimmerman, K. Monte Carlo Techniques for Direct Lighting Calculations. *ACM Transactions on Graphics* 15, 1 (1996), 1–36.
- [38] Sobol, I. M. *A Primer for the Monte Carlo Method*. CRC Press, 1994.
- [39] Talbot, J. F., Cline, D., and Egbert, P. Importance Resampling for Global Illumination. In *Rendering Techniques* (2005), pp. 139–146.

- [40] Tokuyoshi, Y., and Harada, T. Stochastic Light Culling. *Journal of Computer Graphics Techniques* 5, 1 (2016), 35–60.
- [41] Veach, E., and Guibas, L. J. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of SIGGRAPH* (1995), pp. 419–428.
- [42] Vogels, T., Rousselle, F., McWilliams, B., Röthlin, G., Harvill, A., Adler, D., Meyer, M., and Novák, J. Denoising with Kernel Prediction and Asymmetric Loss Functions. *ACM Transactions on Graphics* 37, 4 (2018), 124:1–124:15.
- [43] Wald, I. On Fast Construction of SAH-Based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing* (2007), pp. 33–40.
- [44] Walter, B., Arbre, A., Bala, K., and Greenberg, D. P. Multidimensional Lightcuts. *ACM Transactions on Graphics* 25, 3 (2006), 1081–1088.
- [45] Walter, B., Fernandez, S., Arbre, A., Bala, K., Donikian, M., and Greenberg, D. P. Lightcuts: A Scalable Approach to Illumination. *ACM Transactions on Graphics* 24, 3 (2005), 1098–1107.
- [46] Ward, G. J. Adaptive Shadow Testing for Ray Tracing. In *Eurographics Workshop on Rendering* (1991), pp. 11–20.
- [47] Zimmerman, K., and Shirley, P. A Two-Pass Solution to the Rendering Equation with a Source Visibility Preprocess. In *Rendering Techniques* (1995), pp. 284–295.
- [48] Zwicker, M., Jarosz, W., Lehtinen, J., Moon, B., Ramamoorthi, R., Rousselle, F., Sen, P., Soler, C., and Yoon, S.-E. Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. *Computer Graphics Forum* 34, 2 (2015), 667–681.



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

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

Paper IV



Dynamic Many-Light Sampling for Real-Time Ray Tracing

P. Moreau^{1,2}, M. Pharr¹ and P. Clarberg¹

¹NVIDIA

²Lund University, Sweden

Abstract

Monte Carlo ray tracing offers the capability of rendering scenes with large numbers of area light sources—lights can be sampled stochastically and shadowing can be accounted for by tracing rays, rather than using shadow maps or other rasterization-based techniques that do not scale to many lights or work well with area lights. Current GPUs only afford the capability of tracing a few rays per pixel at real-time frame rates, making it necessary to focus sampling on important light sources. While state-of-the-art algorithms for offline rendering build hierarchical data structures over the light sources that enable sampling them according to their importance, they lack efficient support for dynamic scenes. We present a new algorithm for maintaining hierarchical light sampling data structures targeting real-time rendering. Our approach is based on a two-level BVH hierarchy that reduces the cost of partial hierarchy updates. Performance is further improved by updating lower-level BVHs via refitting, maintaining their original topology. We show that this approach can give error within 6% of recreating the entire hierarchy from scratch at each frame, while being two orders of magnitude faster, requiring less than 1 ms per frame for hierarchy updates for a scene with thousands of moving light sources on a modern GPU. Further, we show that with spatiotemporal filtering, our approach allows complex scenes with thousands of lights to be rendered with ray-traced shadows in 16.1 ms per frame.

CCS Concepts

• Computing methodologies → Ray tracing;

1. Introduction

Complex illumination is a critical ingredient for the visual richness of rendered images. Images that include the soft shadows and diffused lighting that is characteristic of large area light sources have a markedly more realistic appearance than images rendered with small numbers of point or directional light sources, which give stark and harsh lighting effects. However, with more than few light sources it is infeasible to shade them all individually, especially under the constraints of real-time rendering. Culling and/or stochastic selection of a subset of lights is necessary. In this work, we focus on stochastic sampling in order to be able to support many contributing light sources and still compute unbiased results.

With this approach, it is necessary to define a discrete probability density function (PDF) $p_l(\mathbf{x}, i)$ that gives the probability of sampling the i th light as seen from a point \mathbf{x} in the scene. The more closely proportional $p_l(\mathbf{x}, i)$ is to the reflected light at \mathbf{x} due to the light i 's emission, the less error will be present in the image. Unfortunately, an accurate p_l cannot be easily precomputed as there are millions of shading points \mathbf{x} , a scene may have tens of thousands of lights i , and generally, the optimal sampling distribution varies drastically between different parts of the scene.

An elegant solution exists for offline rendering: a single *bounding volume hierarchy* (BVH) is built over all lights and at each point

\mathbf{x} , the tree is stochastically traversed [KWR*17, CEK18]. At each level of the traversal, the relative contributions of the children nodes are estimated such that the full distribution p_l is never represented explicitly and only $\log(n)$ computations per shading point (where n is the number of lights) are required. This idea is illustrated in Figure 1. For offline rendering, the cost of constructing the light BVH is negligible compared to the rendering time. This is not the case for real-time rendering, where many fewer rays are generally traced per frame and no more than a few milliseconds per frame are available. The goal of this paper is to adapt light BVH methods to be suitable for real-time ray tracing of dynamic scenes. We make the following contributions:

- We organize light sources in multiple bounding volume hierarchies, arranged in a two-level hierarchy.
- We show that refitting light BVHs without modifying their topology can be implemented efficiently on the GPU, and that this approach works well for moderate amounts of light source motion.
- We demonstrate that top-level BVHs can be rebuilt asynchronously to maintain close-to-optimal overall tree topology. Thus, our approach can support a wide range of motion without an increase in sampling error due to sub-optimal BVHs.
- We present results based on a real-time path tracer implemented in Direct3D 12 using the DirectX Raytracing API.

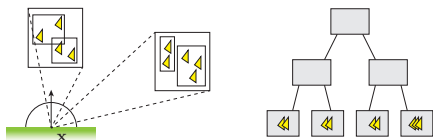


Figure 1: Light sources are stored in a bounding volume hierarchy (illustrated in 2D on the left, and as a tree on the right). To sample a light at a shading point \mathbf{x} , the tree is stochastically traversed by estimating the contributions from the two children (light clusters) at each node. Important clusters are given a higher priority and a random decision is made about which branch to follow.

We show that with our approach, the cost of maintaining the light acceleration structure is less than 1 ms per frame for dynamic scenes with tens of thousands of emitters, with mean-squared error (MSE) within 6% compared to a single optimized light BVH.

2. Previous Work

Kajiya suggested taking a single light sample at each path vertex regardless of the total number of lights [Kaj86]. Since his work, a number of researchers have investigated ways of computing accurate estimates of the contributions of the light sources to be able to choose among them more effectively.

Ward introduced the idea of generating a discrete PDF over lights at each shaded point [War91] and tracked how often each light source was visible. Shirley et al. estimated lights' contributions to compute per-light probabilities [SWZ96], using an octree to classify light sources into "bright" and "dim" sets. Zimmerman and Shirley used a uniform spatial subdivision rather than an octree and also maintained estimates of each light's visibility [ZS95]. Wald et al. generated a light sampling PDF using a sparse path tracing pass [WBS03]. Their approach works well in densely occluded environments but does not effectively distinguish between local lights that are important at some points but less so at others.

A number of light transport algorithms have been developed based on hierarchical representations of illumination encoded as point lights, including Lightcuts [WFA*05] and its predecessor [PPD98]. Closely related are global illumination algorithms based on virtual point lights (VPLs) [DKH*14]. These approaches all use point lights for illumination and not just for sampling, which introduces the possibility of error from the discretization and the weak singularity from the $1/r^2$ term close to the point lights.

Tiled shading [And09, OA11, Har12, OC17] bins lights into screen-space tiles, where the depth bounds of the tiles reduce the number of lights that need to be processed in each one. These screen space acceleration structures are not applicable to indirect intersection points with ray tracing. Further, clamped light ranges can cause undesired darkening. To address the darkening, Tokuyoshi and Harada used a bounding sphere hierarchy and stochastic light ranges to reject unimportant lights [TH16].

A number of researchers have investigated other approaches

based on building hierarchies over the light sources and traversing them to sample lights. Iray uses a hierarchical light importance sampling scheme based on a BVH [KWR*17]. Conty Estévez and Kulla [CEK18] take a similar approach for cinematic rendering with a 4-wide BVH that clusters lights in world space based on both orientation and surface normal bounding cones. Moreau and Clarberg describe a GPU implementation of their algorithm [MC19]. Vévoda et al. [VKK18] recently described an approach based on online learning of the importance of light sources based on clustering with a hierarchy and a Bayesian approach. Their method has relatively high memory use, does not support dynamic light sources, and does not readily map to GPUs.

