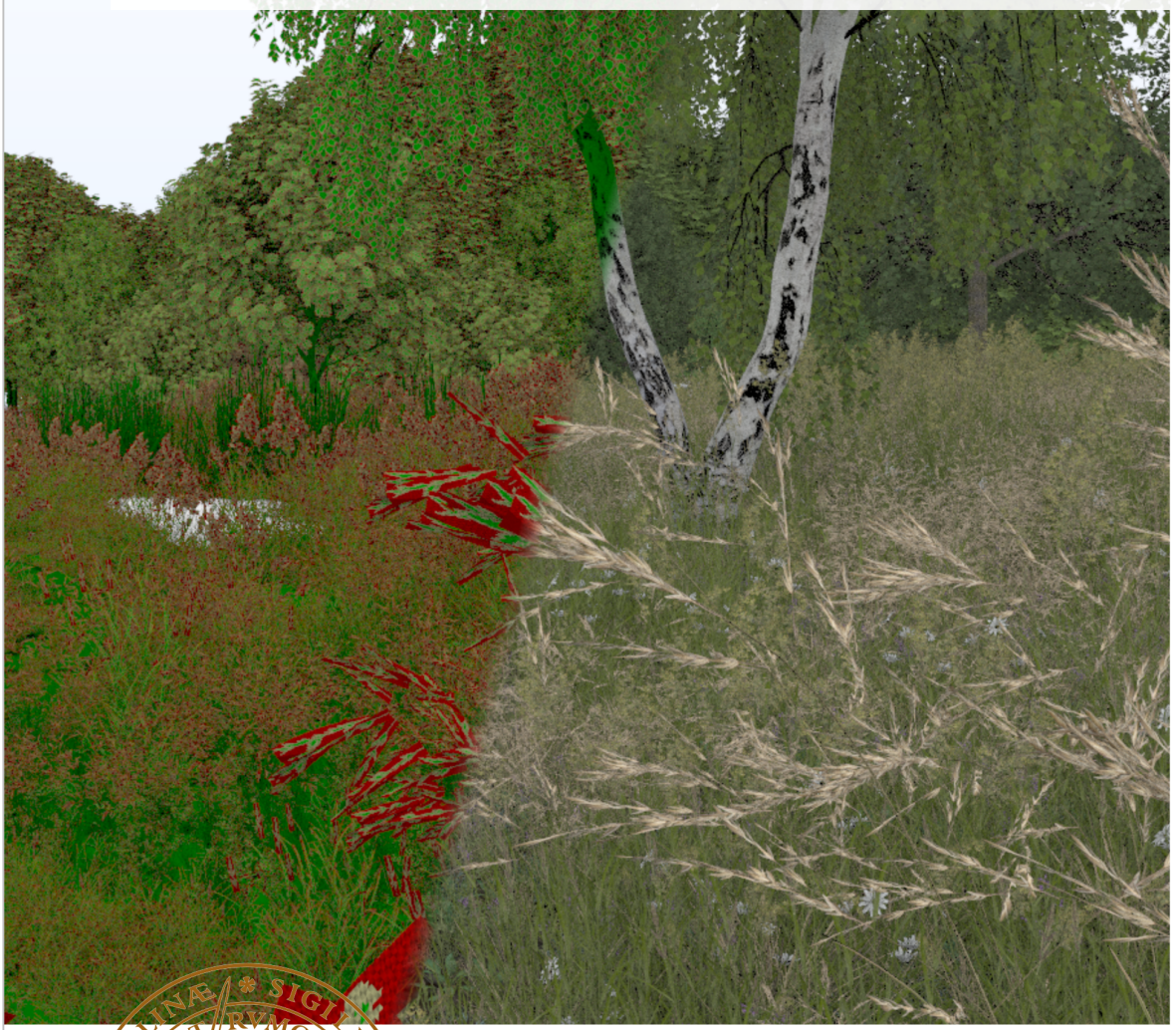
Rendering Small Things
Hardware Micromaps and Particles

Gustaf Waldemarson





Doctoral Dissertation, 2026

Department of Computer Science
Lund University

Gustaf Waldemarson

Rendering Small Things — Hardware Micromaps and Particles

2026

Rendering Small Things
Hardware Micromaps and Particles

Rendering Small Things
Hardware Micromaps and Particles

by Gustaf Waldemarson



LUND UNIVERSITY

LTH
FACULTY OF ENGINEERING

Thesis for the degree of Doctor of Philosophy

Faculty Opponent:	Associate Professor Jeppe Frisvad
Thesis Advisor:	Associate Professor Michael Doggett
Assistant Advisors:	Doctor Simone Pellegrini, and Associate Professor Flavius Gruian

To be presented, with the permission of the Faculty of Engineering of Lund University, for public criticism in lecture hall E:1406 at the Department of Computer Science on Friday, the 24th of April 2026 at 13:15.

Organization	Document name
LUND UNIVERSITY Department of Computer Science Box 118 SE–221 00 Lund Sweden	**DOCTORAL DISSERTATION** Date of disputation 2026-04-24 Sponsoring organization WASP and Arm Sweden AB
Author(s) Gustaf Waldemarson	

Title and subtitle

Rendering Small Things: Hardware Micromaps and Particles

Abstract

In computer graphics, there are numerous aspects that must be considered when rendering images of virtual scenes: What physical light-generating phenomena do we care about? How should object and material surfaces be described? And how should these be stored to ensure as fast and efficient image rendering as possible?

As a part of this thesis, a method for rendering images of scenes lit by virtual atomic particles traveling at superluminal speeds is presented that handles both the particle interactions and light generation in a unified ray-tracing framework.

However, this form of ray-tracing can be very time-consuming. Thus, this work includes an investigation into accelerating one aspect of this process: Parallelizing the construction of the spatial split bounding volume hierarchy in a simple and straightforward way with the OpenMP framework.

A similar ray-tracing process is then optimized for real-time rendering of partially-transparent triangle meshes by efficiently leveraging and compressing a structure known as micromaps that enables ray-tracing to work more efficiently for various alpha-masked geometries such as grass and foliage.

This structure is subsequently generalized and extended to arbitrary surface attributes, with a thorough analysis of its performance, quality trade-offs, and potential future use-cases in the hardware accelerated real-time ray-tracing pipeline.

Key words

rendering, ray-tracing, acceleration structures, micromaps

Classification system and/or index terms (if any)

Supplementary bibliographical information	Language
	English
ISSN and key title 1404-1219	ISBN 978-91-8104-872-8 (print) 978-91-8104-873-5 (pdf)
Recipient's notes	Number of pages 165 / Price
	Security classification

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 _____

DOKUMENTDATABLAD enl SIS 61 41 21

Rendering Small Things

Hardware Micromaps and Particles

by Gustaf Waldemarson



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 authors.

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: Images of the "Landscape" scene by Laubwerk [94] that contains many complex instances of alpha masked geometry and where the *opacity micromaps* used in Paper III are particularly important.

Starting on the back and to the left, the number of alpha-tests in the scene are visualized for each pixel, first without, and then with opacity micromaps.

On the front page, the micromaps themselves are visualized to the left, and the scene itself to the right, all rendered using a ray-tracing framework developed as a part of this work.

Additionally, the scene itself was imported into Blender [58] with the tooling described in section 6 and exported to `glTF` [68] with micromaps using the tools from section 7.

Funding information: This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and partially by Arm Sweden AB.

© Gustaf Waldemarson 2026

Faculty of Engineering, Department of Computer Science

ISBN: 978-91-8104-872-8 (print)
ISBN: 978-91-8104-873-5 (pdf)
ISSN: 1404-1219

Typeset using LuaLaTeX
Printed in Sweden by Tryckeriet i E-Huset, Lund University, Lund 2026

Dedicated to my siblings
Pelle — Sophia — Victoria

Contents

Rendering Small Things: Hardware Micromaps and Particles

Preface i

Acknowledgements . . . iii
Popular Summary in English . . . v
Populärvetenskaplig Sammanfattning på Svenska . . . vii
List of Publications and Author Contributions . . . ix

Background 1

1 Overview . . . 1
1.1 Computer Graphics . . . 2
2 Images, Colors and Blending . . . 3
3 Virtual Scenes and Objects . . . 5
3.1 Geometry . . . 5
3.2 Cameras . . . 6
3.3 Light Sources . . . 7
3.4 Materials . . . 8
3.5 Textures . . . 10
Alpha Masking . . . 11
Displacement and Normal Maps . . . 11
4 Ray-Tracing . . . 13
4.1 Light Transport . . . 13
The Rendering Equation . . . 14
Ambient Occlusion . . . 20
4.2 Acceleration Structures . . . 22
Grids . . . 22
Kd-Tree . . . 23
Bounding Volume Hierarchies . . . 23
Quality . . . 24
Split Methods . . . 25
Spatial Splits . . . 27
4.3 Parallelization . . . 31

4.4 OpenMP . . . 32
4.5 Compute Shaders and OpenCL . . . 32
4.6 Hardware Accelerated Ray-Tracing . . . 34
4.7 Micromaps . . . 39
Opacity Micromaps . . . 39
Displacement Micromaps . . . 41
5 Evaluation and Methodology . . . 45
5.1 Image Comparisons . . . 45
5.2 Performance Measurements . . . 47

Contributions **49**
6 PBRT Scene Importing for Blender . . . 53
7 Extending glTF Scenes from Blender . . . 58
8 Scientific Contributions . . . 62
9 Conclusions and Looking Forward . . . 64

Bibliography **67**
Academic References . . . 67
Software and Application Programming Interfaces . . . 72
Scenes and Assets . . . 75

Scientific Publications 77
Included Papers . . . 77
Paper I: Photon Mapping Superluminal Particles . . . 79
Paper II: Parallel Axis Split Tasks for BVH Construction with OpenMP . . . 87
Paper III: Succinct Opacity Micromaps . . . 101
Paper IV: Color and Attribute Micromaps . . . 121

Appendix **139**
Conference Posters . . . 139
A: Photon Mapping Superluminal Particles . . . 141
B: Hardware Based Ray-Traced Transparency with Opacity Micromaps . . . 142
C: Succinct Opacity Micromaps . . . 143
D: Parallel Axis Split Tasks for BVH Construction with OpenMP . . . 144
E: Real-Time Path-Tracing of Ray-Tracing Benchmarks of Yore . . . 145
F: Color and Attribute Micromaps . . . 146

Acknowledgements

First and foremost, I want to thank my main supervisor, *Michael Doggett*, who has guided me through this work while still giving me a large amount of leeway in the research topics, which I just *may* have used a bit too liberally. Thanks for staying with me for all this time, even when I strayed a bit too long in the various dead-ends.

Furthermore, I am immensely grateful to *Rikard Olajos*, *Pierre Moreau*, and *Arseni Ivanov* in the Lund University Graphics Group, and *Gareth Callanan* and *Michail Boulasikis* from the Parallel Systems Group for all our discussions on everything from politics to games and actual graphics!

I also want to thank all of the other PhD students at the Computer Science department and within *WASP* that I have interacted with: Listed here in no-specific order, but with a special nod to *Anton Risberg Alaküla* who seeded many of our wacky lunch time conversations: *Matthias Mayr*, *Noric Couderc*, *Alexander Dürr*, *Faseeh Ahmad*, *Momina Rizwan*, *Alexandru Dura*.

Furthermore, I would also like to thank my colleagues at *Arm*, and especially the compiler team that have endured this decidedly *non-standard* adventure with grace, and particularly to *Simone Pellegrini*, who helped out as my industrial supervisor. *Mathieu Robart* deserves a special thanks for helping me get started on the topic of *micromaps*, which has been instrumental for much of this PhD. Another round of thanks goes to my managers: *Philippe Caucaud*, *Robert Barucak* and *Max Andersson* for giving me the time I needed to finish up this work.

Moreover, I want to send thanks to the Lund Nvidia research team for our sporadic lunches and fun conversations, and particularly to *Pontus Ebelin* for our various WASP related interactions.

And finally, I want to give a very special thank you to my girlfriend *Sofia* for giving me the final push I needed to actually finish the PhD, and this thesis itself: Thank you!

On a professional level, I would like to thank *Matt Pharr*, *Wenzel Jakob*, and to *Greg Humphreys* for their work on the *Physically Based Ray Tracer* (`PBRT`) and the *Physically Based Rendering* (`PBR`) book, which I have frequently used as a source material to implement many important graphics algorithms.

I would also like to extend thanks to all developers of the various *Blender* frameworks and tools, many of which have been invaluable throughout this work, and an extra thank you to *Julien Duroure* for his work on the `glTF` importers and exporters that I referenced extensively in my own work.

Additionally, numerous scenes and 3D assets have been used to reach this achievement, and I would like to thank them all for their work and for making it available for use in research, but a special rounds of thanks to:

- The Khronos Group for their `glTF` sample assets [95],
- Frank Meinl, Marko Dabrovic, CryTek, and Intel for the various "Sponza" scenes [91, 92, 98],
- Amazon, Nvidia, and Epic Games for the Open Research Content Archive (`ORCA`) scenes and assets [87],
- Guillermo M. Leal Llaguno for the "San-Miguel" scene [96], and
- Laubwerk for their fantastic "Landscape" scene [94].

All of which have been used liberally and been prominently displayed throughout this work.

Funding

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and partially by Arm Sweden AB.

Popular Summary in English

In computer graphics, there are numerous small but important things that need to be considered when creating images of virtual scenes: Some are related to what we actually want to show in our images, and some are used to generate the images as quickly and efficiently as possible.

There are many algorithms and methods that can be used to create these images, but *ray-tracing* in particular is one of the most flexible choices: By approximating light-rays as straight lines that are sent through each pixel in our image and out towards the virtual objects in our scene, it is relatively easy to simulate a large number of real phenomena, some of which are illustrated in figure 1. Ray-tracing has even proven to be useful enough that dedicated hardware has been developed for it, just to make using it all the more efficient. Thus, even if new and improved algorithms come along, as long as they can be expressed in terms of these approximate light-rays, they can still benefit from this hardware.

To that end, in this work a collection of new techniques related to ray-tracing have been developed. Starting with a novel method for visualizing *Cherenkov radiation*; an almost eerie blue light typically seen around active nuclear reactors in the trail of charged particles moving at very high speeds.

Dedicated ray-tracing hardware can often assist these algorithms, but sometimes it needs some help along the way: Most forms of ray-tracing use a data-structure known as the *acceleration structure*, which as the name suggest, accelerates the processing of each of the virtual light-rays by reducing the total number of virtual objects they have to consider. However, this structure can be relatively complex, and must be built before ray-tracing can be done. Thus, in this thesis we have developed a simple way for speeding up the construction it.

Unfortunately, even with a good acceleration structure some types of objects are surprisingly challenging to create efficient images of, such as *grass* and *foliage*, as their small, numerous, and partially-transparent aspects forces the acceleration structure to do much more work than necessary. To that end, a lightweight data structure, known as *opacity micromaps* was introduced to shrink each individual leaf or straw to simplify this work as much as possible, and with the techniques described in this thesis, the micromaps themselves can be greatly simplified.

This micromap structure can even be used for other purposes, such as describing how an object should interact with our virtual light-rays, and since the micromap structure is closely aligned with the ray-tracing hardware, it can enable very fast operations on a selection of ray-tracing algorithms.

Finally, another small but important aspect of this work is making sure that the data needed

for these types of algorithms are actually readily available, and that appropriate scenes with these virtual objects can easily be found to evaluate new algorithms on. As a result, part of this thesis include a set of tools for the well-known *Blender* framework that aims to simplify these issues.

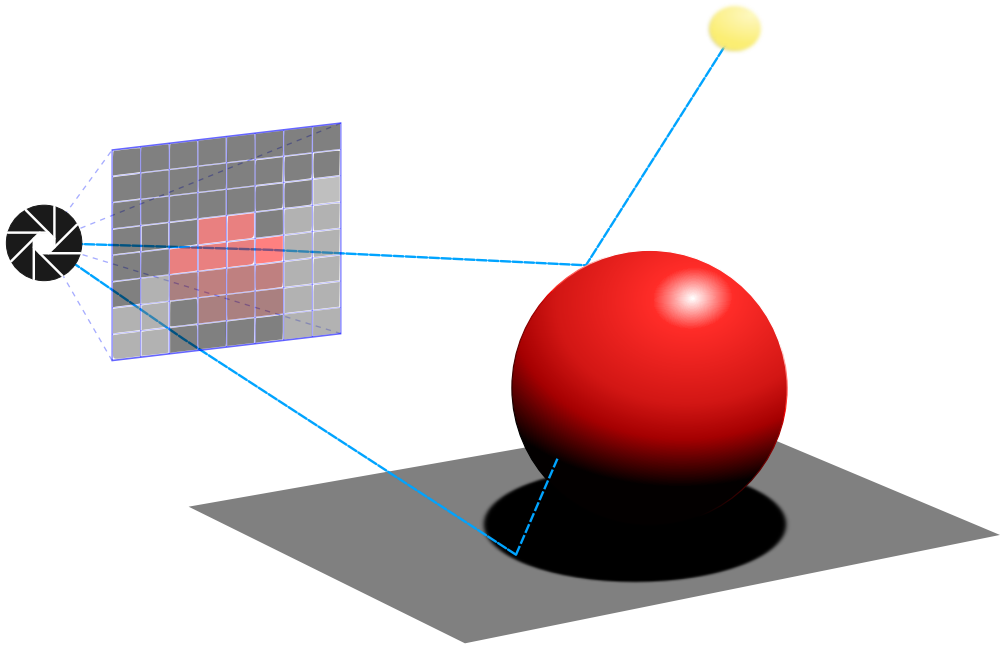


Figure 1: In computer graphics, a large number of lighting phenomena can be simulated by treating light-rays as straight lines by sending such rays through each pixel in an image and then allowing them to interact with virtual objects. The shadows cast by the sphere in this figure is a good example of this.

Populärvetenskaplig Sammanfattning på Svenska

Inom datorgrafiken finns det många små, men viktiga saker som man måste ha i åtanke när man skapar bilder av virtuella scener: Vissa av dem är relaterade till vad vi faktiskt vill visa upp i bilderna, och andra till hur de ska kunna genereras på ett så snabbt och effektivt sätt som möjligt.

Under årens gång har många algoritmer och metoder utvecklats för att skapa sådana bilder, men det har visat sig att *strålspårnings*-algoritmer (eng. *ray-tracing*) är bland de mest flexibla valen: Genom att behandla ljusstrålar som raka linjer från en tänkt kamera och vidare ut genom varje bildpixel mot de virtuella objekten i scenen kan en mängd olika ljusfenomen simuleras, några av vilka kan överskådas i figur 2.

Strålspårning har även visat sig vara en tillräckligt användbar metod för att motivera utvecklingen av speciella hårdvarukomponenter för att göra denna typ av algoritmer snabbare och mer effektiv. Nya och förbättrade metoder kan därav också dra nytta av dessa komponenterna: Det enda de behöver göra är att se till att de uttrycker sig i termer av dessa virtuella ljusstrålar.

Denna avhandling presenterar således en samling nya tekniker relaterade till strålspårning: Först ut bland dessa är en ny metod för a simulera och visualisera *Cherenkovstrålning*; ett nästan kusligt blått ljus som vanligtvis enbart syns i närheten av aktiva kärnreaktorer i spåren av laddade partiklar som rör sig i mycket höga hastigheter.

Den moderna strålspårningshårdvaran kan ofta påskynda dessa algoritmer, men ibland behöver även de hjälp på vägen. De flesta former av strålspårning använder sig av en så kallad *accelerations-datastruktur*, vilket som namnet antyder, accelererar behandlingen av de virtuella ljusstrålarna genom att minska antalet virtuella objekt de behöver ta hänsyn till. Dessvärre är denna struktur ganska komplicerad och måste byggas i förväg innan strålspårningsprocessen ens kan börja. Således presenterar denna avhandling en simpel metod för att påskynda dess konstruktion.

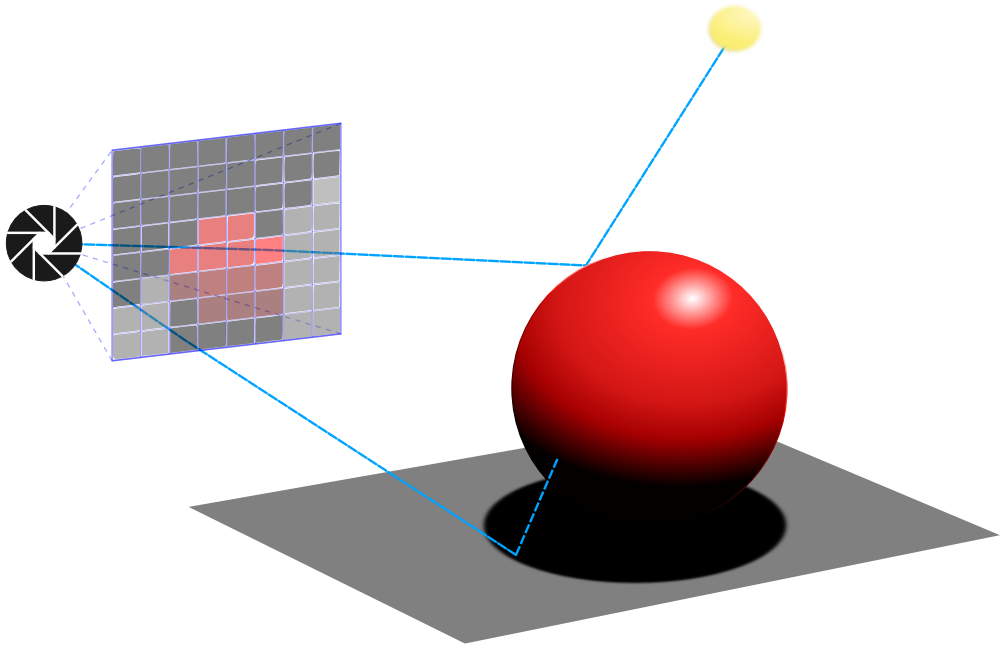
Olyckligtvis är vissa vanliga typer av virtuella objekt som *gräsmattor* och *lövverk* fortfarande förvånansvärt svåra att snabbt och effektivt göra bilder av med strålspårning, helt på grund av den stora mängden små och halvt genomskinliga grässtrån eller löv som de består av.

För att råda bukt på detta utvecklades en ny typ av hårdvaru-nära datastruktur: Den så-kallade *opacitetskartan* (eng. *opacity micromap*), vars uppgift är att försöka krympa varje grässtrå eller löv så mycket som möjligt, allt för att förenkla arbetet för accelerations-datastrukturen. Allt detta utan att själv använda särskilt mycket extra datautrymme, och i denna avhandlingen visar vi hur detta utrymme kan minskas ytterligare.

Samma typ av datastruktur kan också användas för andra ändamål, exempelvis för att

beskriva hur våra virtuella ljusstrålar ska interagera med objekt i scenerna, och då strukturen redan är väldigt nära knuten till strålspårningshårdvaran kan den således användas för att utveckla väldigt snabba metoder för specifika typer av strålspårningsalgoritmer.

En sista liten, men mycket viktig detalj inom detta område är att försäkra sig om att all data som behövs för dessa algoritmer är lätt att få tag på, och att det är enkelt att testa dem på lämpliga scener. Följaktligen inkluderar denna avhandling ett par verktyg till det välkända mjukvaruramverket *Blender* för att underlätta just dessa problem.



Figur 2: Inom datorgrafiken kan ljusstrålar behandlas som raka linjer: Genom att skicka ut dem genom varje pixel i bilden och därefter låta dem interagera med virtuella objekt kan en rad olika ljusfenomen simuleras. Skuggan under sfären i denna figur är ett bra exempel på detta.

List of Publications

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

I **Photon Mapping Superluminal Particles**

Gustaf Waldemarson and Michael Doggett
Eurographics 2020 - Short Papers

II **Parallel Axis Split Tasks for Bounding Volume Construction with OpenMP**

Gustaf Waldemarson and Michael Doggett
Proceedings of the 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - GRAPP

III **Succinct Opacity Micromaps**

Gustaf Waldemarson and Michael Doggett
Proceedings of the ACM on Computer Graphics and Interactive Techniques

IV **Color and Attribute Micromaps**

Gustaf Waldemarson and Michael Doggett
In submission

Related Publications

US Patent 2025/0157145 A1 [84] **Graphics Processing**
Gustaf Waldemarson *Based on the work of Paper III*

Author Contributions

Table 1: Summary of the contributions to the scientific publications included in this thesis made by the author as symbolized by the filled in octants of a circle for the various phases of the publication.

Paper	Concept	Implementation	Evaluation	Writing
Paper I (2019–2020)	◕	●	●	◕
Paper II (2024–2025)	◑	●	●	◕
Paper III (2023–2024)	◕	●	●	◕
Paper IV (2025–2026)	◕	●	●	◕

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, editing and reviewing the paper.

Background

Scientia est potentia. — Knowledge is power.
- Sir Francis Bacon

1 Overview

Today, images generated with three dimensional graphics are ubiquitous around the world, appearing in many wildly different contexts, ranging from education and work, to many forms of entertainment. These images must all be created somehow however, and in the particular case of *computer graphics*, this is done by creating digital images in a process referred to as *rendering*.

There are many methods for creating such images, and numerous details to consider for each of them, such as what subjects they contain, what material are those subjects made from, how do they interact with one another, and so on.

This particular thesis will focus on one such method that approximates the propagation of rays of light in virtual three-dimensional scenes. A method typically known as *ray-tracing*. The core of the thesis relates to the rendering of images with very *small* things whose large-scale effects have thus far been overlooked, or *small* things that are critical for the efficient rendering of these images. The former of these relate to the concept of capturing light emitted by high-energy particles, typically known as Cherenkov radiation, and in the latter case, the problem of efficiently rendering multitudes of small and overlapping objects, such as the leaves that make up grass and foliage.

Outline

The thesis background will start with a historic overview of graphics, leading up to what is considered the start of the field of *computer* graphics. This is followed by a brief background into the various sub-fields that directly relate to the included articles: Section 2 will introduce the concept of digital images, which everything else is built upon. Section 3 will present a selection of the components that form the basis of the virtual three-dimensional scenes that images are often made from before the details of the light propagation algorithms provided by *ray-tracing* are described in section 4, including how modern versions utilize hardware to accelerate the process. Finally, section 5 will briefly present the scientific methodology applied on the included papers and which more broadly is used to evaluate advances in computer graphics.

1.1 Computer Graphics

Some of the oldest relics of humanity have been found in the form of art, such as the geometrical engravings found in the Blombos caves [24], or the Sulawesi and Chauvet cave paintings [6, 46], all of which date back tens of thousands of years. These artworks are all arguably examples of *graphics*, that may have been created to simply appreciate the surrounding beauty in nature, or they may have been used to convey important information to future generations that has now been thoroughly lost in the mist of time.

Throughout the ages, we have repeated this process many times over. We have simply changed the medium on which we create our graphics, but it would be many millennia until we started using machines to generate our artworks, such as with the Jacquard loom from the 19th century [12].

With the invention of the computer, we found even more inventive ways to create art, such as using oscilloscopes to draw geometric shapes and images [34], but the work by Sutherland [44] is often considered the start of the field of *computer graphics*. He introduced several foundational ideas that are still core to the field, such as how to draw directly on a screen, how to create virtual scenes and objects and how to create geometrical relations between them. Most notably, he demonstrated *interactive* graphics, showing that it is not limited to static imagery, but can be used to create new images in real-time.

In the decades that followed, an enormous amount of work was made by both academia and industry to use computer graphics for a wide variety of things, such as to explain difficult concepts or visualize data complex data, to provide early visualizations of new products without expensive prototypes, or even to create safe virtual environments to practice dangerous skills, such as piloting aircraft. It even became one of the primary drivers for many forms of entertainment, such as video games.

The devil is in the details however, and a surprising number of them are needed for modern graphics to look correct. Consequently, this thesis includes a provide a brief background into computer graphics and the sub-fields that directly relate to the included articles such that an interested reader with a technical graduate education should be able to follow it. For more detailed explanations, I would recommend textbooks on these topics, such as PBRT [40], or online resources like the *The Graphics Codex* [37].

To begin, the foundation upon which all of computer graphics is based upon will be presented: The digital image itself.

2 Images, Colors and Blending

Digital images are traditionally made from a two-dimensional grid of **picture elements**, typically referred to as *pixels* that each hold some *color*, which itself is usually a triplet of scalar values that represent the amount of red, green, and blue light that shines through the pixel, as is shown in figure 3.

Occasionally, a fourth component known as the *alpha* channel is added to describe the opacity of the color. This value allows colors from multiple images to be blended together, or equivalently, allows one image to be shown on-top of another one, as illustrated in figure 4.

The exact values of the underlying scalars can vary a lot depending on the context the images are used in, but in this work we will assume that all colors, with or without alpha, are represented with scalars in the range of 0.0 to 1.0, or sometimes from 0.0 to ∞, and that they depict a linear change in intensity. In other words, it is a linear *color space*.

This is generally a major simplification, as a lot of work needs to be done for converting colors between various formats, and even when displaying them on physical prints or computer monitors in a way that our eyes can actually interpret, all of which falls under the topic of *color science*. These details are obviously very important for computer graphics, but in this thesis we can disregard these issues and simply treat all images as if they belonged to this idealized linear color space.

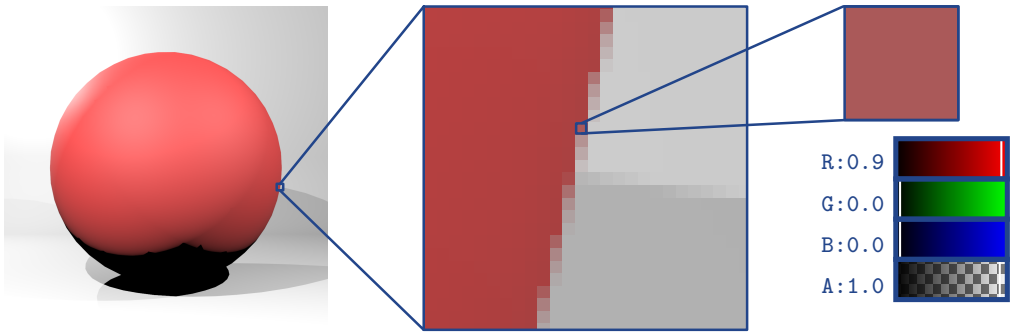


Figure 3: In computer graphics digital images are normally constructed from a two-dimensional grid of *pixels*, each of which can contain a *color* which in turn is often represented by up to 4 numbers: The intensity of the red, green, and blue light that shines through the pixels, with a fourth optional value that can be used for *blending* colors, as is described in figure 4.

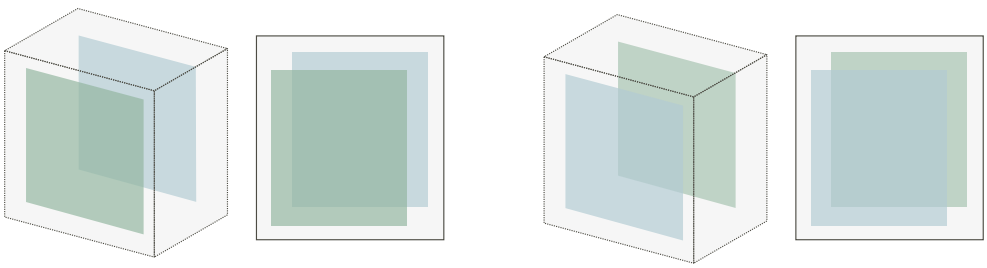


Figure 4: The colors from multiple images can be blended together to give the impression of partial transparency. However, the order of the blending is important: Blending a green color with a blue one (**left**) will not necessarily produce the same result as blending a blue color with a green one (**right**).

3 Virtual Scenes and Objects

Most images, real or otherwise, are some kind of interpretation or projection of one or more objects that we typically refer to as a *scene*. Similarly, digital images made with computer graphics are generally derived from virtual scenes with virtual objects. Thus, in order to render such an image, we first need a way of representing both of these things. However, they are not the only thing we need: The projection of the scene unto the image is controlled by what we call a *camera-model*, and without *light sources* to illuminate the scene, such a camera would typically only generate dark images. Consequently, a *scene* in this context can be thought of as a container for a camera and a collection of light sources and objects that we want to generate digital images from, and this section will briefly describe each of these.

3.1 Geometry

Over the years, a large variety of objects have been used within computer graphics, but none of them have proven to be as versatile as the humble *triangle*. With three non-colinear points, usually referred to as the triangle *vertices*, it is possible to define a subsection of a three dimensional plane in a very compact way, making it a useful primitive for many algorithms. A triangle is not particularly interesting by itself, however. It is only when composited together with many other ones that they truly become useful in the form of triangle-*meshes*: A structured group of triangles that collectively describe the surface of much more complex objects, as shown in figure 5.

However, meshes described purely with points typically have a very faceted appearance due to these planes, particularly if the individual triangles are few or very large. For this reason, additional *attributes* are often added to each vertex of the mesh to describe more granular aspects of the surface, such as its *color*, or the *surface normal*, which defines how the surface should *appear* to be oriented at each of the vertices. However, to avoid the facets, we also need a method for computing these attributes at an arbitrary point on the triangle, and not just on the vertices. Thus, at each point X inside a triangle with points P_0, P_1, and P_2, three sub-triangles can be formed, the area of which in relation to the original forms the

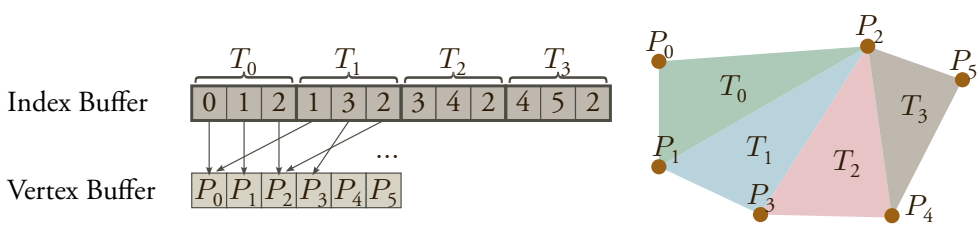


Figure 5: An example of a triangle mesh as they are often used computer graphics: Every set of three indices from the *index buffer* identify which three points P_i from the *vertex buffer* that constitute a triangle T_j.

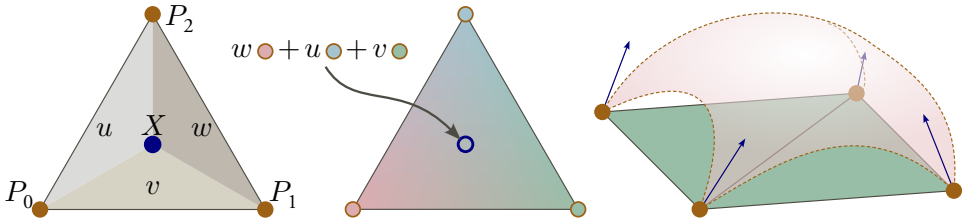


Figure 6: Schematic view on the interpolation of triangular surface attributes using normalized barycentric coordinates. The left figure describe how the normalized barycentric coordinates u, v, and w are constructed from the area of the sub-triangles from the points P_0, P_1, P_2, and X. The center figure shows how these coordinates are applied on color attributes from each of the vertices to blend these colors over the triangle surface. Finally, the right-most figure describes how the same interpolation scheme can be applied to a *surface-normal* vector to give the appearance of a curved surface.

so-called *normalized barycentric coordinates* u, v, and w:

$$\begin{cases} u = \triangle P_0 X P_2 \,/\, \triangle P_0 P_1 P_2 \\ v = \triangle P_0 X P_1 \,/\, \triangle P_0 P_1 P_2 \\ w = \triangle P_1 X P_2 \,/\, \triangle P_0 P_1 P_2 \end{cases}$$

These coordinates can in turn be used to smoothly interpolate each attribute at any point on the surface by multiplying the coordinate with the attribute from each vertex, thus, allowing colors to be smoothly blended, and generally creating the *appearance* of a much more detailed object, as is shown in figure 6.

It is also possible to map images onto the surfaces as well with the help of the *texture coordinate* attribute in a process that is described in more detail in section 3.5.

3.2 Cameras

There are numerous types of cameras that can be used in the field of computer graphics, but the arguably most important is the so-called *pinhole camera*, sometimes called the *camera obscura*. The earliest known descriptions of this camera date back to ca. 500 BCE with the Chinese Mozi writings [27] and Aristotle [3] ca. 300 BCE, with continued scientific descriptions by e.g., Ibn al-Haytham in the 11$^{\text{th}}$ century [23].

Physically, there are a number of ways to construct this camera, but it is typically formed by starting with a very dark box or room in which a tiny *pinhole* is made towards some lights and objects that we want to render an image of. If the box or room is dark enough relative to the outside, a faint image of the scene will be rendered on the wall opposite to the pinhole as depicted in figure 7. This image will be flipped both horizontally and vertically, but will typically be very sharp. In fact, the smaller the pinhole gets, the sharper the image. However, this will also result in less light physically making its way through the pinhole,

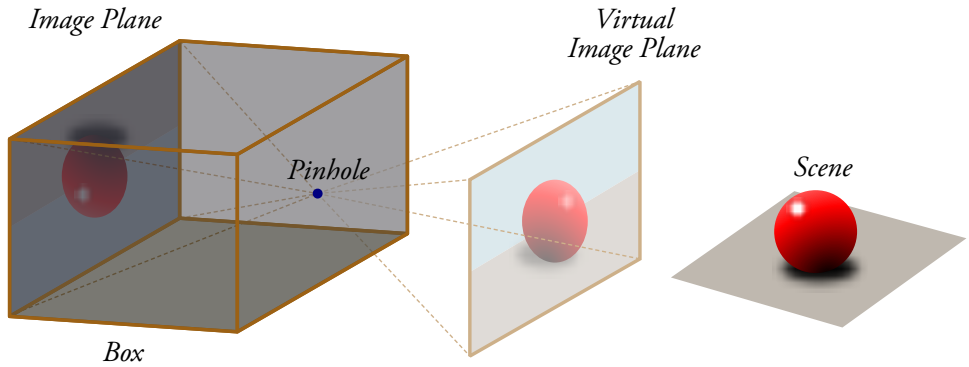


Figure 7: A simple box with a pinhole is the basis of the *pinhole camera model*: By letting rays of light go through the hole they cast a horizontally and vertically flipped picture of the objects outside the box on the image plane shown here. In computer graphics, this flipping can be corrected by moving this plane to a new location, appropriately called the *virtual image plane*.

making the image fainter. Consequently, it is difficult to create a real camera based on this principle.

These drawbacks are not a problem within computer graphics however, as the field is not bound by reality. As an example, the image-plane can be freely moved to any logically equivalent location, such as in-front of the camera where everything is "the right-side up". Similarly, the light-dampening effect of the pinhole can be completely ignored, creating a perfectly sharp image of the virtual scene. Consequently, the pinhole becomes the center of projection for the image, and is why this model typically refer to it as the camera center.

3.3 Light Sources

Similar to the geometry and cameras; numerous kinds of virtual light sources have been devised to simulate various kinds of effects, but in this thesis the only types that we care about are *point lights* and *area lights*, both depicted in figure 8. As their names suggest, a point light represents a source of light that is concentrated to a single point in space that emits light uniformly around it, and an area light-source is a light source that emits light from the surface of some object. Typically, this surface is defined by a triangle mesh, but it is also frequently represented as rectangles or disks as that can be more practical, and even faster for some algorithms.

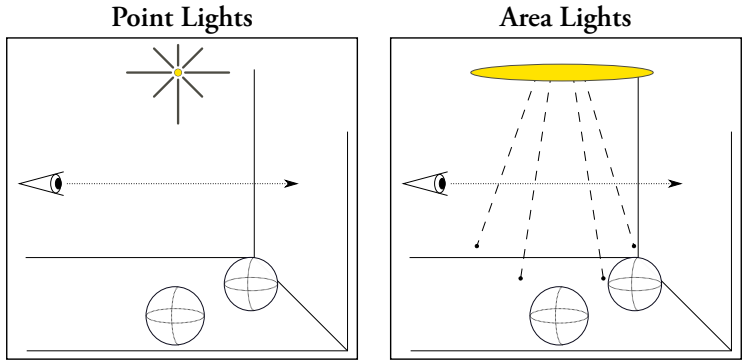


Figure 8: The two most common types of light sources in computer graphics are arguably the *point lights* (**left**) that emits light uniformly in all directions from a single point in space and *area-lights* (**right**) that emit light from the surface of an object, typically a triangle mesh.

3.4 Materials

Occasionally, attributes are used to describe how light should interact with surface geometry, e.g., by letting us see if the surface is matte or shiny.

Taken together, these attributes are often grouped into a concept referred to as *materials*, that collectively define how light should interact with the surface, or even more generally: How the energy that arrives at the surface is later radiated away from it. This concept is itself a large sub-field within computer graphics, consequently this work limits itself to two material models: The *Lambertian*, and the *Trowbridge-Reitz* model, or the *Ground Glass X* (*GGX*) model, which it is also known as. Both of these are schematically described in figure 9.