The idea of "refitting" bounding volume hierarchies for ray intersection acceleration with animated geometry was introduced by van den Bergen [vdB97] for collision detection, and later applied to ray-object intersection [LYMT06, WBS07]. These approaches take advantage of the fact that for relatively small amounts of object motion, the original BVH's topology can be maintained with node bounds updated to account for moving objects' new positions. Doing so saves the computational expense of rebuilding the BVH. To our knowledge, these techniques have not been applied to light sampling BVHs. We note that with collision detection and ray-object intersection, sub-optimal BVHs cause an increase in computation but do not introduce error. With light sampling, low quality BVHs instead lead to inaccurate light contribution estimates, which in turn may lead to variance in rendered images—i.e., error rather than inefficiency. We apply refitting and study these effects.

3. Algorithm

The light sampling distribution should ideally be proportional to each light i 's contribution to reflected radiance L at the point \mathbf{x} being shaded: $p_l(\mathbf{x}, i) \propto L(\mathbf{x}, \omega_0)$, where ω_0 is the view direction. The more closely the distribution matches L , the lower the error will be. This principle is known as *importance sampling* in Monte Carlo integration [PJH16]. However, it is not feasible to determine a p_l that is exactly proportional to L :

$$L(\mathbf{x}, \omega_0) = \int_{A_i} \frac{f(\omega \rightarrow \omega_0) L_e(\mathbf{x}_l, -\omega) V(\mathbf{x} \leftrightarrow \mathbf{x}_l) \cos \theta \cos \theta_l}{\|\mathbf{x} - \mathbf{x}_l\|^2} d\mathbf{x}_l, \quad (1)$$

where the integration domain A_i is the surface of the i th light, f is the bidirectional scattering distribution function (BSDF), ω is the normalized vector from \mathbf{x} to a point \mathbf{x}_l on the light, L_e is the radiance emitted by the light, and V is a visibility term that is one if the two points are unoccluded and zero otherwise. The two cosine terms are with respect to the surface normals at \mathbf{x} and at \mathbf{x}_l .

We use the approach by Conty Estévez and Kulla [CEK18] as implemented by Moreau and Clarberg [MC19], which takes the total emitted flux, the $1/r^2$ falloff, and the relative orientations of the shading normal and light source into account using bounding cones. Instead of computing these terms for all the light sources, lights with nearby locations and directions are grouped together into a light BVH [KWR*17, CEK18]. The BVH allows hierarchical approximation of these quantities, reducing the per-sample complexity from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ and making it feasible to perform these computations at every shading point. This algorithm can be applied to point or area lights, as well as emissive triangles.



Figure 2: Care has to be taken when deciding which light sources to put in each bottom-level hierarchy. Left: it is difficult to accurately estimate the contribution from the large overlapping BLASes, leading to higher variance. Right: it is preferable to place lights that are spatially nearby in the same hierarchies.

3.1. Two-Level Light Acceleration Structures

Previous approaches used a single BVH for all light sources. The BVH must be rebuilt from scratch if even a single light moves or changes intensity. That approach isn't suitable for real-time rendering with dynamic lights due to the cost of rebuilding the BVH [MC19].

The problem is closely related to managing data structures for ray-intersection testing in dynamic scenes. APIs like DirectX Ray-tracing or Vulkan ray tracing and libraries like OptiX [PBD*10] and Embree [WVB*14], use *two-level BVHs* that store collections of geometry in separate bottom-level acceleration structure (BLAS) and maintain a separate top-level acceleration structure (TLAS) that stores the BLASes. A moving object only causes its BLAS and the TLAS to be rebuilt. The cost of hierarchy updates is kept low if static geometry is stored separately from dynamic objects.

With light BVHs, that partitioning is not ideal for static lights as it would lead to large BVH nodes and therefore poor estimates of node contributions due to having many emissive primitives and high uncertainty regarding their positions and orientations within the node. Other strategies such as sorting based on material can similarly be counterproductive; see Figure 2. We have found that storing each emissive mesh in its own BLAS generally gives a good balance. Figure 3 shows an example from one of our test scenes. In future work, it would be interesting to investigate automatic ways to partition emissive geometry into BLASes.

We sample our two-level light BVH by first traversing the TLAS down to a leaf node by evaluating an importance function [CEK18] for each of the current node's children and stochastically selecting one of them. Each leaf node points to a BLAS, and the same technique is used to select a light in it. The overall probability of sampling a light is the product of the probability of sampling the BLAS it is in and the probability of sampling it in its BLAS.

3.2. Updating the Two-Level Acceleration Structure

By design, if a light source has been modified then only the TLAS and BLAS to which it belongs need to be updated. Those updates can either take the form of fully rebuilding the hierarchy or refitting it. The latter keeps the topology of the hierarchy intact—i.e., the parent/children relationships between the nodes stay the same, as well as which primitives are stored in each leaf node—but the aggregate attributes like bounding boxes are updated to account for the object motion [LYMT06, WBS07]. For our light BVHs, the



Figure 3: Each yellow box is the root node of a bottom-level acceleration structure (BLAS), here shown for the Bistro scene. Notice that both static light sources, e.g., the street lights and hanging light bulbs, and dynamic emissive objects are each represented by one or more BLASes, here in total 142 for the full scene.

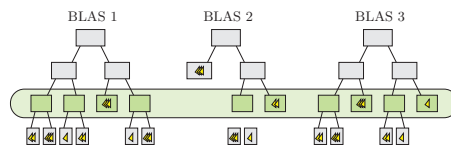


Figure 4: Our light BVH refit operation is performed bottom-up on all modified bottom-level hierarchies in parallel, with one compute shader dispatch per tree level (as shown in green). For this purpose, each tree stores a list of node indices sorted by tree level.

aggregate attributes also include the total emitted flux and normal bounding cones. Thus, even static light sources can trigger a rebuild or refit if their flux changes (for example, flashing lights).

After object animation, we dispatch compute shader passes on the GPU for all modified BLASes, with a separate dispatch for each tree level bottom-up, updating the current row's nodes based on their children's aggregate attributes; see Figure 4. We then also refit the TLAS on the GPU, to let rendering immediately proceed. However, we have found it worthwhile to rebuild the TLAS to keep it as accurate as possible. Therefore, we perform an asynchronous full rebuild of the TLAS on the CPU; this lets us exploit idle CPU cycles while the GPU is busy rendering without introducing extra latency. Figure 5 shows an execution timeline.

4. Results

We evaluated our approach using two complex scenes with dynamic light sources; see below for general statistics and Figure 3 and 6 for representative images. The scenes contain both skinned animation and multiple rigid objects that follow animation paths. Note that we perform BVH refit for all moving objects, independent of their amount of motion and type of animation. The algo-

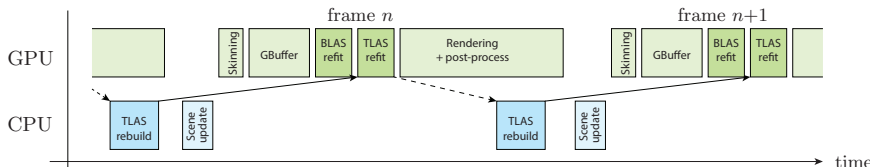


Figure 5: The bottom-level (BLAS) and top-level (TLAS) light BVHs are refitted on the GPU based on the current vertex positions and flux values for the emissive geometry. To maintain a good topology, the TLAS is rebuilt asynchronously on the CPU based on last frame’s data.

gorithms were implemented in the Falcor real-time rendering framework [BYF*18] using Microsoft Direct3D 12 and DirectX Ray-tracing (DXR). All results were measured on an NVIDIA GeForce RTX 2080Ti GPU with 12GB RAM using driver 419.67, on a computer with an Intel Xeon E5-1650 v4 CPU at 3.60GHz and 32GB RAM. The output resolution was 1920×1080 pixels.

Scene	Bistro	Emerald Square
Total triangles	3,038,170	9,687,074
Emissive triangles (static)	19,948	19,440
Emissive triangles (dynamic)	6,495	66,172

Using a two-level BVH rather than a single unified one may reduce importance sampling accuracy (recall Figure 2.) In order to evaluate the importance of this issue, we first measured rendering time and the mean squared error (MSE) for static scene using one light sample per pixel (spp) with three sampling approaches: uniform probabilities over all of the light sources, a single light sampling BVH on the GPU [MC19], and our two-level BVH. Error was measured with respect to reference images rendered with 10,000 spp. Times were averaged over a few hundred frames and only include the ray-tracing pass to compute lighting—the time to generate the G-buffer (roughly 2 ms) and for tone mapping (less than 1 ms) is not included. All techniques were combined with BRDF sampling using multiple importance sampling (MIS) [VG95], taking one BRDF sample for each light sample. For clarity of presentation, only direct illumination was evaluated; the total number of rays per pixel was therefore 2×spp (one shadow ray and one BRDF scatter ray).