The *Lambertian* model, named after the mathematician Johann Heinrich Lambert [33], simply states that all energy that arrives to the surface is equally likely to be radiated away in any direction on the hemisphere of the surface, or equivalently: Light is scattered across the whole hemisphere. Consequently, this model gives the surface a matte appearance.

The *GGX* model [45, 54] is considerably more complex, and is better described in the more recent paper by Walter et al. [54], but it is relatively easy to get a feeling for how a simplified version of it works: First, the *Lambertian* model is used to describe the default surface interaction, but the matte appearance can now be modulated with additional attributes that changes how *rough*, and how *metallic* the surface should be, thus creating a much greater range of appearances, as can be seen in figure 10.

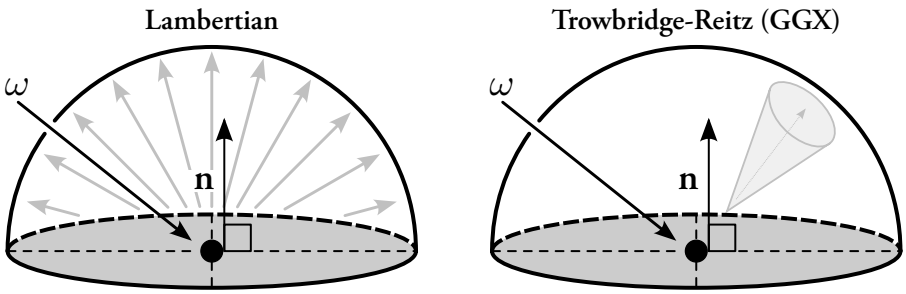


Figure 9: Overview of two common material models: The *Lambertian* model scatters light from the incoming direction ω uniformly on the hemisphere around the surface normal **n**, and the *Trowbridge-Reitz* (*GGX*) model, that can selectively change these scattering directions based on parameters such as *metalness* and *roughness*, as can be seen in figure 10.

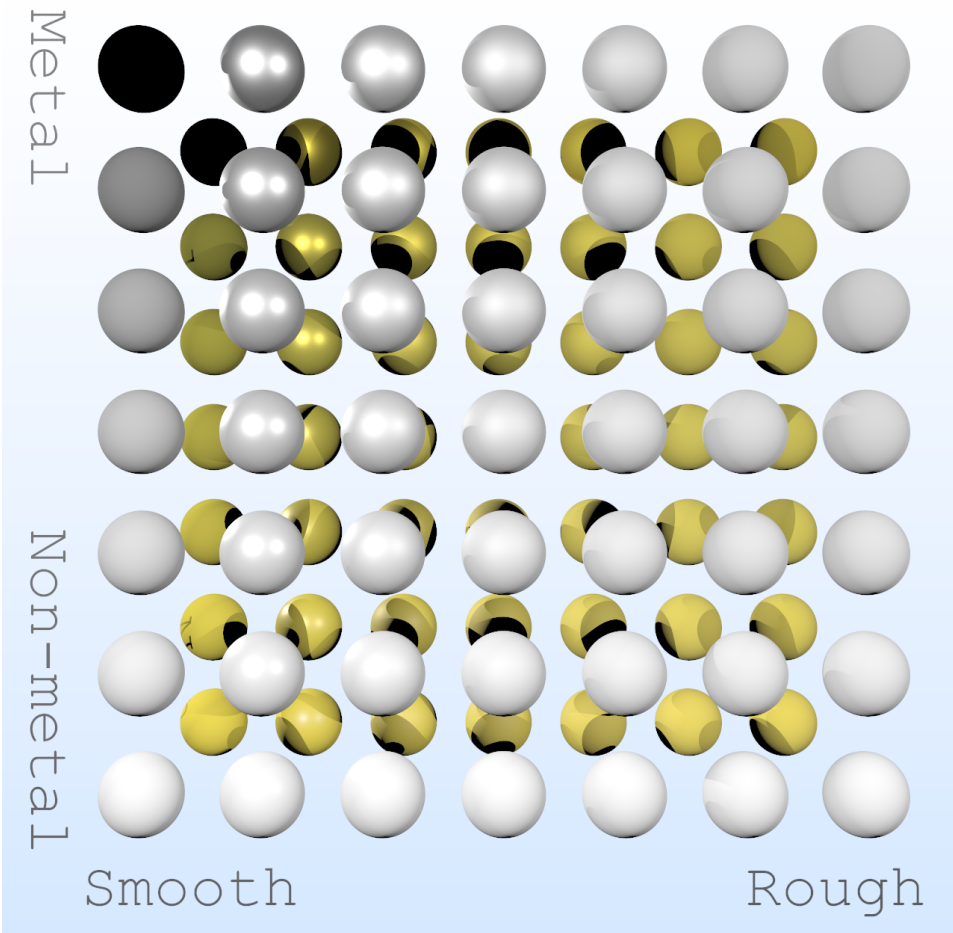


Figure 10: The Trowbridge-Reitz (GGX) material model [45, 54] has a number of parameters that can be changed to achieve various appearances on the same object, two of which are the *metalness* and *roughness* properties that are visualized here.

3.5 Textures

Vertex attributes can give triangle objects a surprisingly large amount of details, but sometimes, it is desirable to add more details than what those can provide. To that end, Edwin Earl Catmull [7] invented the concept of *texture mapping* that makes it possible to intuitively *map* existing images onto the surfaces of objects. Initially, this was only applied to rectangular objects, but eventually, *texture coordinates* were added as a vertex attribute to allow triangles to selectively map a sub-section of an image onto its surface, as can be seen in figure 11.

These images are usually referred to as *textures*, and they are not limited to changing the base surface look of an object: They can be also used to describe more granular *surface-normals* or materials properties, such as the *roughness* and *metalness* features from section 3.4. They can also be used to selectively turn parts of a triangle fully- or partially transparent, and even displace part of the triangle, both of which will be described in the next section.

Naturally, there is a trade-off between using detailed triangle mesh objects with many vertex attributes and having a simpler geometry with intricate textures. Which is used in practice often depends on the application or the personal taste of the asset creator.

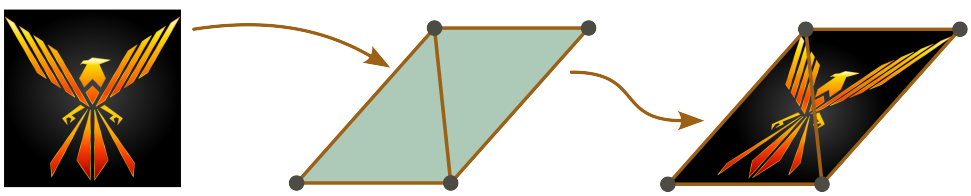


Figure 11: Given an image and a set of *texture coordinates*, it is possible to map this image onto the surface of a mesh. Here shown with only two triangles, but any arbitrarily complex mesh may be used for this purpose.

Alpha Masking

Mapping images onto objects may make them look a lot more detailed, but geometrically, they are still only made by triangles. This can be changed with the help of either, or both of so-called *alpha*- and *displacement*-textures, which use the same image-to-triangle mapping approach, but then uses various strategies to make the geometry appear even more intricate based on the texture data.

For *alpha*-textures, the image data tells the rendering algorithm that some parts of the geometry is fully- or partially-transparent, and that it should be possible to *see* through it, e.g., by using the blending techniques described in section 2. This is often used for foliage, where the leaves are constructed with a relatively simple geometry known as *cards* that enables some articulation and prevents them from looking completely flat. Then, a texture cuts out, or *masks* the precise parts of the geometry that we actually want to keep, thus creating the illusion of a very detailed object, particularly when repeated in multiple orientations, as is shown in figure 12. This approach is often known as *alpha-masking*, and is a technique that is improved upon in Paper III.

Displacement and Normal Maps

In contrast to *alpha-masking*, which only masks away parts of the geometry, *Displacement* textures modify it by indenting or extruding it based on the image data. Furthermore, the direction of the displacement can be controlled either by the vertex normals, or by another texture, typically known as a *normal map*. Consequently, these textures *actually* make the geometry more detailed, as can be seen in figure 13.

Some care is needed when rendering geometry with displacements however: In order to be efficient, most applications are initially only provided with a limited amount of knowledge of the geometry, such as the points that make up the triangles, allowing it to avoid processing them if they are not visible, or angled away from the camera. This works for most graphics algorithms, as they usually only modify the surface characteristics, not the geometry itself, or as is the case for *alpha-masking*, remains a strict subset of the original surface.

Displacements break this assumption as they can be used to deform the geometry arbitrarily. Thus, either this data must be made available to the application, or some other conservative estimate for the maximum displacements must be used to ensure correct rendering from all angles.

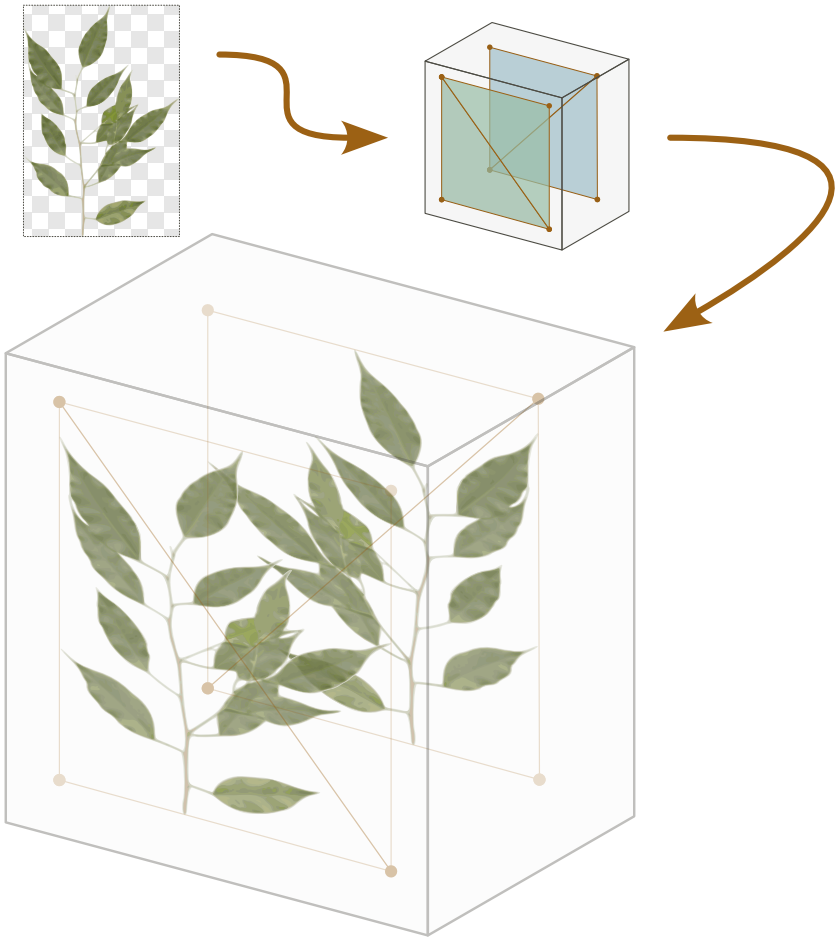


Figure 12: Textures can be used to turn parts of the geometry fully-, or partially transparent, a technique often used for foliage where large sections of leaves are represented with relatively few triangles, but in turn uses detailed textures to mask out the individual leaves, which in the field is often known as *alpha-masking*.

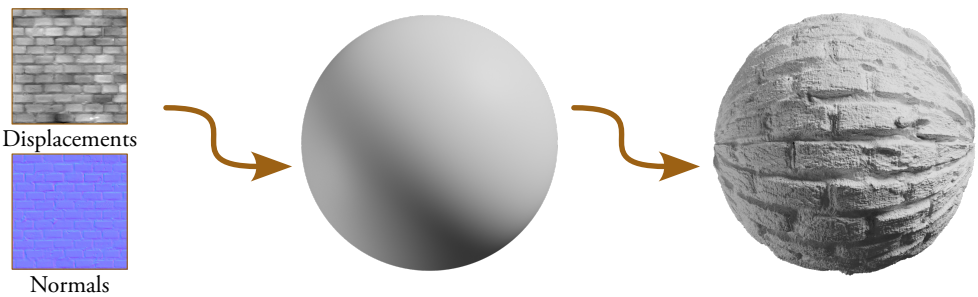


Figure 13: Textures can even be used to make the geometry itself much more granular. Here, one texture describes how far the surface should be indented or extruded, i.e., the *surface-displacement*. The second texture provides a more detailed *surface-normal*, which determines the direction of the displacements.

4 Ray-Tracing

In this thesis, a *ray* is mathematically defined by a single point, usually known as its origin O, and a direction D, that forms a line throughout 3D space. Consequently, any point on this line can be reached by starting at the origin and then walking some factor t of the ray direction, as is succinctly stated with the equation $R(t) = O + D \cdot t$ and illustrated in figure 14.

Thus, ray-*tracing* is simply the act of following such a ray to find some point that is of interest to us. A *ray* can represent any kind of real or virtual entity, but in the context of computer graphics and rendering, it is typically used to approximate the propagation of light-rays, as is discussed in more detail in the *Light Transport* section. However, it is also possible to use rays to approximate other physical quantities, such as atomically charged particles, as presented in Paper I.

4.1 Light Transport

The most fundamental thing that ray-tracing is used for is to approximate how much energy or light is radiated from a light source to a surface or from a surface onto another one. Typically, this concept is known as *light-transport*.

In computer graphics, this generally boils down to estimating how much of the light that is radiated by the light-sources in the scene actually reach the camera.

Consequently, the rendering process may start in either end: One may choose to emit virtual rays from the lights and attempt to link them to the camera, an approach typically referred to as *forward ray-tracing*. Rendering methods such as the photon mapping family of algorithms used in Paper I belong to this category.

Many of the rays emitted from these light-source never actually reach the camera however, so spending computational resources on them may be quite wasteful. Thus, somewhat

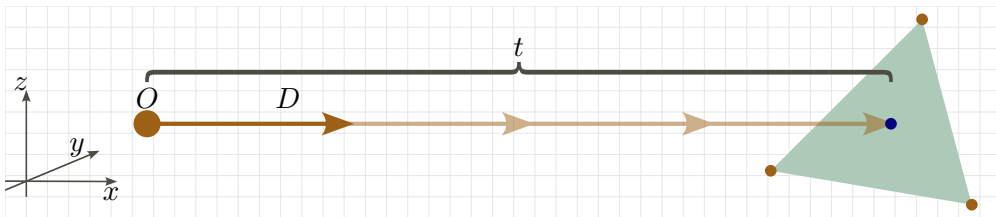


Figure 14: Mathematically, a *ray* consists of a point in 3D space known as its origin O, and a direction D along which it will propagate. The function $R(t) = O + D \cdot t$ can thus be used to describe any point on this line, such as the intersection point of a triangle t units further along the ray in this example.

unintuitively, it is often more efficient to start the process from the camera and attempt to find the light-sources from there instead, which is sometimes referred to as *backward ray-tracing*. Most ray-tracing algorithms belong to this category, one important of which is the *path-tracing* algorithm, first presented by Kajiya [28] together with the so-called *Rendering Equation*, that provided an intuitive mathematical view of the light-transport problem.

The Rendering Equation

All light that reaches some point will either be absorbed by the material, or radiated away again in some direction. This was succinctly summarized by Kajiya [28] in 1986, in the so-called *Rendering Equation*, a modern variant of which can be stated as:

$$L(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o) L(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n}) \, \mathrm{d}\omega_i \tag{1}$$

Where the various quantities of this equation can be described as:

$\mathbf{x}$	The point hit by a ray of light.
ω_i	The incoming direction of the ray hitting the point.
ω_o	The outgoing direction of the ray hitting the point.
Ω	The hemisphere centered around the point.
$\mathbf{n}$	The surface normal at the point.
L	The amount of light leaving, or arriving at the point.
L_e	The amount of light emitted by the point itself.
f_r	The Bidirectional Reflectance Distribution Function at the point.

This type of equation is known as a *Fredholm equation of the second kind* [14], i.e., an integral equation where the task is to find the function $L(\mathbf{x}, \omega)$ that satisfies it. This type of equation is generally very difficult to solve, and this particular one can only be partially solved in limited cases, such as for a perfect Lambertian sphere with known environments [9]. While it is hard to solve, it is relatively easy to get an intuitive understanding on how it works in practice:

At its core, it is all about conserving energy: All energy that arrives at a point $\mathbf{x}$ must either be absorbed or modulated by the angle it arrives at and the material at the point before it is re-emitted. Both of these aspects are covered by the function f_r, the *Bidirectional Reflectance Distribution Function* or *BRDF* at the point $\mathbf{x}$, which for most purposes can be thought of as the *material* from section 3.4 at that point.

Thus, in order to estimate the energy that leaves in a particular direction ω_o, we need to know the energy from all incoming directions w_i, i.e., *all* directions in the domain that

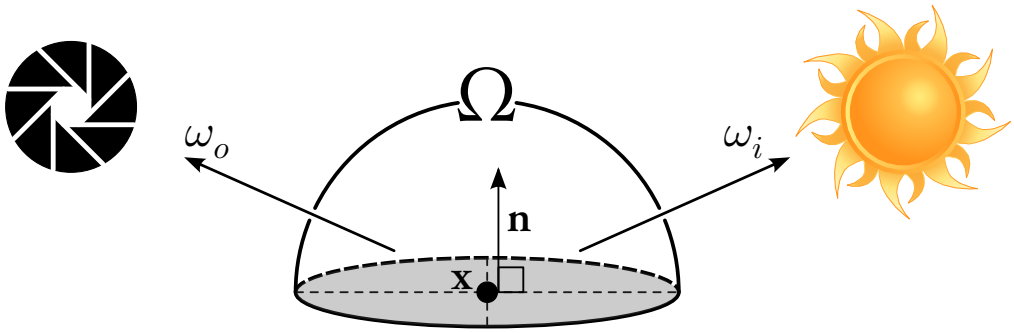


Figure 15: A schematic overview of the so-called Rendering Equation. The Light that leaves a point **x** with surface normal **n** depends on the light from all possible directions ω_i on the hemispherical domain Ω, modulated by the material itself at this point.

we are interested in, which is typically the hemisphere Ω around the point. If the point **itself** also radiates energy, i.e., it acts as a light-source, that energy needs to be added as well, which is captured by the L_e term. These concepts are shown schematically in figure 15.

Naturally, it is not feasible to investigate every possible direction, so various approximation have been employed over the years both before and after the introduction of the rendering equation.

Ray Casting One of the earliest example of this is sometimes known as the *Utah* approximation [28], here simply called *Ray Casting*: Here, a ray is *cast* from the camera out into the scene, and the light is approximated from there by **only** sending additional rays towards the light-sources in the scene and summing up each of their contribution. This can be stated as:

$$L(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + \sum_{\omega_l} f_r(\mathbf{x}, \omega_l, \omega_o)(\omega_l \cdot \mathbf{n}) \tag{2}$$

Where ω_l is the direction towards one such light-source. Note in particular that contributions from non-light-source directions that were previously captured by the $L(\mathbf{x}, \omega_i)$ term in equation 1 have been removed with this approximation. Consequently, it is not possible to describe phenomena such as reflections or *indirect* lighting with this approach.

Sampling Lights If the light-sources are area-lights, there are many directions that lead to the same light. In these cases, one typically sample a number of these to generate an average contribution. Similarly, if the scene contains a very large number of light-sources, it may be computationally unfeasible to send rays towards all of them, thus sampling only some of them and averaging their contributions may be necessary in practice. Thus, finding the light-sources that contributes the most to each point becomes an important issue, but in this work we will mostly disregard this problem as the number of light-sources considered here is sufficiently small.

Whitted The second, and probably most well known approximation to the rendering equation is known as *Whitted ray-tracing* after Turner Whitted [56].

Similar to ray-casting, this method sends rays from the camera, but after summing up the contributions from the surrounding lights, the algorithm checks if the underlying material is reflective: If that is the case, a new ray is emitted in the *reflected* direction, and the contributions from that directions is added based on the reflectivity of the material. Similarly, if the object is *transmissive*, such as for glass, a ray can be transmitted through the material as well, providing additional contributions from the materials *refracted* direction. Similar to ray-casting, this can be stated as:

$$L(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + rL(\mathbf{x}, \omega_r) + tL(\mathbf{x}, \omega_t) + \sum_{\omega_l} f_r(\mathbf{x}, \omega_l, \omega_o)(\omega_l \cdot \mathbf{n}) \tag{3}$$

Where r and t are the material reflectivity and transmissivity, and ω_r and ω_t are the reflected and transmitted directions.

Recursion Note that the function $L(\mathbf{x}, \omega)$ is re-used multiple times in equation 3 to create sharp reflections and transmissions, thus creating a *recursive* process. This could in theory repeat indefinitely, but for practical reason this is normally stopped by limiting it to a fixed number of terms, or when the cumulative weight from multiple jumps drops below some predefined threshold.

Distribution Ray-Tracing The Whitted approach can be further extended by sending additional rays in even more directions, creating what is sometimes known as *Distributed* or *Distribution* ray-tracing [11]. However, similar to the problem of sampling many light-sources, this quickly becomes impractical as the number of rays that needs to be traced grows exponentially after each recursion.

Path-Tracing Ray-casting and Whitted style ray-tracing can only provide rough approximations to the rendering equation, however, with its presentation, Kajiya [28] included an algorithm that provides a much more accurate approximation based on a randomization technique known as *Monte Carlo integration*. Given an arbitrary multi-dimensional integral, of the type:

$$I = \int f(\mathbf{x})\, d\mathbf{x} \tag{4}$$

It is possible to evaluate this integral as long as it is possible to perform repeated evaluations of the functions $f(\mathbf{x})$ at arbitrary points in the domain by using a so-called *Monte Carlo Estimator* function:

$$F_n = \frac{1}{n} \sum_{i}^{n} \frac{f(X_i)}{p(X_i)} \tag{5}$$

Where X_i is a uniformly sampled random variable inside the integral domain, with the normalized probability density function (PDF) $p(\mathbf{x})$, and n is the number of such samples. Furthermore, it can be shown that the estimate $E[F_n]$ of this function converges to the integral from equation 4:

$$E[F_n] = E\left[\frac{1}{n}\sum_i^n \frac{f(X_i)}{p(X_i)}\right] = \frac{1}{n}\sum_i^n E\left[\frac{f(X_i)}{p(X_i)}\right] = \frac{1}{n}\sum_i^n \int \frac{f(\mathbf{x})}{p(\mathbf{x})} p(\mathbf{x})\, d\mathbf{x} = \int f(x)\, d\mathbf{x}$$

Thus, in the limit with an infinite number of samples, this approximation converges to the integral, even if it is a complex one like the rendering equation.

This insight lead to the development of the *path-tracing* method, which similar to the ray-casting and Whitted approximations in equations 2 and 3 can be summarized as:

$$L(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + \frac{1}{n}\sum_i^n \frac{f_r(\mathbf{x}, \omega_r, \omega_o) L(\mathbf{x}, \omega_o)(\omega_r \cdot \mathbf{n})}{p(\omega_r)} \tag{6}$$

Where ω_r represent a random directions on hemisphere that is sampled from a distribution with the probability density function $p(\mathbf{x})$.

Similar to the Whitted approximation, *path-tracing* can easily handle effects such as reflections and transmissions, but also other phenomena such as *color bleeding* is trivially simulated. I.e., the effect of the strong color of one object diffusely reflecting onto another one. *Path-tracing* also avoids the computational explosion of rays this effect would need if simulated with *distribution ray-tracing*, as only a single *path* of rays is followed at any time, thus giving the method its name. This comes with one significant drawback however: The rendered images are no longer fully deterministic, and the quality of them depends on the number of samples that were used in the approximation. In general: The more complicated the lighting is in the scene, the more samples are needed to render it with acceptable quality, but in return, *path-tracing* is able to accurately approximate a vast number of lighting phenomena. A few example of such renders can be seen in figure 16.

Photon Mapping Path-tracing is often sufficient to faithfully render a wide variety of complex scenes, but there are situations were it struggles: Scenes where light must traverse or bounce off of one or more surfaces can be very difficult to render, as the paths from the camera are often terminated before they are able to find any light source. In fact, if the scenes are illuminated by point-lights, it can even be impossible to find such a path without directly sampling these light-sources, as is illustrated in figure 17.

An extreme case of this effect occurs when geometry focuses light into small regions, forming so-called *caustics*: A small, but prominently bright section of the scene. This effect *can* be rendered with Path-tracing, but an unfeasibly large number of samples are required to render it with acceptable quality.

This was improved upon with *bi-directional* ray-tracing approaches, such as *photon mapping* [26, 22, 21]: Instead of only following rays from the camera, another set of rays can be traced from light-sources as well. When rays from each of these sets are connected, the lighting contribution from a relatively complicated path can be added as a sample.

The *stochastic progressive photon mapping* (*SPPM*) [21] used in Paper I accomplishes this by first sending out a set of rays from the camera and placing a virtual and spherical *hit-point* at the intersected locations. Another set of rays are then sent from the light sources, and whenever they intersect with these *hit-points*, their lighting contribution is passed back to the camera, as is illustrated in figure 18.

This approach is substantially more complicated than *path-tracing*, and requires a lot of extra memory for storing the virtual *hit-points*. In return however, it is able to quickly sample relatively complicated light-paths, as can be seen in the renders in figure 19.

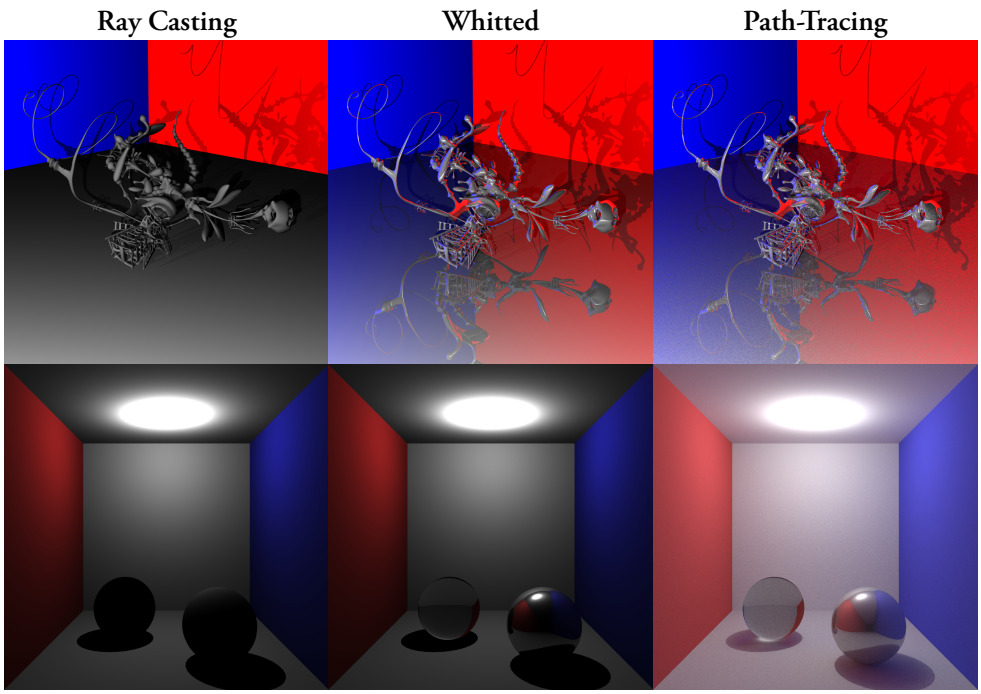


Figure 16: Comparison of three different approximations to the rendering equation for the *Yeah Right* [90] (**top row**) and the Cornell Box [18, 89] (**bottom row**) scenes. Note that *ray-casting* cannot simulate any kind of reflections or transmissions, both of which are prominent in the *Whitted* and *path-tracing* methods. *Path-tracing* can also recreate more subtle effects, such as *color bleeding*, see e.g., the ceiling next to the red and blue walls and compare them for the various approaches.

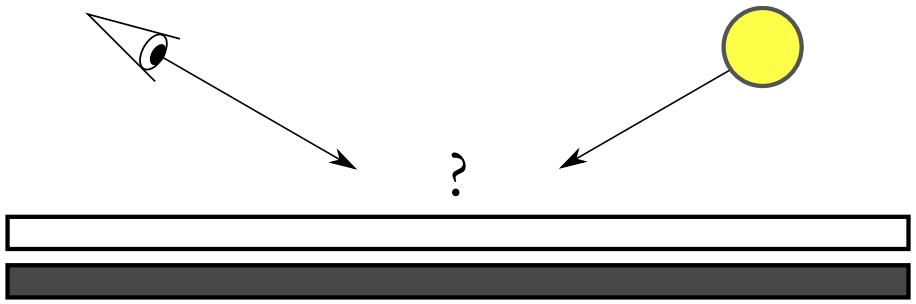


Figure 17: In this scene, a diffuse object (**black**) is fully covered by a perfectly transmissive slab of glass, with the camera and point-light above it. In this scenario, even *path-tracing* is unable to light this scene without special handling of point light-sources.

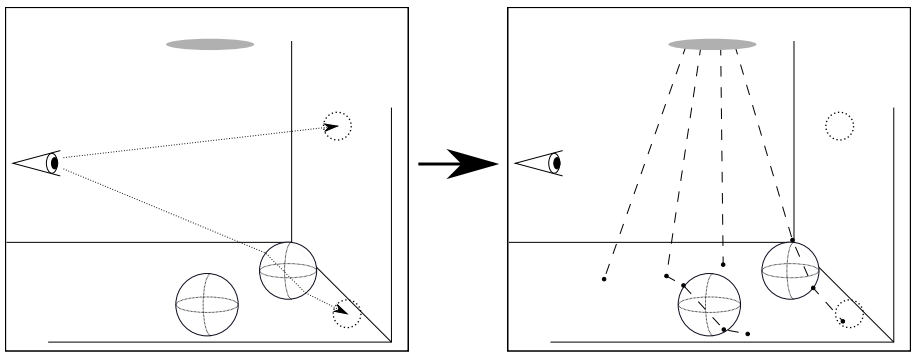


Figure 18: *Photon mapping* works by first sending out rays from the camera and constructing virtual *hit-points* at these locations (**left**). Subsequent rays sent from the light-sources that intersect with these spheres can then pass the lighting contribution from that light-source back to the camera.

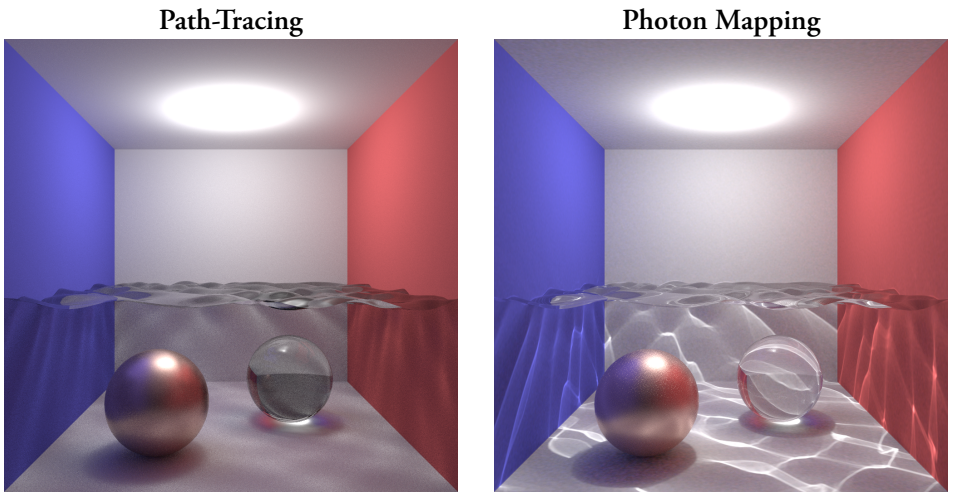


Figure 19: An illustrative comparison of images rendered with the *path-tracing* and *photon mapping* approaches for one version of the Cornell Box [89] with a refractive, semi-transparent, and wavy surface creating irregular patterns of caustics all over the floor of the box. In such a scenario, *path-tracing* is barely able to resolve any of these patterns, in stark contrast to the *photon mapping* method.

Ambient Occlusion

Ray-tracing, and particularly path-tracing, is able to generate visually impressive images by approximating the rendering equation. However, they are often computationally expensive to generate, and typically require the scene to be fully setup with geometry and proper materials and light-source before anything can actually be rendered. Consequently, alternative rendering methods that are *not* based on light-transport may be more appropriate at times.

Ambient Occlusion, or *AO*, is one such rendering method. Instead of computing how much light arrives at a point, it creates an estimate of how much the surrounding geometry occludes the point in question. Mathematically, this can be captured in an equation that is superficially similar to the rendering equation:

$$A_{\mathbf{x}} = \frac{1}{\pi} \int_{\Omega} V(\mathbf{x}, \omega)(\mathbf{n} \cdot \omega) \, \mathrm{d}\omega$$

Where:

$\mathbf{x}$	The point in question.
$\mathbf{n}$	The surface normal at the point.
Ω	The hemisphere centered around the point.
ω	The outgoing direction from the point.
$V(\mathbf{x}, \omega)$	The geometric visibility in the direction ω from the point.
$A_{\mathbf{x}}$	The ambient occlusion metric itself.

This is illustrated schematically in figure 20. Similar to path-tracing, the *AO* metric can be estimated using a ray-tracing based *Monte Carlo* method: Once a first intersection point is found, new rays can be emitted uniformly in the hemisphere around it. The ratio between the rays that intersect any adjacent geometry and all emitted rays effectively approximates the ambient occlusion metric.

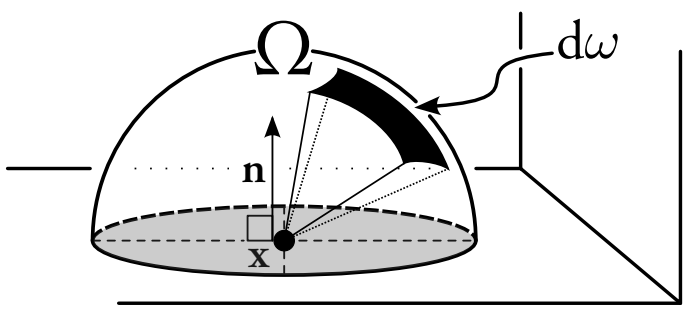


Figure 20: Schematic overview of how the ambient occlusion metric is found by estimating how much of the area $\mathrm{d}\omega$ of the hemisphere Ω around a point $\mathbf{x}$ with surface normal $\mathbf{n}$ is covered by adjacent geometry.

Unlit Rendering The ambient occlusion metric is rarely particularly interesting in and of itself, but it can be combined with many other material properties or rendering methods to provide a much greater sense of depth, without introducing any kind of light-transport. The easiest of this is an approach that is sometimes called *Unlit* rendering, where the material base-color is multiplied by the ambient occlusion, as shown in figure 21.

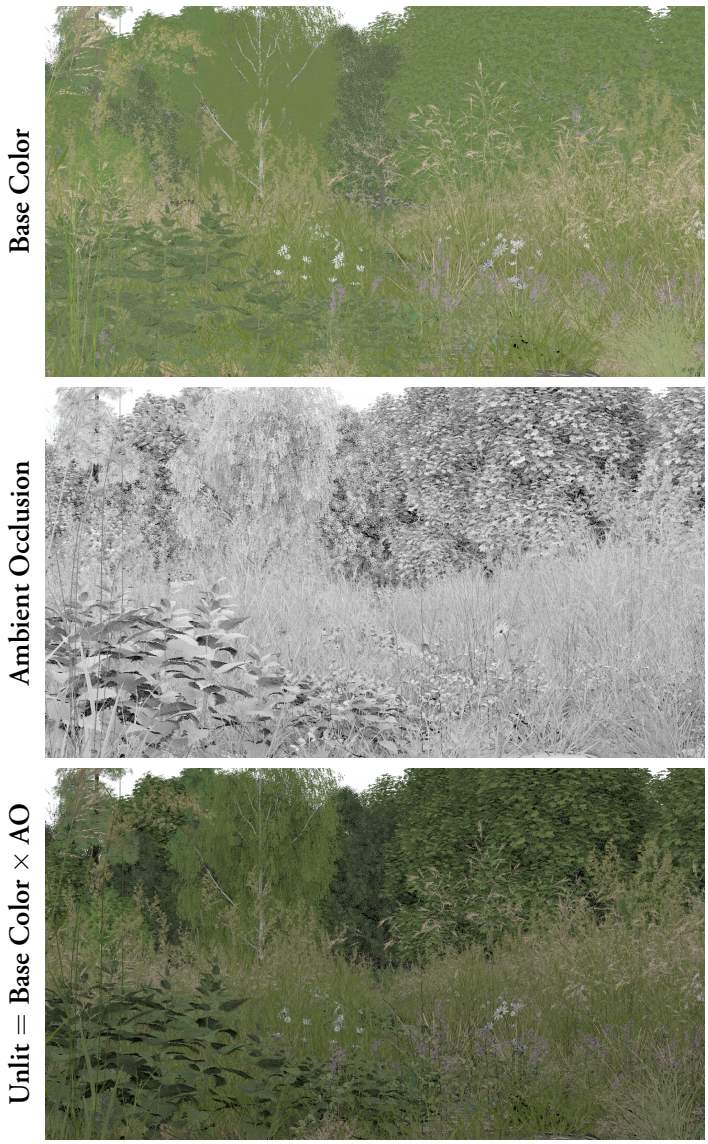


Figure 21: The "Landscape" scene [94] rendered with only the material base color (**top**) multiplied with the ambient occlusion metric (**middle**) can provide a convincing approximation to light transport algorithms, creating a method sometimes referred to as *Unlit* rendering (**bottom**), as there are no lights to illuminate the scene.

4.2 Acceleration Structures

At first glance, ray-tracing might seem like a simple and straight-forward approach to estimate light-transport, but scaling it can be problematic: It might be easy to follow a single ray, but a typical image consists of 1920×1080 pixels, and the light-transport needs to be estimated for each of them to fully render the image. Thus, at *least* one ray is normally sent through each pixel in the virtual image plane, as illustrated by figure 1. However, we typically need many of them, as we usually follow a *path* of rays, each of which can branch in unpredictable directions, and each of these paths only account for a single light-transport *sample*. A pixel may need hundreds of samples before it reaches an acceptable level, and sometimes more if the scene lighting is particularly challenging. All-in-all, it is not uncommon to trace millions to billions of rays in order to render a single image.

The primary objective for these rays is to find an *intersection* with the scene objects, e.g., the closest point from the ray origin to one of the triangles in a mesh in the ray direction, as shown in figure 14. However, a single scene may contain thousands of mesh objects, which can contain millions of triangles each, and may even be replicated multiple times at different locations in the form of *instances*, as summarized in table 1 for the scenes from section 6.

Table 1: The complexity of a scene can be roughly estimated by measuring a few of the key elements that it contains, such as the number of *materials*, *textures*, and *objects* described in section 3. This table shows these metrics for the scenes shown in table 5.

Scene	Materials	Textures	Instances	Meshes	Triangles
Killeroo	7	0	5	5	16 816
ArcSphere	5	0	61 708	26	52
LTE Orb	7	1	7	7	191 970
Zero Day	390	28	9119	7308	4 532 949
San-Miguel	291	370	160 727	803	2 501 620
Landscape	629	336	407 824	358	26 050 093
Transparent Machines	3	0	1114	678	2 386 019
Watercolor	193	124	1002	948	24 732 407
Kroken	81	30	305	250	40 199 746

Evidently, testing each mesh and triangle in turn to find the closest intersection is unfeasible in all but the simplest of cases. To address this, so-called *acceleration structures* were invented to drastically reduce the number of objects a ray needs to be tested against to find this point, or to determine that the ray does not actually hit the scene geometry at all. This section will briefly introduce three of these structures: *Grids*, *kd-trees*, and *bounding volume hierarchies*, all of which are illustrated in figure 22.

Grids

One simple way to reduce the number of required intersection tests is to group together adjacent geometry and surround them with a box: If the ray does not hit the box, it cannot

hit any of the geometry inside it either, thus effectively culling all of those tests.

The earliest example of this applied on graphics focused on subdividing the scenes evenly into three-dimensional grids thus creating implicit boxes for all geometry; forming what we today refer to as *uniform grids* [15, 1]. While typically efficient thanks to the simple intersection tests and spatial coherency, this kind of approach can suffer from the so-called "teapot in a stadium" problem [40]: A highly detailed object in an otherwise sparsely decorated scene that drastically affects performance whenever a ray enter the grid-cell it was placed inside. For this reason, adaptive approaches, such as *adaptive grids* and *octrees*, are often employed to alleviate this issue while maintaining most of the spatial coherency properties [16, 10, 42].