The results are presented in Table 1. As has been demonstrated previously [CEK18, MC19] and is evident here, a uniform light sampling PDF is ineffective in scenes with many light sources. For these scenes, the increase in MSE from replacing a single BVH with a two-level BVH is insignificant. Note also that there is a negligible difference in runtime performance between these variants.

We next performed a set of experiments to compare four approaches for rendering scenes with moving light sources: uniform light sampling; a single-level BVH that is built from scratch when any light changes; a single-level BVH that is refit when a light changes; and a two-level BVH where bottom-level BVHs are refit and the top-level is rebuilt. Lacking an efficient GPU algorithm to build the entire BVH, the second approach is not suitable for real-time applications—a full rebuild for these scenes takes over 90 ms—but it provides a baseline that gives the best sampling probabilities and thus the lowest MSE.

Because all of these sampling methods are unbiased, *Monte*

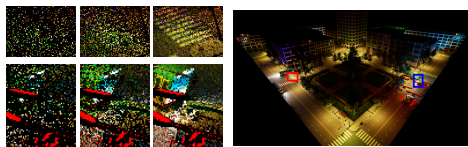


Figure 6: The Emerald Square scene with crops rendered using 1 spp, 4 spp, and 16 spp, respectively. This scenes has 143 dynamic emissive meshes for a total of 66,172 moving emissive triangles.

Carlo efficiency is a useful metric. It is defined as the inverse product of rendering time t and variance v [PJH16], which MSE is an estimate of: $\epsilon = \frac{1}{t \cdot v}$. Monte Carlo efficiency cleanly accounts for the interplay between computation time and error in the integration: because variance (and MSE) decreases at the rate $\mathcal{O}(1/N)$ for a number of samples N , it correctly indicates that, for example, a method that takes twice as much time as another to deliver half as much variance is no improvement: we could equivalently take twice as many samples with the first and expect the same variance.

The results are reported in Table 2. As before, timings are the average over a few hundred frames and MSE is computed with respect to a reference rendering with 4,000 spp. We can see that the slight reduction in update time from refitting a single BVH is not worth it: the increase in MSE is such that its Monte Carlo efficiency is on par with or lower than our approach. In a similar fashion, we can see that although a single-level BVH that is built from scratch when lights move gives the best MSE, the cost to build the BVH also is not worth it in terms of overall efficiency.

Next, we measured the effect of the accumulation of error from refitting BVHs over long animations—extensive motion of light sources may cause the original BVH topology to become inappropriate. See the accompanying video in order to see the animation. The results are presented in Figure 7 and 8. We can see that rebuilding the TLAS is worth the small computational cost.

Finally, as presented thus far, our method produces consistent, unbiased results. For practical use at low sample counts, it is essential to pair it with a reconstruction filter to remove the residual Monte Carlo noise. As a proof of concept, we have integrated *spatiotemporal variance-guided filtering* (SVGF) [SKW*17] in the renderer. Figure 9 shows an example rendered at 1 spp using uniform sampling and our method, before and after denoising. Note that the denoised result with our method is much closer to ground truth, thanks to the denoiser having better input to work with. Our

Scene	Bistro			Emerald Square		
	Uniform	One-level BVH	Two-level BVH	Uniform	One-level BVH	Two-level BVH
Time (ms)	6.2	10.4	10.7	7.7	11.3	11.6
MSE	16.8	2.12	2.14	19	0.49	0.50

Table 1: Rendering time and error using 1 spp for the first frame of the animations of the two scenes with time paused, i.e., no dynamic updates were performed. The two-level BVH only introduces a negligible increase in error of 1–2% compared to a single-level BVH.

Scene	Bistro				Emerald Square			
	Uniform	One-level (rebuild)	One-level (refit)	Two-level (rebuild/refit)	Uniform	One-level (rebuild)	One-level (refit)	Two-level (rebuild/refit)
BVH update time (ms)	0	~90	0.17	0.85 / 0.18	0	~300	0.22	0.89 / 0.35
Sampling time (ms)	0.34	2.3	2.4	2.6	0.32	2.0	2.0	2.2
Total time (ms)	6.2	101	10.8	12.0	7.7	311	11.3	12.6
MSE	16.5	1.56	1.95	1.65	20	0.58	0.67	0.61
MC efficiency ϵ	0.0097	0.0064	0.048	0.050	0.0065	0.0055	0.13	0.13
ϵ w.r.t. uniform	1 \times	0.66 \times	4.9 \times	5.2 \times	1 \times	0.85 \times	20.3 \times	20.1 \times

Table 2: Performance and error for with two different scenes rendered at 1 spp. Error was measured after 269 frames in order to capture large amounts of light movement. By comparing the Monte Carlo efficiency of the various approaches, we can see that our approach (two-level) has a much higher efficiency than one-level rebuild, without suffering from the robustness issues of refitting only, as pictured in Figure 8. Notice that we have summed the CPU and GPU execution times in this table, even though in reality they overlap.

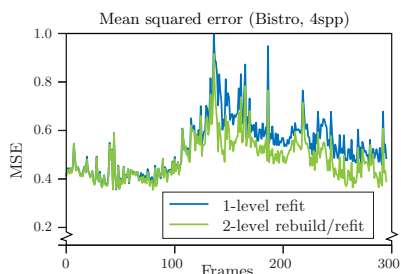


Figure 7: Error over time for an animation of the Bistro scene rendered at 4 spp. The benefit of rebuilding the top-level BVH results in an up to 1.4 \times reduction in MSE compared to using a one-level BVH that is refitted every frame. Notice that locally the differences can be much larger; see Figure 8.

SVGF implementation has not been optimized, and currently runs in roughly 4 ms per frame, for a total frame time of 16.1 ms for this scene. The supplemental video shows SVGF filtering with animation. The video was rendered with 4 spp, giving a frame time of 31.7 ms. Note that the filtered output is generally of high quality, with only minor temporal artifacts and ghosting. Gradient estimation [SPD18] would presumably reduce the ghosting.

5. Conclusion and Future Work

The arrival of ray tracing as a first-class visibility primitive in modern graphics APIs presents an opportunity for substantial improvements in the realism and richness of real-time graphics. We have introduced an approach for unbiased many-light sampling on GPUs that is suitable for ray tracing dynamic scenes. Our method is based

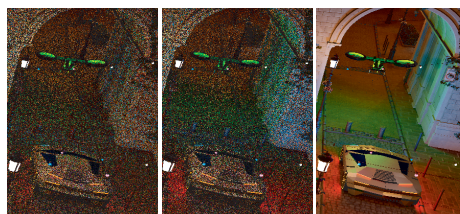


Figure 8: Crops of the Bistro scene for frame 269 of the animation at 4 spp, using a one-level BVH that is refitted each frame on the left, our two-level BVH in the center, and a reference on the right.

on a two-level hierarchy of acceleration structures over the lights and includes efficient update and sampling algorithms.

In the future, we would like to develop algorithms for building light BVHs from scratch on the GPU. Not only would it be desirable to eliminate CPU-GPU copies, but a sufficiently efficient algorithm would also allow the option of building a single BVH from scratch for moderate numbers of dynamic light sources. We would also like to investigate heuristics for determining when a refit light BVH has come to be ineffective and should be rebuilt. We hope that our work will inspire forward looking game engines and further research in rendering with complex lighting.

Acknowledgments Thanks to Nicholas Hull and Kate Anderson for creating the scenes. The Bistro scene is based on assets kindly donated by Amazon Lumberyard. The car model was made by Turbosquid user barteks2, and the helicopter asset by Sketchfab user f3nix. We would also like to thank Lund University, Aaron Lefohn and NVIDIA Research for supporting this work, and the Swedish Research Council for funding Pierre under grant 2014-5191.

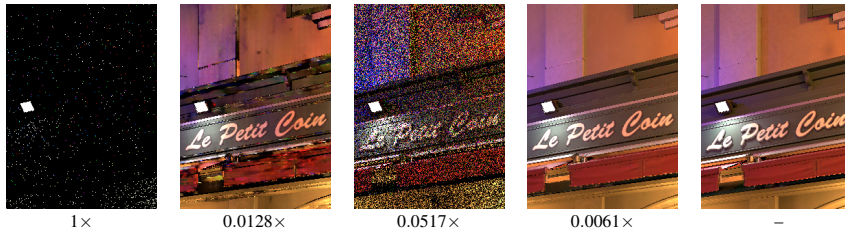


Figure 9: Crops of the Bistro scene and MSE with respect to uniform light sampling. From left to right: uniform light sampling at 1 spp; filtered with SVGF [SKW*17] (5 frames accumulated); two-level BVH sampling at 1 spp, filtered output (5 frames acc.); reference image.

References

- [And09] ANDERSSON J.: Parallel Graphics in Frostbite—Current & Future. Beyond Programmable Shading, SIGGRAPH Courses, 2009. 2
- [BYF*18] BENTY N., YAO K.-H., FOLEY T., OAKES M., LAVELLE C., WYMAN C.: The Falcor rendering framework, 05 2018. URL: <https://github.com/NVIDIAGameWorks/Falcor>. 4
- [CEK18] CONTY ESTÉVEZ A., KULLA C.: Importance Sampling of Many Lights with Adaptive Tree Splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 25:1–25:17. 1, 2, 3, 4
- [DKH*14] DACHSBACHER C., KRÍVÁNEK J., HAŠAN M., ARBREE A., WALTER B., NOVÁK J.: Scalable Realistic Rendering with Many-Light Methods. *Computer Graphics Forum 33*, 1 (2014), 88–104. 2
- [Har12] HARADA T.: A 2.5D Culling for Forward+. In *SIGGRAPH Asia 2012 Technical Briefs* (2012), pp. 18:1–18:4. 2
- [Kaj86] KAJIYA J. T.: The Rendering Equation. *Computer Graphics (SIGGRAPH)* (1986), 143–150. URL: <http://doi.acm.org/10.1145/15922.15902>, doi:10.1145/15922.15902. 2
- [KWR*17] KELLER A., WÄCHTER C., RAAB M., SEIBERT D., VAN ANTWERPEN D., KORNDÖRFER J., KETTNER L.: The Iray Light Transport Simulation and Rendering System. arXiv, <https://arxiv.org/abs/1705.01263>, 2017. 1, 2
- [LYMT06] LAUTERBACH C., YOON S. E., MANOCHA D., TUFT D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–46. 2, 3
- [MC19] MOREAU P., CLARBERG P.: Importance sampling of many lights on the GPU. In *Ray Tracing Gems*, Haines E., Akenine-Möller T., (Eds.), Apress, 2019, pp. 255–283. <http://raytracinggems.com>. 2, 3, 4
- [OA11] OLSSON O., ASSARSSON U.: Tiled Shading. *Journal of Graphics, GPU, and Game Tools 15*, 4 (2011), 235–251. 2
- [OC17] O'DONNELL Y., CHAJDAS M. G.: Tiled Light Trees. In *Symposium on Interactive 3D Graphics and Games* (2017), pp. 1:1–1:7. 2
- [PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (July 2010), 66:1–66:13. URL: <http://doi.acm.org/10.1145/1778765.1778803>. 3
- [PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016. 2, 4
- [PPD98] PAQUETTE E., POULIN P., DRETTAKIS G.: A Light Hierarchy for Fast Rendering of Scenes with Many Lights. *Computer Graphics Forum 17* (1998), 63–74. 2
- [SKW*17] SCHIED C., KAPLAYAN A., WYMAN C., PATNEY A., CHAITANYA C. R. A., BURGESS J., LIU S., DACHSBACHER C., LEFOHN A., SALVI M.: Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 2:1–2:12. 4, 6
- [SPD18] SCHIED C., PETERS C., DACHSBACHER C.: Gradient Estimation for Real-Time Adaptive Temporal Filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 24:1–24:16. 5
- [SWZ96] SHIRLEY P., WANG C., ZIMMERMAN K.: Monte Carlo Techniques for Direct Lighting Calculations. *ACM Transactions on Graphics 15*, 1 (1996), 1–36. 2
- [TH16] TOKUYOSHI Y., HARADA T.: Stochastic Light Culling. *Journal of Computer Graphics Techniques 5*, 1 (2016), 35–60. 2
- [vdB97] VAN DEN BERGEN G.: Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools 2*, 4 (1997), 1–13. URL: <https://doi.org/10.1080/10867651.1997.10487480>. 2
- [VG95] VEACH E., GUIBAS L. J.: Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of SIGGRAPH* (1995), pp. 419–428. 4
- [VKK18] VÉVODA P., KONDAPANENI I., KRÍVÁNEK J.: Bayesian online regression for adaptive direct illumination sampling. In *ACM Transactions on Graphics* (2018), pp. 125:1–125:12. 2
- [War91] WARD G. J.: Adaptive Shadow Testing for Ray Tracing. In *Eurographics Workshop on Rendering* (1991), pp. 11–20. 2
- [WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Interactive Global Illumination in Complex and Highly Occluded Environments. In *Proceedings of the 14th Eurographics Workshop on Rendering* (2003), pp. 74–81. URL: <http://dl.acm.org/citation.cfm?id=882404.882415>. 2
- [WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (Jan. 2007). URL: <http://doi.acm.org/10.1145/1189762.1206075>. 2, 3
- [WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: A Scalable Approach to Illumination. *ACM Transactions on Graphics 24*, 3 (2005), 1098–1107. 2
- [WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33*, 4 (July 2014), 143:1–143:8. URL: <http://doi.acm.org/10.1145/2601097.2601199>. 3
- [ZS95] ZIMMERMAN K., SHIRLEY P.: A Two-Pass Solution to the Rendering Equation with a Source Visibility Preprocess. In *Rendering Techniques* (1995), pp. 284–295. 2

Paper v



Real-Time Rendering of Indirectly Visible Caustics

Pierre Moreau^a, Michael Doggett^b

Faculty of Engineering, Lund University, Lund, Sweden
{pierre.moreau, michael.doggett}@cs.lth.se

Keywords: Real-time, Ray tracing, Caustics rendering

Abstract: Caustics are a challenging light transport phenomenon to render in real-time, and most previous approaches have used screen-space accumulation based on the fast rasterization hardware of GPUs. This limits the position of photon collection points to first hit screen space locations, and leads to missing caustics that should be visible in a mirror's reflection. In this paper we propose an algorithm that can render caustics visible via specular bounces in real-time. The algorithm takes advantage of hardware-accelerated ray tracing support found in modern GPUs. By constructing an acceleration structure around multiple bounce view ray hit points in world space, and tracing multiple bounce light rays through the scene, we ensure caustics can be created anywhere in the scene, not just in screen space. We analyze the performance and image quality of our approach, and show that we can produce indirectly visible caustics at real-time rates.

To be presented at GRAPP 2022
<https://grapp.scitevents.org/Home.aspx/?y=2022>

1 INTRODUCTION

Caustics are a natural phenomenon created by the concentration of light as it is reflected and transmitted through objects. While many techniques exist to generate these lighting effects in images of three dimensional scenes, generating them in real-time for interactive applications is challenging. A popular approach to achieving real-time performance is the use of screen-space algorithms, but these algorithms come with limitations, in particular not working well for scenes with reflective surfaces and semi-transparent objects. For consistent viewing of rendered images, lighting effects like caustics need to remain constant. Screen-space accumulation techniques can lead to inconsistent renderings with lighting effects switching on and off, or even missing in mirrors, as can be seen in Figure 1 for the technique by Kim (2019).

Caustics have always been an important feature

of rendering research and go back to early backward ray tracing techniques (Arvo, 1986). More recent approaches have used the rasterization pipeline and off-screen buffers for a range of techniques such as caustic mapping (Hu and Qin, 2007; Shah et al., 2007; Szirmay-Kalos et al., 2005; Wyman and Davis, 2006). While these techniques create good approximations, many issues remain, including sampling rates, large numbers of photons, and potential coherency problems during animation. With the recent introduction of hardware-accelerated ray tracing, new approaches (Kim, 2019; Ouyang and Yang, 2020a; Yang and Ouyang, 2021) take advantage of this to create high-quality caustics, but still do not handle caustics not directly visible.