Kd-Tree

Instead of grouping objects together in grids, one can instead separate them in space with planes. This is the idea behind the *binary space partitioning* family of algorithms.

One simple way to build such a structure is by first creating a single box around all geometry, then splitting it with an axis-aligned plane to form two new implicit boxes. By repeating this process recursively, a *tree* of splitting planes gradually subdivide space until whatever volume remains only contain a few triangles, forming a leaf node. This particular type of space partitioning is usually known as *kd-trees* [5, 29].

Thus, a ray can determine if it hits any of the contained geometry by traversing the tree through intersections with the dividing planes: If at any point the plane intersection is outside the bounds of the initial box, the traversal can stop. Otherwise, it continues until a leaf node is encountered, at which point the triangles in that volume can be tested, and the closest intersection, if any, can be found.

Bounding Volume Hierarchies

Instead of separating objects in space, *bounding volume hierarchies* (*BVH*) group adjacent objects into explicit boxes. This avoids the problem from the grid based approaches as each box can now be resized based on how much geometry it contains. Furthermore, these boxes can themselves be placed inside *other* boxes, thus forming a type of three-dimensional search tree for ray-intersections [41, 31, 17].

Similar to *kd-trees*, a ray can quickly determine if it hits any of the geometry contained by a BVH by traversing this tree through intersection tests with each contained box: If a box is missed, it cannot hit any of the underlying geometry.

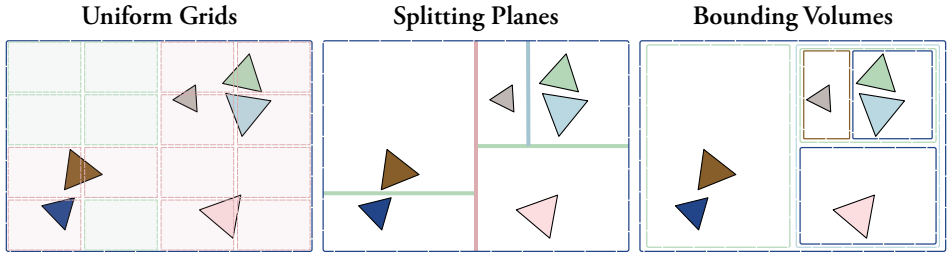


Figure 22: Schematic illustration of three common acceleration structures and the elements they use in their construction. *Uniform Grids* (**left**) store the geometry in uniformly sized cells. Here visualized with red for cells with overlapping geometry, and green for empty cells. Rather than cells, *Kd-trees* (**middle**) uses splitting planes to hierarchically separate geometry, and instead of planes, *bounding volume hierarchies* uses nested boxes to create a hierarchy to contain the objects.

Quality

Each of these approaches have different strengths and weaknesses and that can be visualized by counting the number of times rays interact with the various parts of the acceleration structures. E.g., if a scene is rendered with the ambient occlusion method from section 4.1 while counting the number of *triangle tests*, and *intermediate tests*, i.e., the additional tests against boxes or planes that are done, one can create a rough estimate of the *quality* of the acceleration structure. In table 2, these tests are averaged and visualized in the red and green channel for each pixel in the image, normalized across three typical accelerators. Thus, any pixel that is predominantly green primarily executes *intermediate tests* where one that is red mostly run *triangle tests*.

This metric can be useful for finding regions that are potentially troublesome for performance, but it is also useful for comparing the quality of different kinds of accelerators:

In these scenes, the "teapot in a stadium" problem that *uniform grids* can suffer from is particularly obvious, as is evident from the almost exclusively red hue. Which accelerator that is best between *kd-tree* and *BVH* frequently varies from scene-to-scene, with perhaps a slight edge to *kd-trees* in these cases, as can be seen from the slightly darker green tinge; signifying fewer tests in general. In practice however, *BVHs* often end up being more useful in general. Particularly for their simultaneous use in collision detection systems and comparatively fast rebuild times [48]. Consequently, subsequent research has primarily focused on this structure and modern variations on this approach can be surveyed in the work by Meister et al. [38].

Split Methods

On the surface, grouping together adjacent triangles might seem easy, but it is a detail that can be crucial to the performance of the bounding volume hierarchy. Consequently, a number of heuristics have been developed to group them together quickly and with as high quality as possible, as it is currently believed that finding the "best" possible grouping is an *NP-hard* problem [30].

One typical grouping approach is to work in a top-to-bottom fashion: Starting with a list of all primitives, they are first sorted along one of the axes, and then divided into *left* and *right* subsets. This can then be repeated recursively until only a few objects remain, thus forming a hierarchy of boxes. However, dividing the objects into these subsets can be quite tricky:

In the simplest case, the objects are split into two subsets of the same size, an approach known as the *median-split*. This is fast and works well for uniformly distributed objects, but may create large overlaps between the subsets when the objects are more clustered, as is often the case in practice.

A more balanced approach is to search for the first object in the ordering that overstep the midpoint among all objects, and perform the division from there. This is known as the *midpoint-split* method.

Yet, it is still possible to find better divisions: Whenever an object is added to the *left* or *right* subset, the box corresponding to it grows accordingly. The *Surface Area Heuristic* (*SAH*) method attempts to find the split location where the size of these boxes are smallest; thus minimizing the probability that a ray will intersect with any of them [17, 47].

All of these approaches are illustrated in figures 23 and 24, and their impact on the quality of a BVH can be seen in table 3.

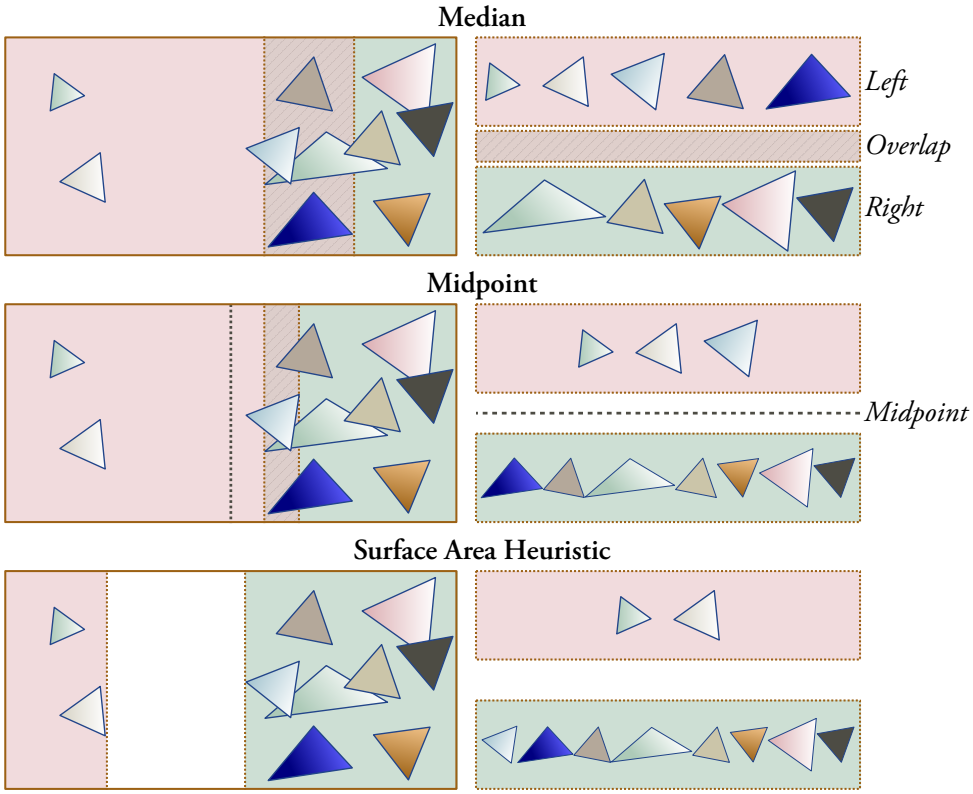


Figure 23: A bounding volume hierarchy built in top-to-bottom fashion can divide its sorted objects in various ways. The *median* split (**top**) creates two equally sized subsets, while the *midpoint* split (**middle**) searches for the first object to exceed the total midpoint. The *surface area heuristic* (**bottom**) systematically searches for the object division to minimize the subset surface areas, and by extension, the probability that a ray will intersect with that box.

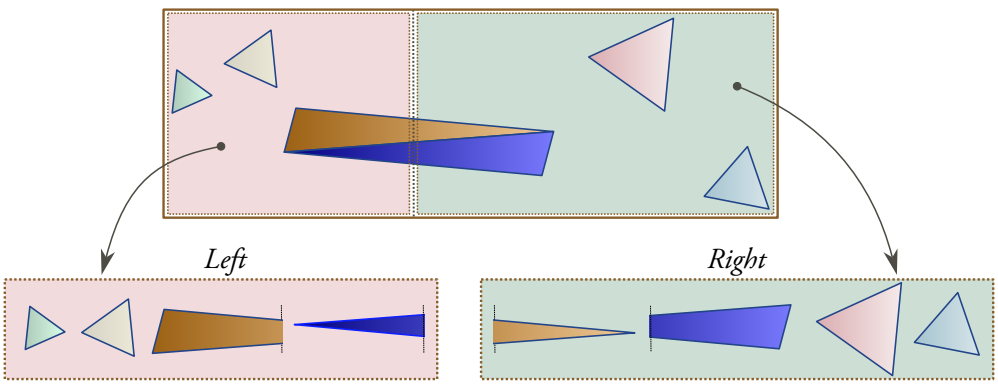


Figure 24: *Spatial splits* effectively divide the objects themselves, thus allowing the boxes around each of the *left* and *right* subsets to become even smaller.

Spatial Splits

The *SAH* split method is often the best possible heuristic for dividing objects into minimally sized subsets, but with additional information about the underlying geometry, it is possible to improve this further.

As an example, for meshes it is fairly common for triangles to become intertwined as seen in the middle of the scene from figure 24. The ideal split would be in the middle of these, but other heuristics typically put both of them in one of the two subsets.

This situation can be addressed with a construction method known as *spatial split bounding volumes*, typically abbreviated as *SBVH* [43], which employs a splitting method called a *spatial split*. In essence: A level of indirection is added to the BVH. Instead of storing objects, it stores *references*, i.e., a link to the primitive and the ideal box that surrounds it. From there, it is possible to shrink that box if it can be proven that that particular segment cannot be hit. Thus, whenever the situation from figure 24 arise, *references* to the objects are added to both subsets, but each idealized box is shrunk against the boundary of the split such that the object itself is effectively split along that boundary. The overall effect is that the number of *intermediate tests* are reduced, and thus the quality is improved a bit further, particularly for larger scenes, as is evident in table 4 from the slightly dimmer green hues.

Additionally, some objects can shrink this idealized box even further when it knows it only contains a small part of the object. E.g., triangles are easily clipped to a plane boundary, thus making it possible to create even tighter boundaries around it compared to just splitting the box itself, as is illustrated in figure 25.

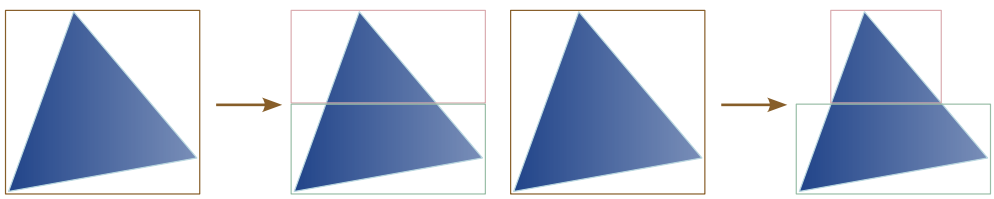


Figure 25: A *spatial split* of the box surrounding an object will always shrink it to that boundary (**left**). However, with additional information about the internal geometry, such as that of the triangle clipped against the splitting plane, the box can be made even tighter (**right**).

Table 2: Visualization of the *quality* of three common acceleration structure types. The red color channel represents the number of triangle tests in a pixel and the green color the intermediate tests when rendering a scene with ambient occlusion. Both are normalized against the maximum number of tests as found in the colorbars for each scene.

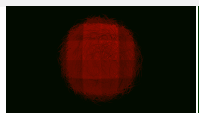
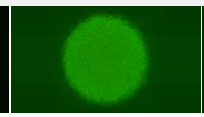
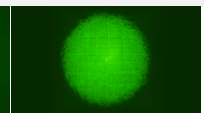
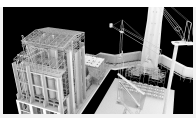
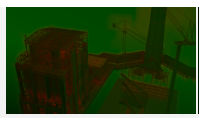
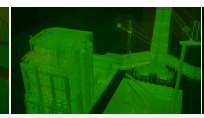
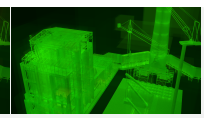
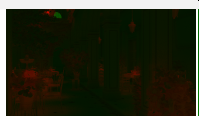
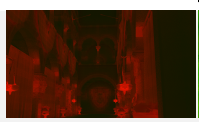
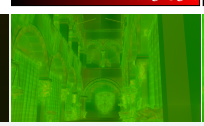
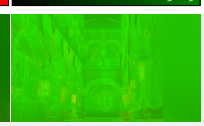
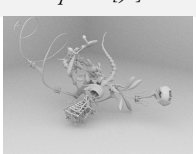
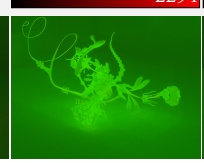
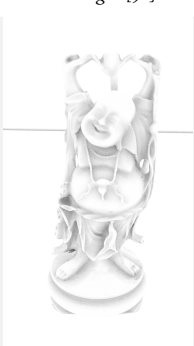
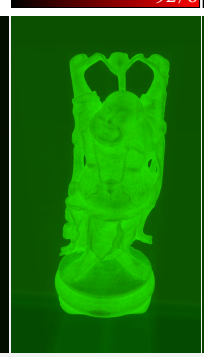
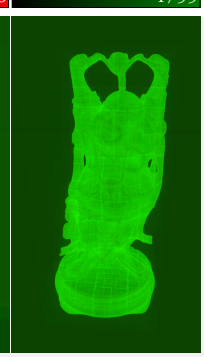
Render	Uniform Grid	Kd-Tree	BVH
 *Bistro [88]*		 30 796	 3692
 *Hairball [32, 97]*		 98 522	 3744
 *Powerplant [93]*		 50 496	 3180
 *San-Miguel [96]*		 223 632	 2545
 *Sponza [92]*		 2294	 1736
 *Yeah Right [90]*		 9276	 1755
 *Buddah [99]*		 25 842	 1716
		Triangle Tests	*Intermediate Tests*

Table 3: Visualization of the quality for three common split methods used by the Bounding Volume Hierarchy acceleration structure. The red color channel represents the number of triangle tests in a pixel and the green color the intermediate tests when rendering a scene with ambient occlusion. Both are normalized against the maximum number of tests.

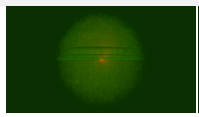
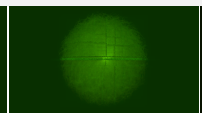
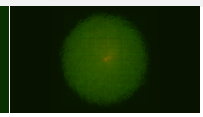
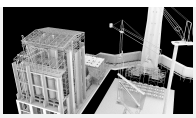
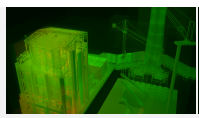
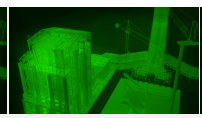
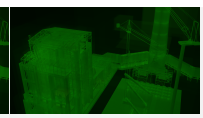
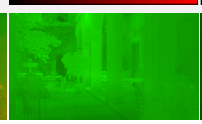
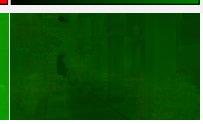
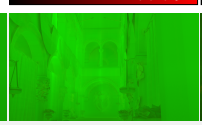
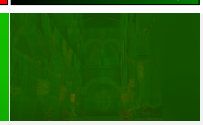
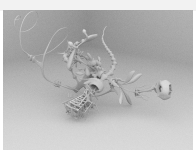
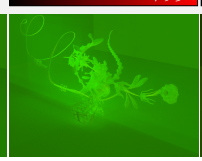
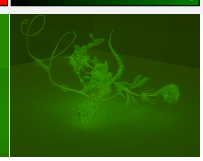
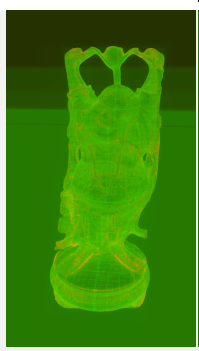
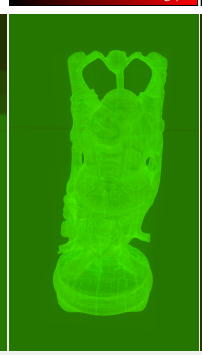
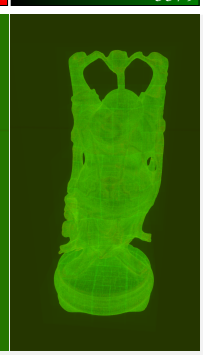
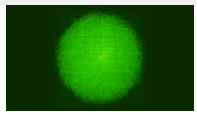
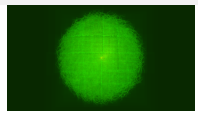
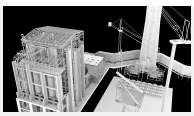
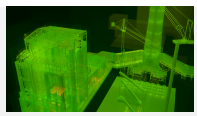
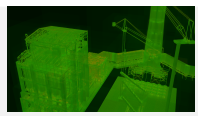
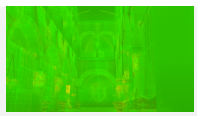
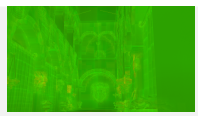
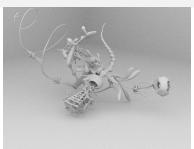
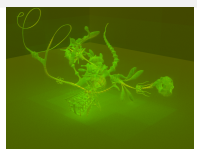
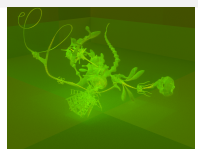
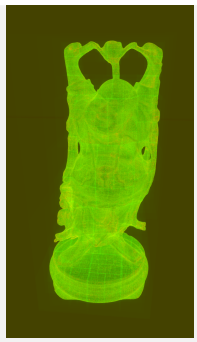
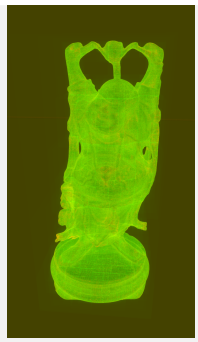
Render	Midpoint	Median	SAH
 *Bistro [88]*		 690 750	 15 813
 *Hairball [32, 97]*		 54 601	 24 385
 *Powerplant [93]*		 1 212 140	 14 028
 *San-Miguel [96]*		 899 951	 20 172
 *Sponza [92]*		 7952	 12 240
 *Yeah Right [90]*		 2372	 3379
 *Buddah [99]*		 1852	 2672
		Triangle Tests	*Intermediate Tests*

Table 4: The bounding volume hierarchy quality visualized with and without *spatial splits*. The red color channel represents the number of triangle tests in a pixel and the green color the intermediate tests when rendering a scene with ambient occlusion. Both are normalized against the maximum number of tests as found in the colorbars for each scene.

Render	Regular Splits	Spatial Splits
 *Bistro [88]*	 6784	 4106
 *Hairball [32, 97]*	 58 711	 3820
 *Powerplant [93]*	 3953	 3452
 *San-Miguel [96]*	 25 394	 2288
 *Sponza [92]*	 1229	 1731
 *Yeah Right [90]*	 443	 1703
 *Buddah [99]*	 460	 1706
	Triangle Tests	*Intermediate Tests*

4.3 Parallelization

In all cases, it is desirable to render images as fast as possible. Thankfully, ray-tracing methods are typically amenable to parallelization, as rays are often independent of one another, placing ray-tracing into a category of algorithms that are known as *trivially parallelizable*, where methods theoretically scale linearly with the available computational resources, potentially allowing images to be rendered arbitrarily fast. In practice however, scaling is not as simple. Each ray might be computationally independent, but they are not *data* independent: They all need access to the same scene. Consequently, ray-tracing is highly dependent on the underlying hardware, and software needs to be structured around it to achieve good performance.

Consider the case of rays traveling in roughly the same direction. These rays will probably intersect the same, or at least adjacent scene geometry, thus, if that geometry is stored in a cache that all of them can quickly access, the work of loading that data for one ray can be amortized across all of them.