Screen-space accumulation techniques that only collect lighting contributions at locations that are directly visible by the current camera are always limited in terms of correctly rendering a scene. For the case of a scene where a caustic is visible via its reflection in a mirror, the lighting contributions would be collected on the mirror itself. This means the photons responsible for the caustic would need to be reflected off the diffuse surface and into one of the collection locations on the mirror. This is unlikely as the BSDF sampling will have a low chance at returning a direction towards the mirror in most cases and that direction needs to be within the tight lobe of a rough mirror, and outright impossible in the case of a perfectly-specular mirror. Furthermore most real-

^a<https://orcid.org/0000-0003-3916-1250>

^b<https://orcid.org/0000-0002-4848-3481>

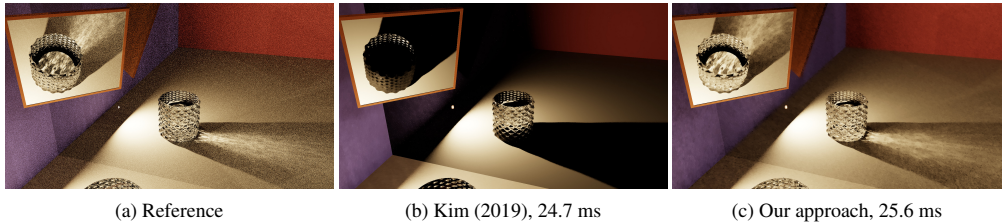


Figure 1: This figure shows a challenging setup for real-time caustics which are only visible via a chain of specular events, such as mirror reflections. As Kim (2019) accumulates photon contributions in screen space, contributions for anything seen via this perfectly specular mirror will be collected on the mirror itself requiring them to come from one very specific direction in order to have any impact; the caustics visible via the different mirrors are completely missing as a result. Our new approach can handle the caustics since its accumulation is based around a world-space acceleration structure. This scene was rendered at 1920×1080 with a path length of 6 segments using different algorithms; (a) uses 125k *samples per pixel* (spp), (b) 1 spp and 14M light paths, and (c), our temporal approach, 1 spp and 1M light paths.

time caustic-rendering techniques stop the light path at the first diffuse hit, preventing the caustic photons from ever reaching that mirror and being accounted for; this cut is performed for performance reasons as caustic paths with two or more diffuse hits will have a more subtle contribution than caustic paths with a single diffuse hit.

This paper proposes to collect photons, on diffuse surfaces, at viewing ray hit points, found using hardware-accelerated ray tracing. At each hit point for a light path, a Bounding Volume Hierarchy (BVH) node is created and constructed into a BVH tree using DirectX Raytracing. By creating a full scene BVH, the algorithm is not limited to screen-space- or view-frustum-based images. We then query the BVH using the algorithm by Evangelou et al. (2021). We also use ray differentials for appropriate sizing of the BVH nodes, and use the BVH structure as a temporal cache for filtering. Our key contribution is enabling real-time indirect caustics, by combining view and light rays with a custom BVH, all using ray tracing hardware acceleration.

2 RELATED WORK

Tracing light rays or photons from a light source to an opaque surface and accumulating light intensity on a surface has been a common approach to generating caustics (Arvo, 1986; Jensen, 2001). Generating caustics in real-time can be done by reproducing photon mapping on a GPU (McGuire and Luebke, 2009), but these approaches are expensive and not capable of generating highly accurate caustics.

Another method for generating caustics in real-time is caustic maps. Caustic maps are generated by first making a photon buffer by emitting photons from the light’s perspective into a two-dimensional buffer

similar to shadow maps. The photons from the photon buffer are then drawn into the caustic map, which is projected onto the final image. Szirmay-Kalos et al. (2005) and Wyman and Davis (2006) improved quality by increasing photon count and Wyman and Nichols (2009) created a hierarchical caustic map, that adaptively processed only the necessary parts of the photon buffer. But caustic maps still limit the locations that photons can be captured at, and the projection of high photon counts into the caustic map quickly becomes expensive when trying to improve image quality.

With the advent of hardware accelerated ray tracing, several new approaches to real-time rendering of caustics have made use of this feature. Kim (2019) uses projection volumes to direct photons towards semi-transparent objects to create photon maps limiting visible caustics to these projection volumes. Gruen (2019) focuses on creating volumetric water caustics in single-scattering participating media where a water surface casts a caustic on an underlying surface, but it does not handle refracted and then reflected light rays. Ouyang and Yang (2020a,b); Yang and Ouyang (2021) introduce caustic meshes for general caustics, but their method requires two passes for reflected and refracted caustics, and only caustics in the current viewport can be seen in this case. They further combine their technique with cascaded caustic maps for water caustics, but are focused on the caustic on the surface underlying the water. Wang and Zhang (2021) only trace rays after intersecting with a semi-transparent object, at which point it fires photons into a caustic map to layer onto the final image. By using a caustic map, their algorithm has similar limitations to screen-space techniques. While all these recent methods address caustics, some are screen space limited, and none of them address the issues of indirect caustics where photons must be accumulated outside the

view frustum to correctly handle reflected caustics, and caustics behind semi-transparent objects.

Evangelou et al. (2021) present a fast radius search by mapping the problem to GPU ray traversal and can therefore take advantage of ray tracing hardware. In our algorithm, we utilise their approach to query the BVH used when accumulating photons as well as when reusing the accumulation results in later frames. They evaluate their algorithm by using it for progressive photon mapping (Hachisuka et al., 2008) by building an acceleration structure around the photons traced from the light, whereas in our algorithm we build the acceleration structure around the camera ray hit points, where photons are collected.

3 ALGORITHM

Unlike screen-space techniques, our approach accumulates photons in a world-space caching structure based on arbitrary points and their surroundings. The size of that area is uniquely computed at each caching point, enabling support for fine-detail caustics.

The stages of our algorithm are shown in Figure 2. The first stage is a path tracer which identifies the first diffuse hit found along each camera path where light contributions will later be collected, similar to *progressive photon mapping* (Hachisuka et al., 2008). In our algorithm we refer to these *hit points* as *collection points*. An acceleration structure is then created around the collection points, and light paths are traced to accumulate light intensity at each collection point. Finally the output is resolved before being processed using a spatiotemporal filter.

We will be using the following notation: c_i refers to a collection point in frame i , \mathbf{C}_i refers to all collection points present in frame i . A collection point has the following attributes:

p	world position
τ	camera sub-path throughput
r	search radius
n	world normal
I	accumulated radiant intensity
m	material ID
L_o	exitant radiance
\tilde{L}_o	interpolated exitant radiance
\hat{L}_o	weighted contributions from previous collection points

Figure 3 shows an overview of our setup with camera and light paths. When path tracing, the collection points are selected based on the BSDF component that was evaluated for generating the reflected ray at a hit: if a diffuse component was used, then we create a collection point at the hit point. In Figure 3,

this is the case for p_1 and m_2 but not m_1 as it sampled the specular component of the mirror.

Selecting a search radius. As collecting from a single point in space is infeasible in practice, we instead gather from a surrounding area. We use the radius of the disk enclosing the pixel footprint in world space for the collection points at primary hits, as a reasonable middle ground between over-blurring and light leakage, and too restrictive radii that would ignore most light paths.

To compute the pixel footprint for collection points at secondary hits, we use half the width of ray cones as described by Akenine-Möller et al. (2021) as our search radius, and take the BSDF roughness into account as described in their Section 4. The ray cones computation is cheaper than ray differentials (Igehy, 1999), and is necessary for texture level of detail calculation.

Creating the acceleration structure. We use a similar approach to Evangelou et al. (2021) for accelerating radius searches. In their work they recommended building the acceleration structure around the photons rather than around the collection points, as atomics can then be avoided for updating the accumulated contributions at each collection point, as having overlapping collection points (for example when the same area is visible both directly and via a mirror) can noticeably decrease performance. The overhead from using atomics is about 5%; please see Section 3.2 for more details.

In contrast, we build the acceleration structure around the collection points for the following reasons:

- more predictable quality reduction when decreasing the number of collection points to improve performance, than reducing the number of photons stored;
- the light contributions accumulated at collection points can be easily reused over multiple frames, as presented in Section 3.1.
- building around photons forces the use of the largest radius for each of them, which can be costly as radii depend on location and intersected shapes and materials;

As numerical precision errors might result in the hit position being slightly above or below the surface, we gather from a cylinder rather than from a disk, similarly to most photon mapping methods. The cylinder is aligned along $c_i.n$, uses the same radius $c_i.r$, and has its height set to a tenth of $c_i.r$. We then compute the smallest AABB containing that cylinder

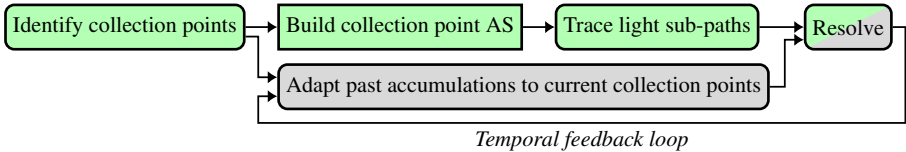


Figure 2: An overview of the different steps of our approach, with all boxes with a green colour constituting the basic algorithm, and the grey ones were either added or modified to support temporal reuse. All steps are performed in programmable shaders, except for the rectangle with non-rounded corners, which is instead taken care of by the API.

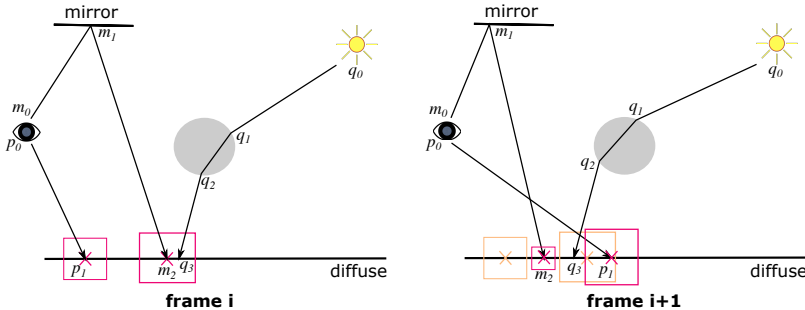


Figure 3: An overview of our indirect-caustic algorithm. Camera paths, p , m , are traced to the first diffuse surface intersection, where collection points (shown as red boxes) are created and placed in a BVH. Then light paths, q , are traced via specular intersections and collected. The path m will generate an indirect caustic in the final image. On the right half of the image is frame $i + 1$, which shows how new collection points are created when the camera rays change, but collection points from previous frames can still be reused.

and use it as the primitive around which the acceleration structure is built.

Tracing light sub-paths. The algorithm does not depend on how the set of light sources and individual light sources are being sampled, so different techniques can be used here such as the recent works by Kim (2019) or by Ouyang and Yang (2020a); Yang and Ouyang (2021).

Regardless of the chosen method(s) for sampling and tracing the light sub-paths, every time the sub-path hits a surface we will query and selectively update all collection points that contain this hit. As we leverage a hardware accelerated BVH and ray tracing API, this is implemented by tracing a very short ray (Wald et al., 2019), starting from the hit, against the acceleration structure storing our collection points. For each intersected collection point, we first check that the hit is actually located within the collection cylinder and that its material identifier matches the one stored in the collection point, before accumulating in $c_i \cdot I$ the flux carried by the photon into radiant intensity reflected towards the vertex prior to the collection point in the camera sub-path (e.g. towards m_1 if accumulating at m_2).

Resolving. Before presenting the results to the user, the accumulated radiant intensity needs to be transformed into radiance, and take account of the throughput of the camera sub-path connecting a collection point to a pixel. This is summarised in the following equation:

$$c_i \cdot L_o = c_i \cdot \tau \frac{c_i \cdot I}{\pi c_i \cdot r^2} \quad (1)$$

3.1 Temporal Reuse

A benefit of our approach is that we can reuse the exact same accumulated radiant intensity and just multiply it by the new camera sub-path throughput when a directly-visible caustic moves behind a transparent object for example, as only the light sub-path contribution gets accumulated. Whereas this accumulated data would usually be discarded by filters on occlusions or disocclusions, having the data stored in world space allows us to reuse it if appropriate.

The reuse happens in two separate steps: first we gather contributions from all past collection points located near current ones, and second, past and present contributions are combined together during the resolve step using an exponential moving average with

$\alpha = 0.8$:

$$c_i.\tilde{L}_o = \alpha c_i.\hat{L}_o + (1 - \alpha)c_i.L_o \quad (2)$$

Reusing past accumulations. This takes place between the identification of collection points in the current frame and resolve. If keeping two acceleration structures in memory is an issue, this step should be performed before updating the collection point acceleration structure for the current frame.

We query all collection points $c_{i-1} \in \mathbf{C}_{i-1}$ containing $c_i.p$, by tracing a very short ray originating from $c_i.p$ against the acceleration structure built during the previous frame. For each intersected collection point c_{i-1} satisfying the system of equations (4) (similar to a bilateral filter), its interpolated radiance $c_{i-1}.\tilde{L}_o$ weighted by a Gaussian kernel $w(c_{i-1})$ of width $\sigma = \frac{2}{3}c_i.r$ is accumulated into $c_i.\hat{L}_o$:

$$w(c_{i-1}) = \exp\left(\frac{-\|c_i.p - c_{i-1}.p\|^2}{2\sigma^2}\right) \quad (3)$$

$$\begin{cases} \|c_{i-1}.p - c_i.p\| \leq c_i.r \\ c_{i-1}.n \cdot c_i.n \geq 0.9 \\ c_{i-1}.m = c_i.m \end{cases} \quad (4)$$

3.2 Implementation Details

General constraints. If path tracing is used by an application, care needs to be taken to avoid the path tracer evaluating the same paths as the photon-based approach, or to weigh them appropriately. As we simply add the results from both approaches, we prevent the path tracer from evaluating paths containing at least one diffuse and one specular bounce as those will be handled by the photon-based approach. Using Heckbert’s light transport notation (Heckbert, 1990), this corresponds to paths of the form $ES^*D(SID)^*S(SID)^*L$.

In all presented results, the search radius was capped to 5 mm as a middle ground between performance of the light tracing stage and quality.

Performance improvements. For the first stage of the pipeline, we found that storing all the data associated to our caching approach as soon as a suitable collection point was identified, rather than at the end to minimise control flow divergence, improved the performance of that stage from 20 ms down to about 4 ms. This is due to not having to keep all that data live across the tracing of path segments.

As current APIs do not expose atomic additions for floats, we first implemented the contribution accumulation using atomic compare-and-swap within a

loop, resulting in a 15% overhead compared to no atomics. By instead using fixed-point values and integer atomic add, the overhead was reduced to 5%. As the accumulated values are well below 1, we only used 4 digits for the decimal part and the remaining 28 for the fractional part.

4 RESULTS

We implemented our approach on top of Kim’s (2019), inside the Falcor (Benty et al., 2020) 4.3 framework using the DirectX 12 and DirectX Raytracing API. All results were obtained on an NVIDIA Geforce RTX 3080 with NVIDIA’s 471.96 drivers, and rendered at 1920×1080 with a maximum path length of 6 segments and using 1024^2 light paths, unless mentioned otherwise. *Spatiotemporal Variance-Guided Filtering* (SVGF) (Schied et al., 2017) was used to filter the computed images before presenting them to the user.

In the following section we will be using the following abbreviations for methods: *OurBasic* which refers to our basic algorithm described in Section 3 (i.e. without the temporal component) and integrated with a path tracer as described in Section 3.2, and *OurTemporal* which is our full algorithm (i.e. *OurBasic* plus the temporal component described in Section 3.1).

All scenes, besides the ones shown in Figure 7, use a similar template of a closed box whose left and back wall and ceiling are specular, with the other parts being diffuse. This box contains a simple emissive mesh as the only light source, a transparent object, and a mirror. Of these scenes, four have animations of different types: *animated camera* (AC), *animated light* (AL), *animated geometry* (AG; it is the transparent sphere), and *all animated* (AA; it combines all previous animations). They are all 10 seconds long and animated at 30 fps. Unless mentioned otherwise, the specular walls and ceiling in the animated scenes have a roughness R of 0.08.

The *Bistro Exterior* and *Bistro Interior* scenes used in Figure 7 were modified to limit the number of emissive triangles by setting the emissivity to 0 for most light sources. Additionally in *Bistro Interior*, light paths were only traced from the lamps placed on tables. To more easily showcase caustics, transparent objects were added on the tables in *Bistro Exterior* while some of the glasses were removed from *Bistro Interior* to make more room on the tables. In both scenes, the windows of the bistro were changed from a very rough glossy surface into a specular mirror.

The source code and the videos can be found on

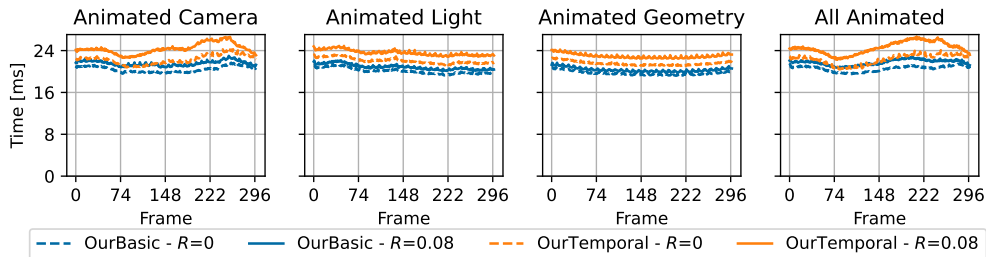


Figure 4: Measuring the total frame time taken by our basic and temporal algorithms for different types of animations. For comparison a path tracer at equal quality would require about 175 spp and take between 2.5 and 3 seconds; this was measured for equal FLIP-mean on AC@150 and AC@298.