However, if the rays are traveling in completely different directions, they may consequently access very different parts of the scene, making it harder to populate a cache with appropriate geometry. Even worse: time could be spent to populate the cache for one ray just to immediately remove it for the next.

Making matters worse: The above only applies to the ray-tracing process itself. Adjacent issues, such as the construction of the acceleration structures from section 4.2 are normally not so easily parallelized.

Evidently, creating fast algorithms for these problems is a difficult proposition, something that is still under active research to this day. Over the years however, a number of frameworks have emerged that make some of this work simpler. This section will present a few such approaches, starting with the simplest, OpenMP®, eventually making our way to *Application Programming Interfaces* (*APIs*) dedicated to accelerating ray-tracing with dedicated hardware.

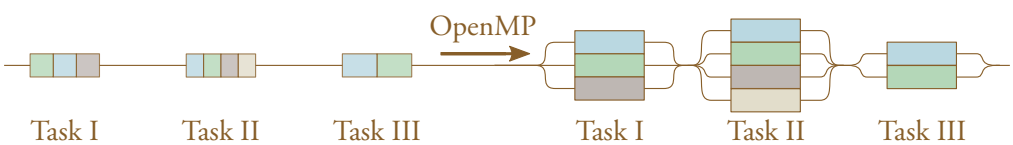


Figure 26: Graphical visualization of the typical `OpenMP` usage for splitting up tasks into parallel regions.

4.4 OpenMP

`OpenMP` is an API for performing numerous types of multi-processing tasks in a convenient and portable fashion in the `C`, `C++` and `Fortran` programming languages with the help of compiler directives and library routines and is jointly governed by all major compiler and hardware developers through a non-profit technological consortium known as the OpenMP Architecture Review Board.

As illustrated in figure 26, `OpenMP` provides a relatively simple way of successively parallelizing portions of a program, and the simplest application of it is typically through the use of the `parallel for` compiler directive, or `pragma`, as shown in figure 27.

```
#pragma omp parallel for num_threads(8)
for (size_t i = 0; i < height; ++i)
{
    for (size_t j = 0; j < width; ++j)
    {
        im[i][j] = ray_trace(i, j);
    }
}
```

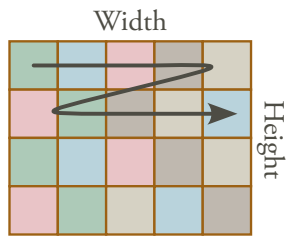


Figure 27: The top-level loops of a simple ray-tracer parallelized using the `parallel for` `OpenMP` compiler directive, splitting up the necessary work over 8 independent threads.

Modern versions of `OpenMP` even enable the creation of parallel tasks that can be submitted to a thread-pool, a process typically referred to as *tasking*. Further, tasks may even recursively create more tasks, as seen in figure 28, thus enabling complex algorithms to be parallelized in a simple fashion [4]. However, some care is still needed to ensure that each task is able to perform a suitable amount of work to account for the overhead of its creation.

Beyond this, `OpenMP` provides directives for automatically converting loop iterations to tasks with the `taskloop` directive, performing complex loop re-structuring, i.e., *vectorization*, and even offloading tasks to co-processors such as *Graphics Processing Units* (*GPUs*). While useful, these latter options require a bit more care to be applied correctly, such as making sure the data is properly aligned to the underlying memory primitives, or that the data is allocated in an a memory region co-processors can share.

4.5 Compute Shaders and OpenCL

As alluded to in the previous section, `OpenMP` *can* be used to parallelize algorithms to co-processors, such as GPUs, but it is not a common practice, and especially not for graphics. In that case, it is often desirable to have more direct control of the GPU through a lower-level

```
int fibonacci(int n)
{
    int fn1, fn2;
    if (n == 0 || n == 1)
        return n;
#pragma omp task shared(fn1)
    fn1 = fibonacci(n - 1);
#pragma omp task shared(fn2)
    fn2 = fibonacci(n - 2);
#pragma omp taskwait
    return fn1 + fn2;
}
```

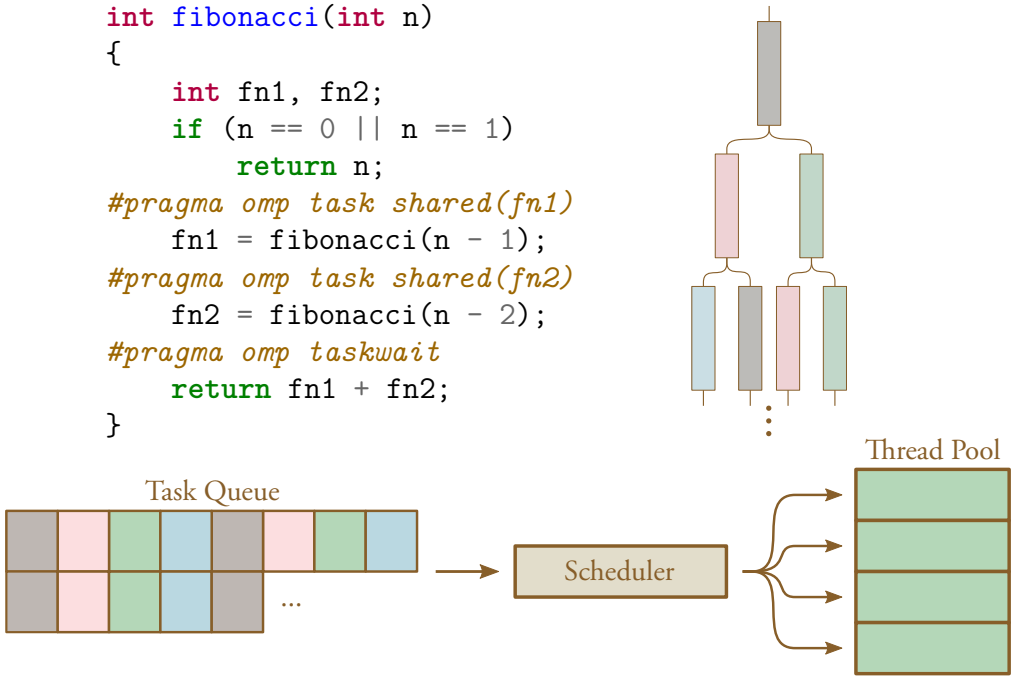


Figure 28: A more complex parallelization example to demonstrate the use of OpenMP® tasking. A simple program (**top left**) can use the `task pragma` to submit one or more tasks to a task-queue internal to the OpenMP® implementation (**bottom**). Each of these tasks can then in turn create more tasks as necessary (**top right**). Note that, while illustrative, this particular example would likely not benefit much from parallelization as the overhead of creating these tasks outweigh the cost of the work itself.

API to achieve higher performance. One particularly simple type of API that can achieve this is the `compute shaders` that are part of all modern graphics APIs [72, 82, 62], or the `OpenCL` API [80], all of which work in a similar fashion.

In the simplest case, data is mapped onto a one-, two-, or three-dimensional grid of *work-items* that a special program known as a *kernel* operate on, typically producing a new equally sized grid of items, as illustrated by figure 29.

This maps well to ray-tracing, as each *work-item* can be mapped to a pixel that a ray should pass through, and the final color for that pixel should be the output. In fact, this is almost equivalent to the loop from figure 27, but each iteration is run by the *kernel*. The ray-tracer used in Paper II is an example of this.

These APIs do not enforce this organization however. More complex approaches can e.g., operate on multiple items to produce a single one to implement so-called *reductions*, but these details are out of scope for this introductory chapter.

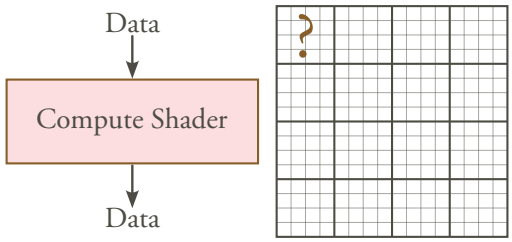


Figure 29: Simple GPU applications used by `OpenCL` and `compute shaders` generally operate by executing a small program known as a *kernel* on each *work-item* in a one-, two-, or three-dimensional grid.

4.6 Hardware Accelerated Ray-Tracing

Parallelization with `OpenMP` can provide a decent performance improvement, but it typically does not make it fast enough to use ray-tracing in real-time contexts, and even for high-performance applications.

Consequently, ray-tracers implemented with complex *kernels* with either `OpenCL` or `compute` shaders was the norm for a very long time. In many cases however, multiple of these high-performance implementations ended up with the same or similar algorithms and methods. The bounding volume hierarchy from section 4.2 is a good example of this.

After some time, Nvidia developed the `OptiX` ray-tracing API [76] to collect many of these common approaches in an organized way. Eventually, a subset of these were determined important enough to warrant the implementation of dedicated hardware to accelerate them. At the same time, these approaches were included as extensions to the DirectX® and Vulkan® APIs [82, 62] making these features more readily available to applications that already make use of them, such as many game engines [81, 64].

These extensions are not trivial to use however: They introduce several new stages and specialized compute shaders, as well as a completely new type of pipeline that must be correctly configured before any ray-tracing can be done. Once properly setup however, these extensions can enable truly high-performing ray-tracing applications. As such, all work related to Papers III and IV are made with these extensions.

This section will briefly present the new shaders and stages introduced by this *hardware accelerated ray-tracing pipeline*, many of which are illustrated in figure 30.

Shaders and Stages

The ray-tracing pipeline introduces five new shaders and one new fixed function stage, i.e., a pipeline stage primarily controlled by the hardware. All of them work in tandem to accelerate ray-tracing applications, and must consequently be properly configured.

Ray Generation In all cases, the *ray-generation*, or `RayGen` shader is always the first shader to be executed when starting up the ray-tracing pipeline. It is tasked with creating the rays that initiate the ray-tracing process.

Consequently, in the simplest case this shader is equivalent to the simple *kernel* program described by figure 27. This is not mandatory however. Rays can be created in any other way that the programmer chooses. E.g., a non-ray-tracing method may be used to find the closest hit point and rays can start from there in the direction of the light sources instead.

Acceleration Structure The performance of ray-tracing applications is heavily dependent on the acceleration structure used to contain the scene objects. However, for the hardware pipeline it was determined that more performance could be gained by letting the hardware control as much of this structure as possible. Consequently, these APIs hide most of the details surrounding this structure, even if it is well known that most hardware vendors [74, 65, 57] currently base it on the *bounding volume hierarchies* that was described in section 4.2.

This is all done to achieve the highest possible performance for ray-tracing applications. Thus, programmers work with a two-level abstraction illustrated in figure 31 to organize their scenes for ray-tracing:

At the bottom level, the appropriately named *Bottom Level Acceleration Structure*, or `BLAS` is found. Here, the programmer inserts the individual objects with an associated *transform* to situate them in the ray-tracing version of the scene. After that, multiple `BLASes` can be inserted into the similarly named *Top Level Acceleration Structure*, or `TLAS`, which is the only structure that the user can actually trace rays against.

This organization allows the programmer to arrange the scene around how the objects are intended to behave: If some set of objects are replicated multiple times over the scenes, the same `BLAS` can be referenced multiple times, effectively creating *instances*. Similarly, if an object is animated over the scene, it is possible to only change the *transform* associated with the `BLAS`, thus avoiding some of the costs necessary for updating the acceleration structure. Additionally, if a large portion of the geometry is completely static, all of those objects can be gathered in a single `BLAS`, and the API can be asked to optimize it for such a use.

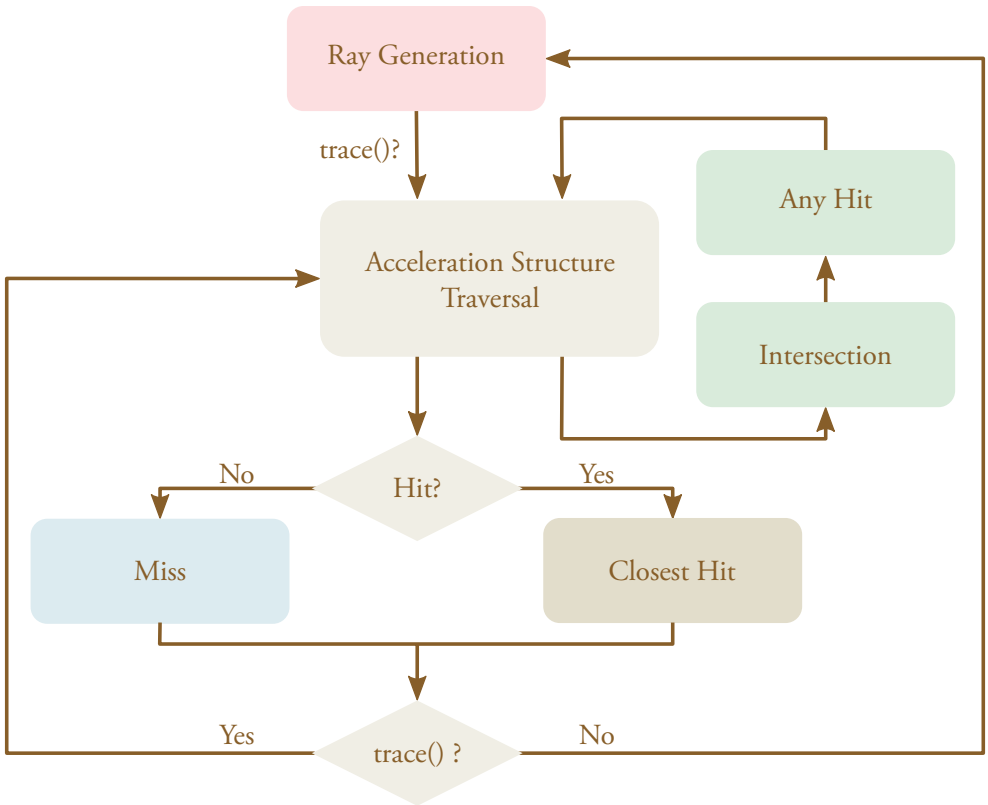


Figure 30: A simplified flow-chart over the hardware accelerated ray-tracing pipeline and its various stages. The programmer controls five of them: The *ray generation*, *intersection*, *closest hit*, *any hit*, and *miss* stages with the help of specialized programs. The last one, the acceleration structure traversal is instead primarily maintained by the hardware itself. Note in particular that this pipeline enables recursive invocations in some of these stages, as is illustrated by the looping arrows.

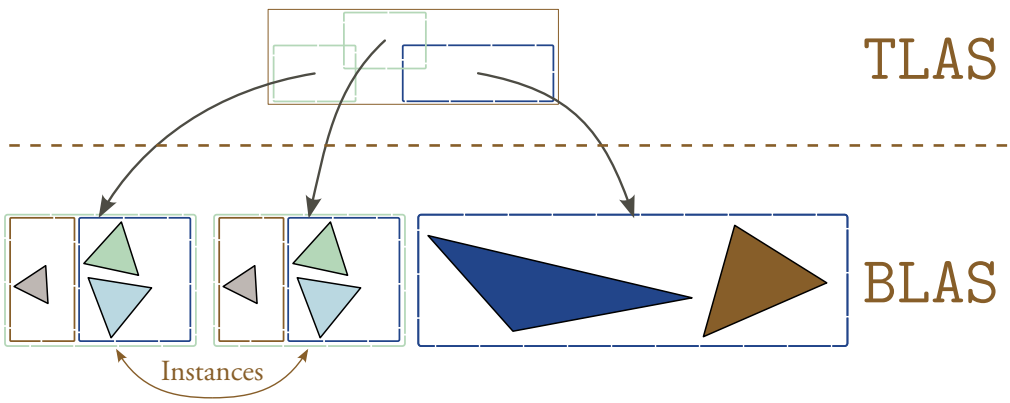


Figure 31: The hardware version of the acceleration structure consists of two levels: The *Top Level Accelerations Structure* (`TLAS`), and the *Bottom Level Acceleration Structure* (`BLAS`). This organization both enables high-performance ray-tracing and features such as cheap replication of objects that are used multiple times, i.e., *instances*.

Intersections The ray-tracing pipeline is arguably intended for scene geometry based around triangle meshes, as the ray-triangle intersection test is especially optimized and often implemented in hardware. However, it is possible to trace rays against any other kind of object; a task that is handled by so-called `Intersection` shaders.

In short, the acceleration structure can be configured to store *Axis-Aligned Bounding Boxes* (*AABBs*) around these objects, and the `Intersection` shader is called to determine if a ray that intersects with such a box intersects with the contained object, and if it does, return relevant information to identify where the intersection occurred, as is illustrated in figure 32.

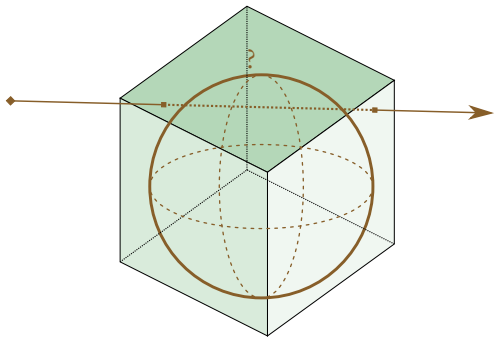


Figure 32: The acceleration structure for the *ray-tracing pipeline* only store the *Axis-Aligned Bounding Boxes* (*AABBs*) around user defined objects. The `Intersection` shader is used to determine if a ray that intersects such a box also intersects with the contained object, such as the sphere shown in this case.

Any-Hits, Closest-Hits, and Misses Since the acceleration structure traversal is controlled by the hardware, that leaves the remaining ray interactions with the users. I.e., the behavior when a ray hits objects or when it misses them. Both of these aspects are controlled by three different shaders: the `AnyHit`, the `ClosestHit`, and the `Miss` shader.

The arguably simplest of these is the `ClosestHit`. It is executed at the ray-intersection point of the object that was closest to the ray origin, as illustrated in figure 33a. This shader is typically used to implement surface shading of the objects using various material models, such as those as described in section 3.4.

In contrast to the `ClosestHit`, the `Miss` shader runs when the ray was not able to intersect with any geometry, which could e.g., return a fixed color to represent the background of the scene.

The last of these, the `AnyHit` shader, is arguably the most complicated one. It is called on objects along the ray that have been marked as *transparent*, and it is tasked with determining if the object is *actually* intersected or not, which can e.g. be used to implement *alpha-masking*, as described in section 3.5.

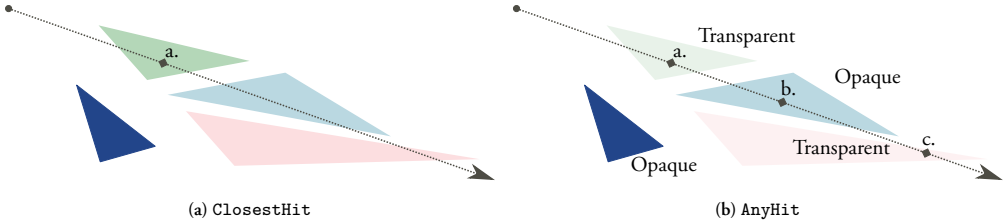


Figure 33: The ray-tracing pipeline contains two different types of hit shaders: the `ClosestHit` and the `AnyHit`. The `ClosestHit` is always executed on object closest to the ray origin (**a**), whereas the `AnyHit` shader runs on all *transparent* objects along the ray (**a**, **c**).

One interesting aspect of these shaders is that any of them are allowed to generate and trace *new* rays, i.e., they can be used to perform ray-recursions, as illustrated by the loops in figure 30. This can be used to implement many types of ray-tracing algorithms, such as *path-tracing*, where each new ray represents a step along its path.

Shader Binding Table One major change from earlier graphics hardware acceleration APIs such as `OpenGL` [72] and `OpenCL` [80] is that there is no longer an immediate mapping from the shader program or kernel to the data it should run on. E.g., as shown in figure 29, there is only a single active kernel that is run on each of the work-items.

The ray-tracing pipeline changes this quite significantly: A ray can intersect with any object in the scene and it is thus possible for multiple different shaders to be called for any ray going through the pipeline. Consequently the GPU need some way to find these shaders. This is done with the help of the so-called *Shader Binding Table*: Each time the user ask the hardware to trace rays, an index to the desired set of shaders is also provided. This can also be used to implement non-ray-tracing related features, such as using the *ray-generation* shaders to implement spatio-temporal filtering instead of generating rays, as was showcased in Poster E [49] to avoid costly GPU pipeline stalls [59].