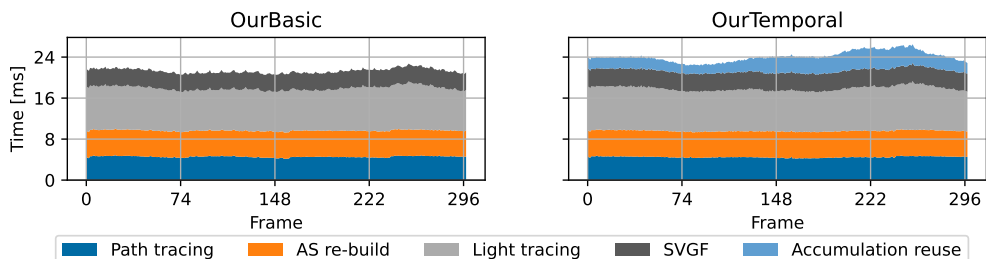


Figure 5: Highlighting the performance of the main stages of the algorithm during the animation in the scene AC. The identification of collection points is performed during the path tracing step, and the G-buffer generation time (a constant 0.6 ms) was also included under the *Path tracing* step, while the *resolve* stage (a constant 0.4 ms) was included under the *SVGf* step. 90% of the re-build time is spent on the bottom level acceleration structure. Not part of these plots, there is an additional 0.27 ms per frame from tonemapping and other miscellaneous operations to reach the timings presented in Figure 4. Note that *accumulation reuse* could be run concurrently to re-building the acceleration structure and tracing the light paths.

the project page¹.

4.1 Performance

To evaluate the performance of our approach, we ran it on different types of animations to see their impact.

Looking first at the total frame times presented in Figure 4, we can see that the cost of our approaches remains relatively constant throughout the light and geometry animations. Larger changes can be observed during the camera animation and seem to mostly correspond to changes in the number of bounces before hitting a diffuse surface from the camera. Most of that time variation comes from the pass reusing the accumulated contributions from previous frames with the current collection points.

Our approach can extend current path tracing-based frameworks and as such reuse some of the computations already performed there. For example the

identification of collection points can be added to an existing path tracer to store additional data without having to re-trace the same rays in a separate pass. We noted an increase in the cost of that pass from about 3 ms to about 4 ms when doing so, for a path tracer ignoring caustic paths. This combined cost is presented in Figure 5, along with the cost of the other steps.

The largest part of the cost of *accumulation reuse* and *light tracing* comes from tracing against the collection point acceleration structure and the invocation of the intersection and any-hit shader, due to overlapping collection points and memory accesses to get the needed information during the evaluation of the shaders. For comparison, the same light tracing but with the accumulation performed in screen space as presented by (Kim, 2019) takes 1–1.5 ms, as opposed to the 8.5–9 ms of our approach.

The second most expensive step is rebuilding the acceleration structure. Rebuilding remains an expensive operation for any real-time ray tracing-based workflow, and as such refitting is favoured for

¹https://fileadmin.cs.lth.se/graphics/research/papers/2022/indirectly_visible_caustics/

Table 1: Average image quality measurements over 10 iterations using SVGF. OurBasic performing slightly better than OurTemporal after filtering in AL@150 can be explained by both the temporal lag and the contribution of longer paths having a larger impact on the final image than in the other scenes. *Equal Time PT (3 spp)* is about the same time as, or slightly more expensive than, OurTemporal. The standard deviation was at or below 3.7% of the mean in all configurations, except for the filtered output in AG@10 for *OurTemporal* which reached 12.7% of the mean.

Scene, FrameID	Measure	Equal Time PT (3 spp)		OurBasic		OurTemporal	
		Unfiltered	Filtered	Unfiltered	Filtered	Unfiltered	Filtered
Animated Camera, Frame 150	MAE	0.074	0.041	0.084	0.022	0.065	0.021
	FLIP	0.335	0.292	0.280	0.190	0.241	0.168
Animated Camera, Frame 298	MAE	0.162	0.063	0.150	0.036	0.111	0.028
	FLIP	0.468	0.363	0.410	0.229	0.309	0.179
Animated Light, Frame 150	MAE	0.117	0.084	0.116	0.037	0.088	0.045
	FLIP	0.562	0.520	0.384	0.273	0.335	0.276
Animated Light, Frame 298	MAE	0.077	0.034	0.097	0.029	0.073	0.025
	FLIP	0.323	0.292	0.338	0.261	0.291	0.218
Animated Geometry, Frame 10	MAE	0.127	0.090	0.135	0.036	0.093	0.034
	FLIP	0.465	0.406	0.364	0.204	0.266	0.173
All Animated, Frame 298	MAE	0.077	0.033	0.096	0.020	0.077	0.019
	FLIP	0.228	0.163	0.233	0.110	0.189	0.093

most frames while rebuilding can be performed asynchronously every now and then to keep the tracing performance optimal. However the location or distribution of collection points seemed to vary too much between frames, resulting in refitting degrading the tracing performance by an order of magnitude as soon as enabled. As the number of collection points depends on the resolution of the rendering and not on the scene, the cost of this step remained the same in the *Bistro Interior* and *Bistro Exterior* scenes from Figure 7.

4.2 Quality

As our approach targets real-time applications with different types of motions, we evaluated the quality at different points during animations rather than on still images. We looked at the quality both prior and after filtering, as well as both numerical (using *mean-absolute error* (MAE)) and perceptual (using FLIP (Andersson et al., 2020)) methods; all measurements were performed on non-tonemapped outputs.

From Table 1 we can see that both *OurBasic* and *OurTemporal* improve for both metrics in all but one scene compared to the baseline. *OurTemporal* further improves compared to our basic algorithm in most scenes, for example in AC@198 the FLIP results are improved by nearly 15%) but also presents some regressions as can be seen in AG@10 for example (though they are within run to run variance).

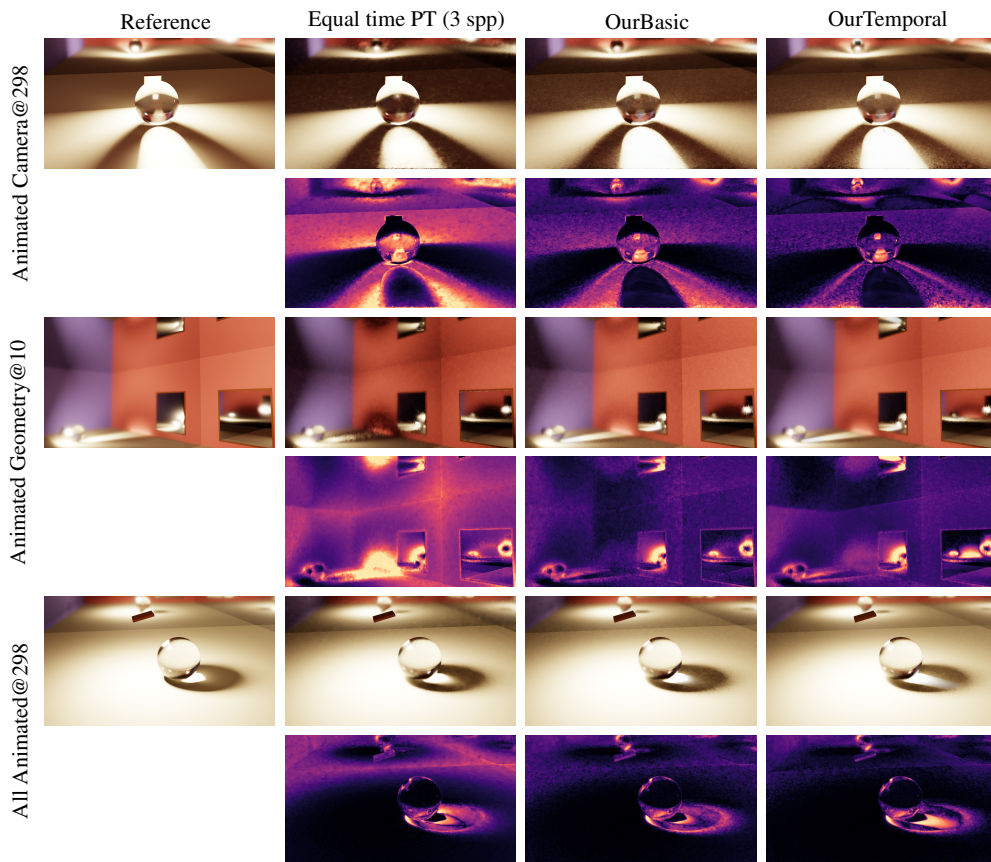
There are multiple reasons for those regressions that can be illustrated with results from Table 2. First, when we reuse accumulated radiance from previous frames we do not know the length of the light sub-

paths having contributed, so we can end up creating longer paths than what was specified, as can be seen for AG@10 in Table 2 at the top of the image where the mirror contains a reflection of the caustic on the ground, but that caustic is missing from the reference. This can be seen as an advantage, as longer paths can be created at no additional cost. A second one, which can be seen in the caustic in AA@298 for *OurTemporal*, is ghosting artefacts due to the temporal reuse simply relying on an exponential moving average; *Equal time PT* and *OurBasic* also suffer from some ghosting introduced by SVGF, but it is not as noticeable. Finally, there is a conflict between the two temporal reuse methods, our reuse at the collection points and the SVGF’s one: as new regions become visible, our temporal reuse will end up creating two different noise levels for a given surface (the more converged one, which was visible for several frames, and the newly uncovered one with very few samples) which will be interpreted by SVGF as two different regions making them more visually distinct.