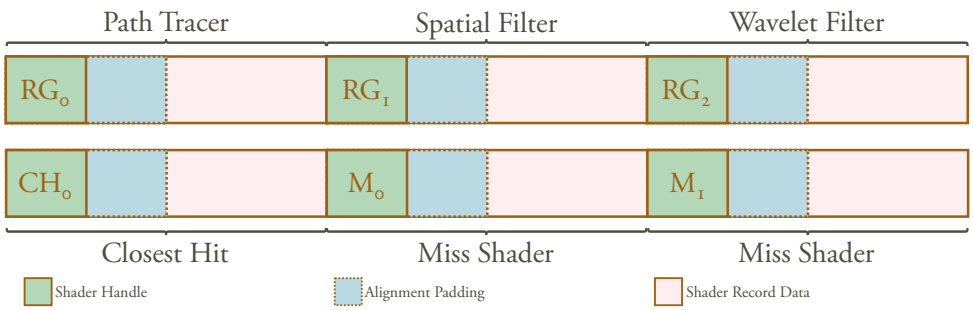


Figure 34: Visualization of the *shader binding table* (*SBT*) modified to perform non-ray-tracing related work, such as spatio-temporal filtering. Instead of switching between several different computer shader pipelines, a single ray-tracing pipeline can switch between the different shaders that are used to implement image filtering algorithms, such as a *spatial* and *wavelet* filter in this case.

4.7 Micromaps

Acceleration structures provide the largest performance improvement for ray-tracing applications, but they have some potential issues in some contexts. One such problem is the handling of fully- or partially transparent geometry, particularly those handled with *alpha* textures, as described in section 3.5.

This issue was particularly problematic for the hardware accelerated ray-tracing from section 4.6, as some effects need to invoke the relatively expensive `AnyHit`-shader, and may consequently interrupt the hardware traversal and drastically reduce its performance [78].

This was addressed with the *opacity micromap* structure, that gives the hardware a quick test for the common cases, allowing it to run uninterrupted for longer.

A tangential problem is the handling of extremely *detailed* and *dense* geometry that may unnecessarily inflate the size of the acceleration structure. To that end a similar structure, known as the *displacement micromap* allow users to selectively, and cheaply, offset small parts of the geometry in the same way as *displacement* textures from section 3.5.

Opacity Micromaps

In brief, an opacity micromap is simply a linear array of values mapped onto fixed sub-areas of a triangle specified by the space-filling curve shown in figure 35. Further, an opacity micromap can only contain a very specific set of opacity values depending on whether it is operating in the so-called `2-state` or `4-state` *mode*:

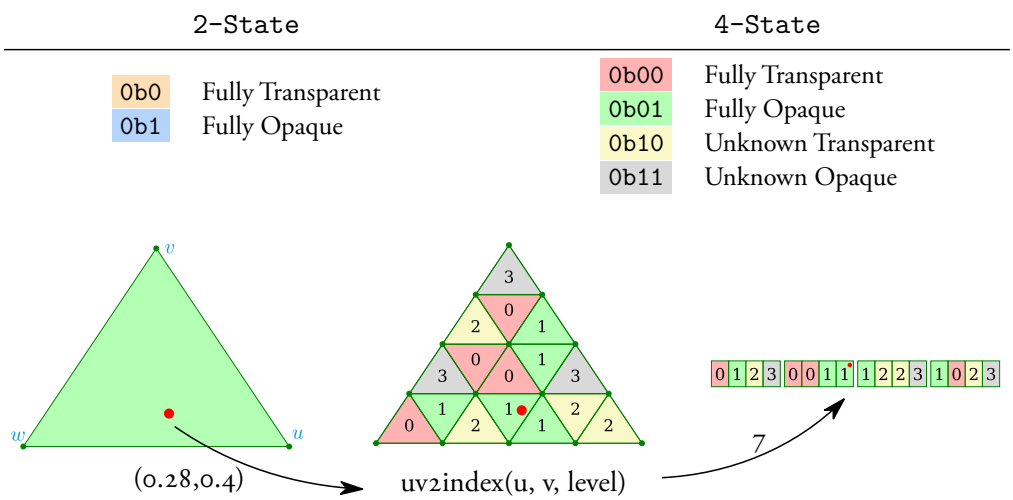


Figure 35: Description of the barycentric coordinate to opacity micromap indexing process.

As these values only occupy either 1 or 2 bits each, the opacity micromap is typically handled as a type of bit-vector. Functionally, the values are mostly self-explanatory:

- Fully transparent and opaque means that that particular sub-triangle is either completely opaque or transparent.
- Unknown values should look up the actual opacity values using some other method, by e.g., looking it up in an alpha texture. Additionally, these values can be converted to an equivalent `2-state` value, e.g., when it is undesirable to perform the alpha texture lookup, such as for shadow-rays in the ray-tracing pipeline.

Theoretically, this could be implemented in any triangle-based rendering pipeline as it only depends on the micromap data and the barycentric coordinates, but it is typically only used in the Vulkan® and DirectX® ray-tracing pipelines. There, the first two values correspond to rejecting or accepting an intersection, and the latter to deferring the decision to the `AnyHit`-shader.

Accordingly, both of these APIs provide the user with a set of functions to create and assign opacity micromaps to individual triangles to allow the acceleration structure to quickly prune large regions during the traversal, as is exemplified in Vulkan® with the `VK_EXT_opacity_micromap` extension [70].

Thus, by providing a relatively small amount of extra data describing when the `AnyHit`-shader calls can be avoided, we allow the hardware to run uninterrupted for longer, and consequently reduce the overall memory bandwidth, as less vertex data and textures need to be loaded.

However, these optimizations are entirely in the hands of the users: It is up to them to generate appropriate micromaps and apply them correctly to see any appreciable improvements.

Highest Useful Subdivision Level In `4-state` mode, the subdivision level is typically only another tool to dial in performance, effectively trading runtime at the expense of memory. In `2-state` mode, the triangle shape and subdivision level directly influence the final geometry as there are no *unknown* values, and consequently no way of calling an `AnyHit`-shader. This shape may be distinctive, especially compared to low resolution alpha texels, as seen in figures 45. Thus, if the intent is to conservatively recreate the alpha texture with micromaps, it is useful to estimate the maximum useful subdivision level n by computing when the texture coordinate length of a subtriangle along the longest edge e is less than one pixel, i.e., when the subtriangles are smaller than the pixels. In other words:

$$\frac{\max(|e_0|, |e_1|, |e_2|)}{2^n} \geq 1 \Rightarrow \log_2 \max(|e_0|, |e_1|, |e_2|) \leq n$$

This is valid regardless of any minification or magnification issues introduced by camera motion as long as the texture coordinates are static. However, this becomes more complicated for primitives with coordinate transforms, but if it is possible to estimate the maximum deformation, the same approach can still be used.

Special Indices In Vulkan®, opacity micromaps are applied on triangles by providing an array of so-called triangle indices at the creation of a bottom level acceleration structure (`BLAS`). However, it is relatively common for micromaps to contain only a single value, which may be wasteful, especially at high subdivision levels. To alleviate this, Vulkan® provides a set of special index values to map directly to these cases:

`-1`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_TRANSPARENT_EXT`
`-2`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_OPAQUE_EXT`
`-3`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_UNKNOWN_TRANSPARENT_EXT`
`-4`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_UNKNOWN_OPAQUE_EXT`

These values can be used in some cases to reduce bandwidth or otherwise simplify the micromap processing.

Displacement Micromaps

In contrast to opacity micromaps that are defined for micro-triangle *faces*, displacement micromaps operate on micro-triangle *vertices*. Despite this, both micromap types can be viewed as an implicit triangle-mesh on top of each triangle with distinct values associated with either the micro-triangles, or the micro-vertices respectively. For displacement micromaps it determines how much a particular micro-vertex should be displaced in space.

The number of micro-vertices are somewhat fewer compared to its micro-faces, and are consequently organized in a different way to avoid redundancies. Some requirements are also imposed on them to avoid gaps and discontinuities between triangles with different subdivision levels, both of which are described in more detail in the next sections.

Micro-triangles and Micro-vertices There are at least two different ways to organize the micro-vertices inside a displacement micromap:

U-Major Arranged bottom-to-top, starting from the edge between the w and v barycentric coordinates, as implemented in the Nvidia *Displacement Micromap Toolkit* [75].

Bird Arranged hierarchically in triplets along the so-called *Bird*-curve, as implemented in the Nvidia `VK_NV_displacement_micromap` Vulkan extension [71].

Each of these methods are summarized in figure 36 for subdivision levels 1 and 2.

Independence and Edge Handling Micromaps are always independent of each other, which is natural for micro-triangles, but for micro-vertices two limitations arise: First, all vertices along the edges of two triangles need to be duplicated, despite technically referring to the same locations. Second, the subdivision level of any neighboring micromap *should* only differ by one: Otherwise, *cracks* may appear in those regions, i.e., discontinuities between the displacement values of neighboring micro-vertices.

This is often handled by ensuring that all micromaps use a fixed subdivision level, or by finding and marking all edges between triangles of different levels, *descimating* them such that those edges fold unto the appropriate indices to avoid cracks, as is shown in figures 37 and 38.

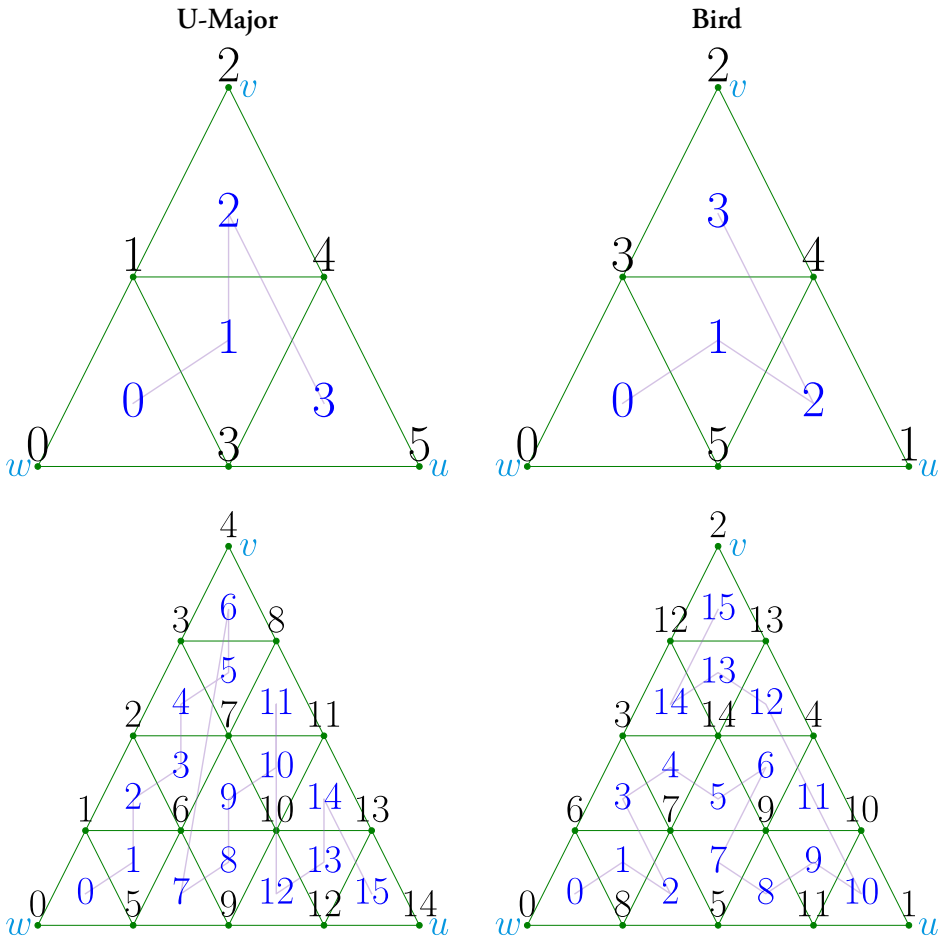


Figure 36: Visualization of the two currently used displacement micromap arrangements, showing both the micro-triangle and micro-vertex indices in blue and black respectively.

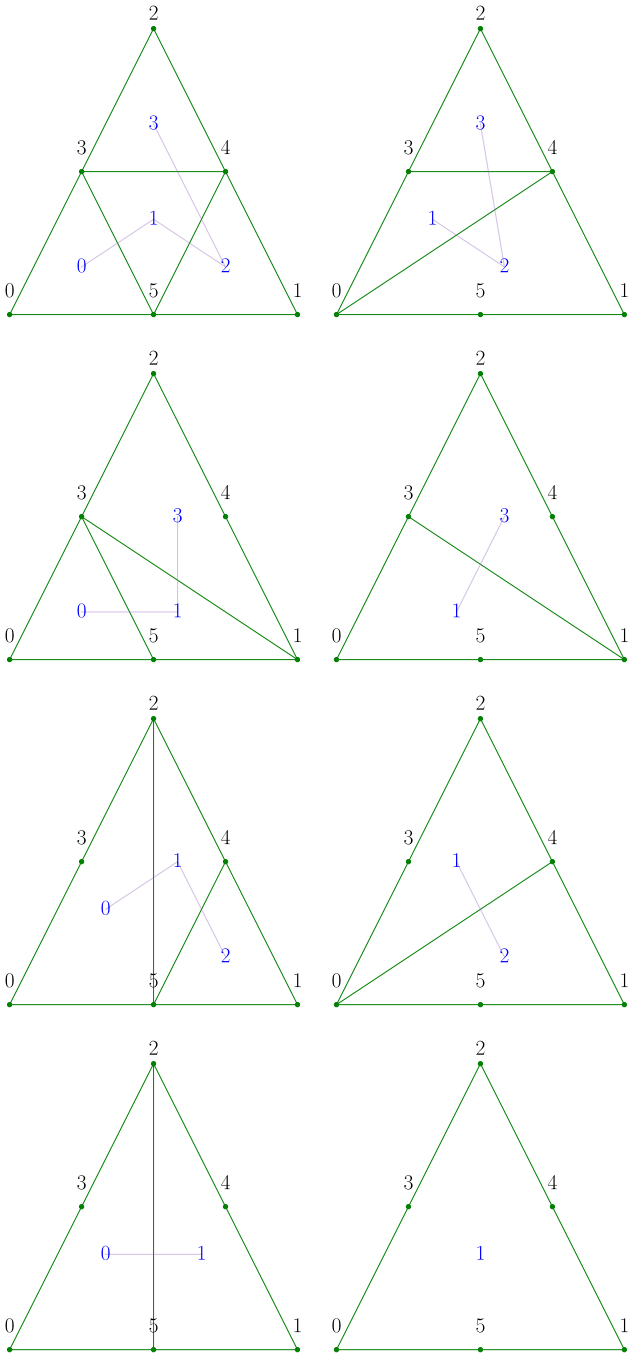


Figure 37: Exhaustive enumeration of the micromap edge decimations used for the first level of micromaps to avoid cracks between neighboring triangles. Observe that vertices are not removed from the mesh: The mapping from the micro-triangle to micro-vertex indices is merely adjusted for the affected edges.

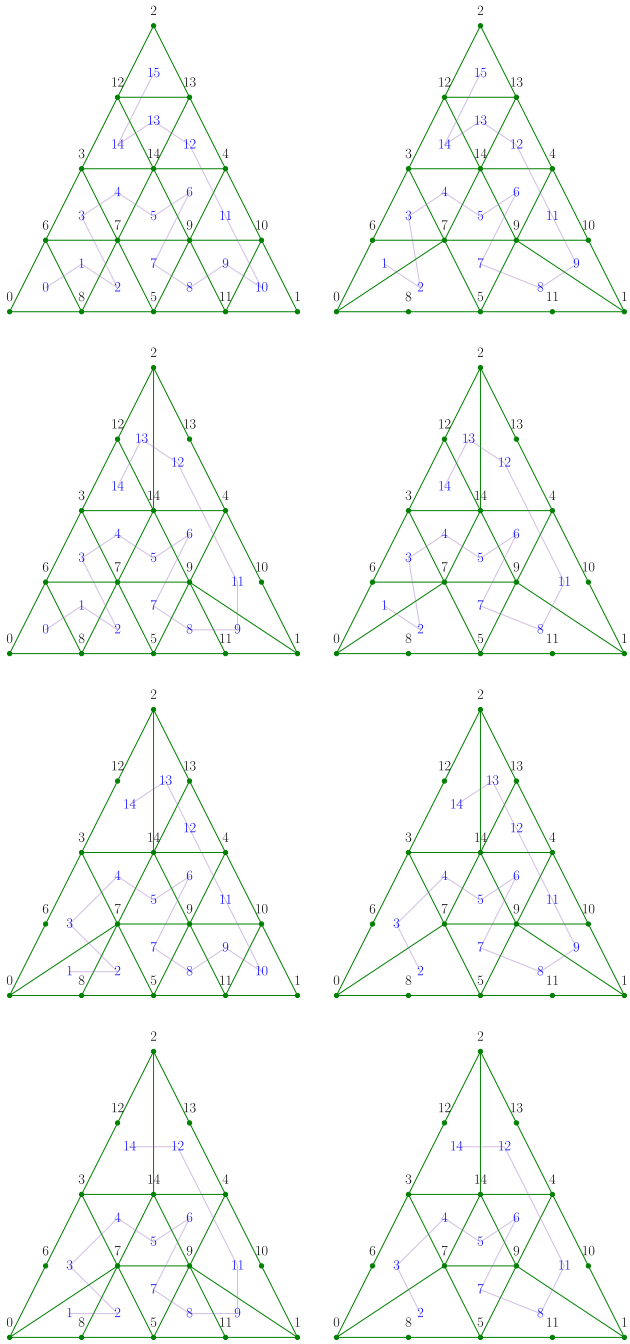


Figure 38: Exhaustive enumeration of the micromap edge decimations used for the second level of micromaps to avoid cracks between neighboring triangles. Observe that vertices are not removed from the mesh: The mapping from the micro-triangle to micro-vertex indices is merely adjusted for the affected edges.

5 Evaluation and Methodology

Compared to many other scientific fields, innovations in computer graphics are not always straight-forward to evaluate, as what makes one approach better than another is sometimes counter-intuitive. Graphics algorithms do not necessarily have to do the *correct* thing; if an approximation yields similar results in a faster way, it may end up being a superior choice.

Consequently, the methodology used for evaluating these algorithms are often a bit different compared to other fields, but in general they fall into one of two camps: Either they invent a new method for rendering something that has not been seen before, or they make existing methods better in some way, e.g., by being simpler to implement, or by performing better by using less memory, or by doing its tasks faster.

Novel approaches, such as the algorithms from Paper I can be evaluated in isolation as there is nothing to compare it to, but methods that primarily change performance, such as Papers II, III, and IV need to be appropriately motivated with metrics such as the ones explained in this section.

5.1 Image Comparisons

As alluded to earlier: Computer graphics algorithms do not necessarily have to be correct; as long as the new algorithms perform better in some other way, it can be considered an improvement. Consequently, methods for *comparing* images are required in order to determine how close images are to one another. Historically, this was often done subjectively: If the images looked about the same, it was often accepted. As time went on however, it became both increasingly time-consuming to manually verify the images and personal bias could skew the results.

Consequently, automated approaches for comparing images were gradually adopted, starting with simple statistical measures such as the *Mean Absolute Error* (*MAE*), *Mean Squared Error* (*MSE*), or derived ones such as the *Peak Signal to Noise Ratio* (*PSNR*), all of which are based on comparing the image pixels directly against a reference image, the formulae of which are summarized below for some test pixels I, with the largest maximum pixel value MAX_I and reference pixels R:

MAE	MSE	PSNR
$\frac{1}{n}\sum_i^n \lvert R_i - I_i \rvert$	$\frac{1}{n}\sum_i^n (R_i - I_i)^2$	$20 \cdot \log_{10}(\text{MAX}_I) - 10 \cdot \log_{10}(\text{MSE})$

These approaches work well when we want to estimate *exactly* how far the overall error is from some *true* value, but they do not help in *localizing* the source of the errors, nor are all errors even necessarily *perceptible* to us humans. For those reasons, another set of metrics

known simply as *perceptual* metrics were developed. The most well known of which is arguably the *structural similarity index measure* (*SSIM*) [55], but the more recent ꟻLIP metric [2] has proven more useful for computer graphics applications, particularly thanks to its error map that can help narrow down the source of the errors, as shown in figure 39.