Another important note is that SVGF relies on motion vectors which are rarely readily available for light patterns such as caustics or shadows, or reflections or refractions, all of which are found all over these scenes. A recent approach by Zeng et al. (2021) shows promise regarding glossy reflections.

Temporal stability is sometimes improved in real-time applications by performing the filtering *after* tonemapping rather than *before*, though at the cost of image quality. This can however result in different samples being merged together due to no longer appearing distinct enough to the filter, such as the few caustic samples in the second picture of Figure 6

Table 2: Highlighting different scenarios: AC@298 where *OurTemporal* improved significantly compared to *OurBasic*, AG@10 presents a regression for *OurTemporal*, and AA@298 with an easier to sample caustic for the path tracer. For each scene, the first row consists of a single frame filtered with SVGF, whereas the second row has error maps generated by FLIP (Andersson et al., 2020).



which were mostly blurred out. Thanks to our approach providing more samples, it can be used along that filtering trick.

Apart from the previously-mentioned ghosting in *OurTemporal*, the temporal quality depends strongly on the light sampling algorithm used and better results could be obtained with Ouyang and Yang (2020a); Yang and Ouyang (2021).

For temporal results, we refer the reader to our supplemental video which contains all 4 animations (using filtering *post*-tonemapping and $R = 0$) presented in this paper, as well as additional combinations for one of the scenes. Additional videos covering all combinations can be found on the project website (see Section 4).

As mentioned in the introduction, screen-space accumulation techniques could technically still be used to collect lighting contributions on glossy surfaces. We tried to use the technique by (Kim, 2019) on such surfaces, but failed to get it to match a reference unless increasing the roughness past 0.25, at which point caustics and objects could no longer be distinguished or seen in the reflections and the mirrors.

5 CONCLUSION

Conclusion. In this paper we presented a new algorithm for real-time rendering of detailed caustics

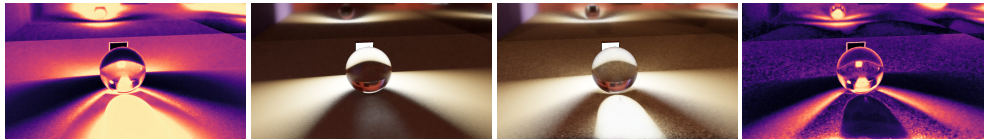


Figure 6: Unlike path tracing, our approach samples the caustics sufficiently that they do not disappear when filtering after tonemapping, to improve temporal stability. From left to right: Φ LIP error map for *Equal Time PT* (2 spp), *Equal Time PT* (2 spp), *OurTemporal*, Φ LIP error map for *OurTemporal*. These can be compared to the reference image and results obtained when filtering prior to tonemapping that are found in Table 2.



Figure 7: Our approach can be applied to more complex scenes (left image: 39 ms, path length of 7 segments; centre image: 46 ms, path length of 6 segments) and scales to more intricate caustics (right image, 1 s, path length of 6 segments; 175k spp for an equal quality path tracer). For the first two images, the main costs are path tracing (17–19 ms), light tracing (8–11 ms), and AS re-build (6 ms). All three images were rendered using our temporal version, and while the first two were filtered with SVGF, the last was accumulated over multiple frames.

appearing in long specular view paths. Our method makes use of recent hardware-accelerated ray tracing for both view and light rays, and for BVH construction. We create a temporal cache of previous frame light intensity to improve temporal filtering. Temporal filtering costs more in frame time, if not performed asynchronously, but improves image quality in most cases. Our results show that performance of 20–28 ms for the *box* scenes, is possible with temporal filtering for scenes with reflective surfaces showing caustics that are not rendered by existing screen-space accumulation techniques. Additionally, our approach can be applied to complex scenes.

Future work. The variation in collection point locations from frame to frame depending on the material sampling goes against the assumptions made by current BVH refitting approaches, resulting in low tracing performance. Temporal filtering of caustics remains an open issue with one of its challenges being the obtention of motion vectors for the caustics, which would help in reducing ghosting artefacts. The data cached by our approach could be extended to include, for example, a reservoir to use ReSTIR (Bitterli et al., 2020) even on surfaces visible via mirror(s).

ACKNOWLEDGMENTS

We thank Jacob Munkberg for valuable insights and discussions. Pierre Moreau was supported by Veten-

skapsrådet, and Michael Doggett is supported by ELLIIT and WASP. The transparent glass used in the teaser and Figure 7 was made by Simon Wend-sche². The *Bistro Interior* and *Bistro Exterior* scenes used in Figure 7 are courtesy of Amazon Lumber-yard (2017).

REFERENCES

- Akenine-Möller, T., Crassin, C., Boksansky, J., Belcour, L., Pantelev, A., and Wright, O. (2021). Improved shader and texture level of detail using ray cones. *Journal of Computer Graphics Techniques (JCGT)*, 10(1):1–24.
- Amazon Lumberyard (2017). Amazon Lumberyard bistro, open research content archive (ORCA). <http://developer.nvidia.com/orca/amazon-lumberyard-bistro>.
- Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. (2020). Φ LIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23.
- Arvo, J. (1986). Backward ray tracing. In *Developments in Ray Tracing (SIGGRAPH 86 Course Notes)*, volume 12.
- Benty, N., Yao, K.-H., Clarberg, P., Chen, L., Kallweit, S., Foley, T., Oakes, M., Lavelle, C., and Wyman, C. (2020). The Falcor rendering framework. <https://github.com/NVIDIAGameWorks/Falcor>.
- Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A.,

²<https://byob.carbonmade.com/>

- and Jarosz, W. (2020). Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Trans. Graph.*, 39(4).
- Evangelou, I., Papaioannou, G., Vardis, K., and Vasilakis, A. A. (2021). Fast radius search exploiting ray tracing frameworks. *Journal of Computer Graphics Techniques (JCGT)*, 10(1):25–48.
- Gruen, H. (2019). Ray-guided volumetric water caustics in single scattering media with dxr. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, chapter 14, pages 183–201. Apress, Berkeley, CA.
- Hachisuka, T., Ogaki, S., and Jensen, H. W. (2008). Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, New York, NY, USA. Association for Computing Machinery.
- Heckbert, P. S. (1990). Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154.
- Hu, W. and Qin, K. (2007). Interactive approximate rendering of reflections, refractions, and caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):46–57.
- Igehy, H. (1999). Tracing ray differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 179–186, USA. ACM Press/Addison-Wesley Publishing Co.
- Jensen, H. W. (2001). *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters.
- Kim, H. (2019). Caustics using screen-space photon mapping. In Haines, E. and Akenine-Möller, T., editors, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, chapter 30, pages 543–555. Apress, Berkeley, CA.
- McGuire, M. and Luebke, D. (2009). Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 77–89, New York, NY, USA. Association for Computing Machinery.
- Ouyang, Y. and Yang, X. (2020a). Generating ray-traced caustic effects in unreal engine 4, part 1. <https://developer.nvidia.com/blog/generating-ray-traced-caustic-effects-in-unreal-engine-4-part-1/>.
- Ouyang, Y. and Yang, X. (2020b). Generating ray-traced caustic effects in unreal engine 4, part 2. <https://developer.nvidia.com/blog/generating-ray-traced-caustic-effects-in-unreal-engine-4-part-2/>.
- Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. (2017). Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High-Performance Graphics*, HPG '17, New York, NY, USA. Association for Computing Machinery.
- Shah, M. A., Kontinen, 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.
- Szirmay-Kalos, L., Aszódi, B., Lazányi, I., and Premecz, M. (2005). Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3):695–704.
- Wald, I., Usher, W., Morrical, N., Lediaev, L., and Pascucci, V. (2019). RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *High-Performance Graphics - Short Papers*.
- Wang, X. and Zhang, R. (2021). Rendering transparent objects with caustics using real-time ray tracing. *Computers & Graphics*, 96:36–47.
- Wyman, C. and Davis, S. (2006). Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 153–160, New York, NY, USA. Association for Computing Machinery.
- 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 *Ray Tracing Gems II*, chapter 30, pages 469–497. Apress, Berkeley, CA.
- Zeng, Z., Liu, S., Yang, J., Wang, L., and Yan, L.-Q. (2021). Temporally reliable motion vectors for real-time ray tracing. *Computer Graphics Forum*, 40(2).