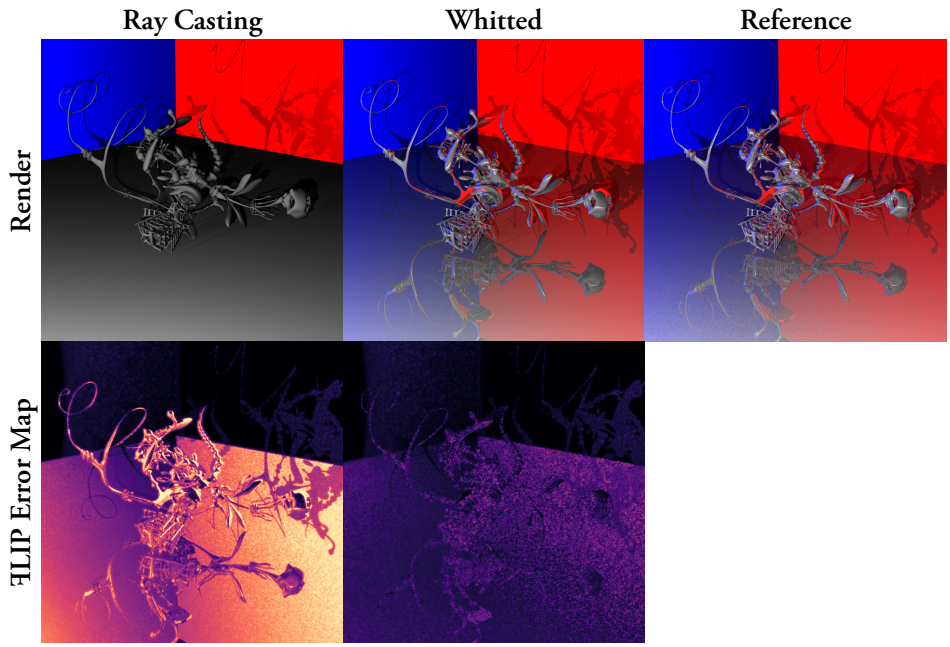


Figure 39: The ꟻLIP [2] metric applied to the *ray casting* and *Whitted* approximations to the rendering equation from figure 16.

5.2 Performance Measurements

In computer graphics, and particularly for real-time rendering, performance metrics generally fall into two categories:

Runtime — How long does it take for a task to complete, and,
Memory — How much data is required or used by this task.

Runtime In the simplest of cases, measuring the amount of time that a task takes to complete is relatively simple in most programming languages, and in this work all *CPU* based workloads uses these primitives to measure time, averaged over multiple runs to create a reliable value. As an example, this is used for the acceleration structure build times from Paper II.

In contrast, rendering algorithms running at real-time rates usually run on a *GPU*, and it needs to generate frames at a pace of around 30, and preferably 60 *frames per second* (*fps*), which translates to creating a new image within 33 or 17 ms respectively. Measuring algorithms running at these rates is usually not a problem, but GPUs typically interleave the rendering of multiple frames to hide resource latency to achieve these high frame rates. This can make it somewhat difficult to provide a reliable *frame-time* measurement, but modern GPU APIs have dedicated support for creating relatively precise timestamps at the beginning- and end of the frame generation process [82, 80, 62], which are used liberally in Papers II, III and IV to estimate the rendering times.

Memory In this work, whenever the memory use for various resources are estimated, it normally refers to the total number of uncompressed bytes required to store that resource, typically in the order of hundreds of MBs to tens of GBs.

These figures are usually estimated manually, e.g., by counting the number of items that are being stored and their through their individual size in bytes, but in a few cases, these values are controlled by the graphics APIs themselves, such as for the hardware micromaps from Paper III. In those cases, the sizes reported by the APIs are used directly instead.

Note that additional data, such as the supporting structures that are used to hold the data itself are not included in any of these figures.

Contributions

The only way to define your limits is by going beyond them.
- Arthur C. Clarke

This chapter will present some of the problems that motivated the projects behind each of the included papers, as well as the creation of two software tools that supported these projects, which are arguably considerable contributions themselves, as is described further in sections 6 and 7. After that, the individual scientific contributions from the papers will be presented in section 8, before delivering a set of concluding statements for each of these projects with some deliberation on their future prospects in section 9.

Motivations

All projects start somewhere, and this section will introduce the issues that incentivized the projects behind each of the included papers.

Paper I: Photon Mapping Superluminal Particles

Generally speaking, light sources in computer graphics are relatively simple things: They are typically either a point in space that emits light, or they are a regular surface object designated to emit light.

In reality however, generating light can be a quite complex process, involving various chemical and physical reactions. Often, these can be ignored and energy can be assumed to radiate from the source with a certain spectral intensity, or in the case of computer graphics: A color. This is not always the case though. Some phenomena need to be simulated

more closely for the emitted light to look *correct*. One such phenomenon is Cherenkov Radiation; light emitted from charged particles traveling at *superluminal* speeds. While well understood by the physics community [8, 13], little to no effort had been done to visualize the phenomenon in a more graphically correct setting. The work in Paper I was done to rectify this.

Paper II: Parallel Axis Split Tasks for BVH Construction with OpenMP

Rendering algorithms frequently work with enormous amounts of data and often run for very long stretches at a time. Consequently, it is important for these algorithms to be as fast as possible. However, many of these algorithms are incredibly tricky to implement in ways that are both correct and performant. As a result, there is a strong incentive to simplify the development of these algorithms in some way.

This spurred the research into using `OpenMP` to simplify some of these algorithms: It is already routinely used to parallelize some aspects of ray-tracing. Perhaps it can be used to simplify the construction of other and more complicated ray-tracing features?

Computer Graphics Scenes

Developing rendering algorithms is often an incredibly complicated and time-consuming process, and evaluating it against earlier approaches is often even harder: The only reliable way is to fully re-implement them and test them on the same set of scenes. Unfortunately, it is rare for earlier implementations to be available, and the scenes they are showcased in are often proprietary or inaccessible.

Thus, whenever scenes are made publicly available, they often end up being re-used as reference points for a very long time. *The Stanford 3D Scanning Repository* [99] and the Sponza scenes [91, 92, 98] are good examples of this.

Not all rendering algorithms are applicable to all scenes however: It is not useful to test an algorithm that changes how *alpha masking* works in a scene without alpha textures. Thus, it arguably benefits the entire field to have more scenes to work with.

Consequently, when the need for more varied scenes with these alpha textures arose, alternatives were soon found among those distributed with the *Physically Based Ray Tracer* (*PBRT*), a so-far largely untapped source of computer graphics scenes.

Consequently, this encouraged the creation of an extra project to develop tooling for accessing and working with these scenes, and by extension, give the computer graphics community easier access to these scenes, and even simplify the rendering process for artists who want to use *PBRT* as their primary ray-tracer.

Extending the Computer Graphics Tooling

Many computer graphics algorithms often utilize methods to pre-process some of the data that will be used in a rendering algorithm, a process that is generally referred to as *baking*. E.g., the textures from section 3.5 are often the result of such a process, and it is typically done to simplify the rendering algorithm itself, thus letting it achieve higher-performance.

However, this *baked* data needs to be stored somewhere such that the rendering algorithms can actually find it again and associate it with the correct object. This is a common pain-point for developers, as they need to maintain this customized data somehow.

Theoretically, the previously described `PBRT` format is flexible enough to contain it, but it primarily supports *static* scene geometry, so for real-time scenes with animations, developers often turn to a different scene format, such as the *Graphics Library Transmission Format*, or `glTF`. As it turns out, this format is similarly flexible: It naturally supports extensions to add new features or new types of data. However, so far it has been a very manual process to add these to `glTF` files.

This spurred the creation of a set of plugins to Blender [58] that greatly facilitated this process, and the presentation of these plugins [85] provided a basis for developers to create similar plugins for their particular use-cases, as is discussed further in section 7. Eventually this created a convenient mechanism for researching various new types of data, such as *micromaps*.

Paper III: Succinct Opacity Micromaps

As presented by Gruen, Benthin and Woop [19], the concept behind opacity micromaps is relatively simple: Provide a quick test to determine if a subset of a triangle is transparent or not. Effectively substituting a very small amount of memory for an up to 40 % faster ray-tracing run-time. While arguably only a modest improvement, the low cost warranted extending the official hardware ray-tracing APIs with a similar feature [82, 70, 62].

At the time, this feature was notably under-specified, and its performance characteristics was not well-known. Further: While opacity micromaps ordinarily only requires a small amount of memory, they also scale exponentially and may need compression to be practical in some cases. The work in Paper III addresses all of these issues.

Paper IV: Color and Attribute Micromaps

The primary goal of hardware opacity micromaps is to improve ray-tracing performance by letting the acceleration structure traversal run uninterrupted in hardware for longer, i.e., reduce the number of times the user-defined `AnyHit` shader needs to be called to make an intersection decision.

Put another way: The opacity micromap data-format is used to make the ray-tracing APIs execute in hardware for longer. Could something similar be done to let other aspects of ray-tracing do the same?

There are quite a few ray-tracing approaches that only need to do simple computations, and does not necessarily benefit from the full complexity of the `AnyHit` shader. One such example is partial transparency: If the object opacity can be provided somehow, the hardware could feasibly be tasked to perform the blending operation directly and stay in the accelerated traversal loop the entire time.

In fact, the hardware micromap APIs are already prepared for future extensions. Consequently, Paper IV investigates the feasibility of generalizing micromaps to contain arbitrary attributes, some of which could be used to make the ray-tracing hardware more autonomous.

6 PBRT Scene Importing for Blender

One particular issue in the field of computer graphics is finding appropriate scenes and assets to test novel rendering algorithms on such that they can be meaningfully compared with existing approaches.

Historically, this has been addressed by using a de facto set of scenes, such as those provided by the *Computer Graphics Archive* by McGuire [97]. However, with advances in graphics hardware and rendering in general, many of these scenes are becoming quite dated. Some effort to address this is being made by e.g., the *Intel Sample Library* by Meinl et al. [98], but that is still only a handful of new scenes.

There is however a large repository of scenes and assets readily available elsewhere: Those distributed together with the *Physically Based Rendering* books [40] and the *Physically Based Ray-Tracer*, or *PBRT* for short, a framework designed for teaching the concepts and theory behind photo-realistic rendering while still being a sophisticated renderer in-of-itself. Unfortunately, these assets are distributed in the equivalently named `PBRT` scene format, making them a bit hard to use elsewhere. Although, the format is simple enough that a number of 3D design tools have created *exporters* **to** this format in order to leverage its rendering capabilities. A few examples of which are listed below:

- Houdini [77]
- Cinema 4D [67]
- Autodesk Maya [63]
- RenderToolBox (Matlab) [60]
- Blender `v3` [66], and `v4` [79, 73]

To date, only a limited amount of work has been done to create *importers* to allow scenes in this format to be used elsewhere, such as the work by Ingo Wald [83], and no such work has been done for importing these scenes to the 3D modeling and rendering framework Blender [58].

There is a simple reasons for this: Scene geometry can often be easily extracted, but the *materials* cannot. In contrast to the material model described in section 3.4, `PBRT` can construct arbitrarily complex materials by mixing various models together such that reconstructing them in other rendering frameworks can be quite challenging.

Consequently, one side effect of this thesis is the development of a tool that can import scenes from *all* current versions of `PBRT` (2, 3, and 4) [86], and convert most, if not all objects and materials to equivalent or similar one for the `Cycles` [61] rendering engine in the form of *material graphs*. An example of this can be seen in figure 40, taken from the Landscape scene [94].

Generally speaking, a PBRT scene contains 4 things: Light sources, a camera, rendering settings, and geometry, the last of which is typically divided into normal objects, and *instances*, i.e., pointers to some other geometry, such that they can more or less be re-used elsewhere in the scene relatively cheaply. The tool imports and organizes all of these in Blender in a structured fashion, as can be seen in figure 41.

As a result, this repository of assets are now more readily accessible to both artists and graphics researchers, even if quite a bit of work remains before the Blender *material graphs* are properly aligned with the ones in *PBRT*, and the respective renders are directly comparable to each other, but a substantial step forward has been made, as can be seen in table 5.

High-performance and real-time algorithms often avoid *material graphs* however, as they can have a substantial performance impact. In those cases, Blender can instead convert them to more appropriate formats, such as glTF [68], as discussed in section 7.

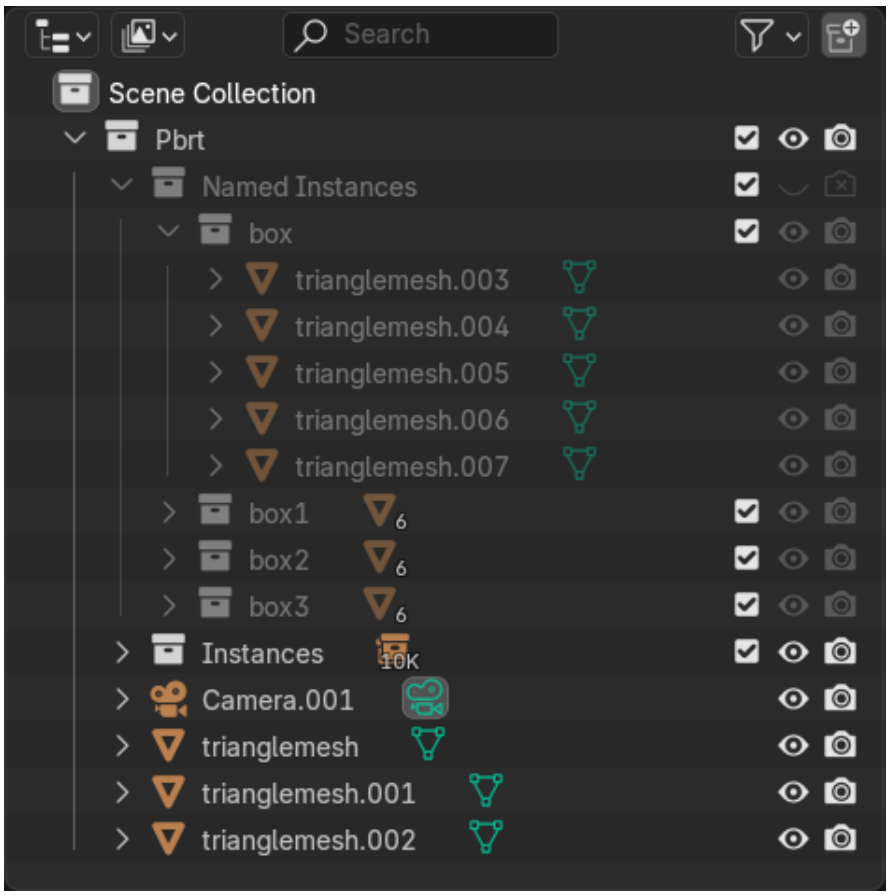


Figure 40: The layout of a PBRT scene that has been imported into Blender with this tool, allowing easy access to edit any of the objects within the scene. This example is taken from the "ArcSphere" scene in table 5.

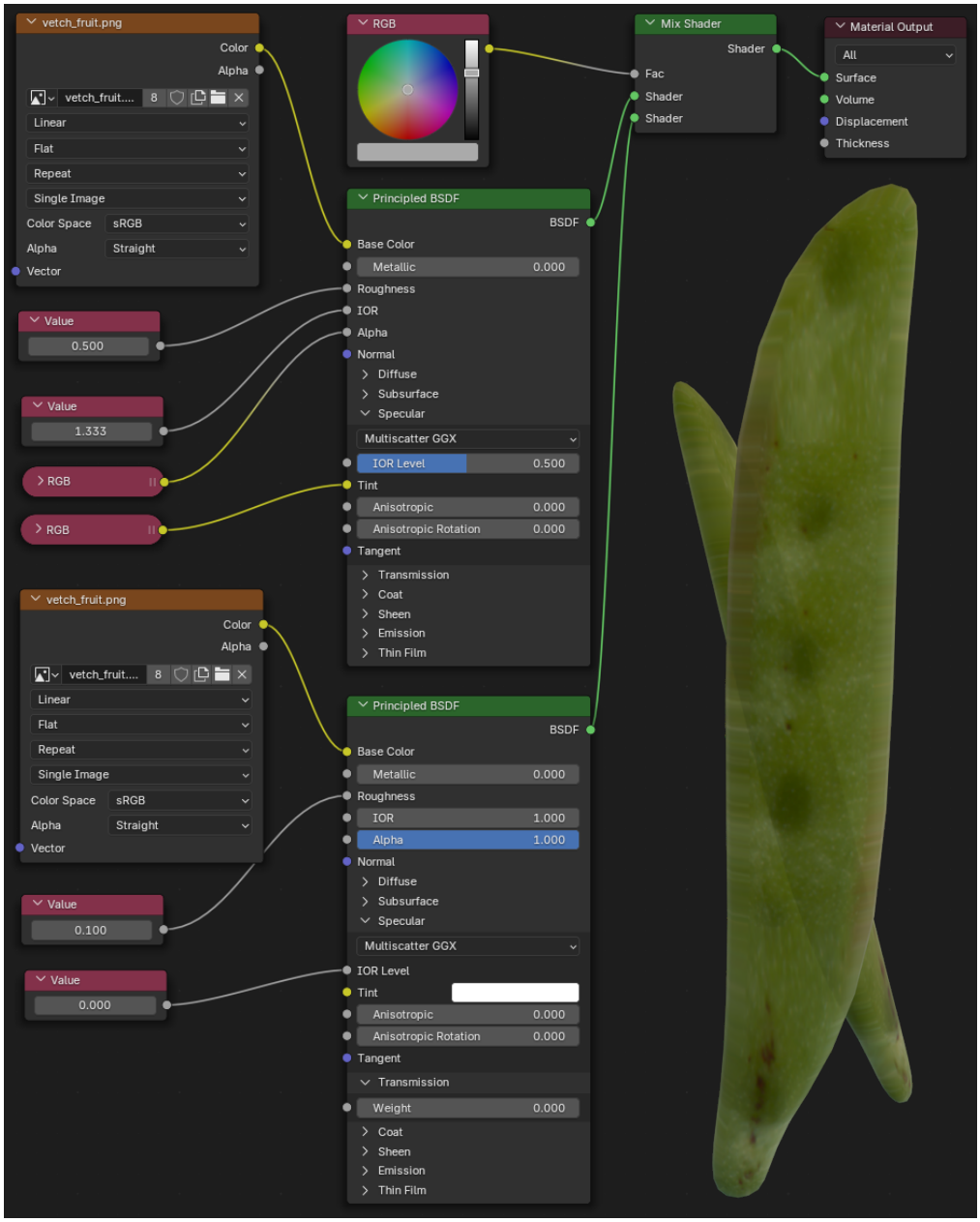
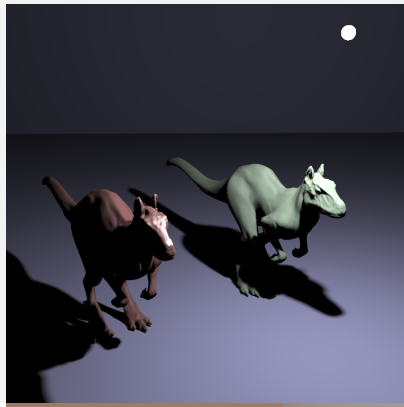
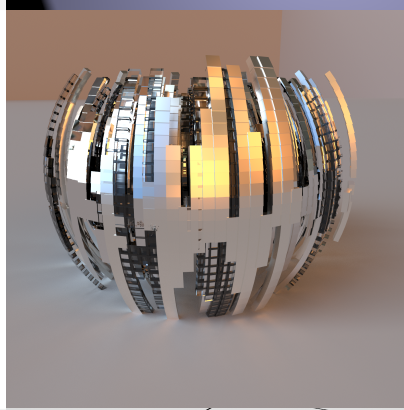
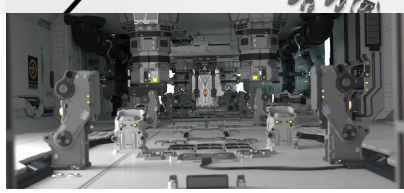


Figure 41: An example of a *material graph* from the Landscape scene [94] with an accompanying rendering of this material in a simplified environment.

Table 5: A selection of scenes as rendered by PBRT [40] and by Cycles [61], the primary ray-tracer in Blender after being imported by this tool.

Scene	PBRT	Cycles
Killeroo		
ArcSphere		
LTE Orb		
Zero Day		

Scene	PBRT	Cycles
San-Miguel		
Landscape		
Transparent Machines		
Watercolor		
Kroken		

7 Extending glTF Scenes from Blender

The PBRT format is a pretty general format for describing materials, but the same expressiveness may end up becoming a performance bottleneck in many situations. Furthermore, the format is intentionally designed for static scenes, with only limited support for animations to create motion-blur. Consequently, something else is needed if performance is a concern, or if the geometry is intended to include non-trivial animations.

One format that Blender [58] can convert scenes into that could be used to address this is the *Graphics Library Transmission Format*, typically abbreviated as glTF [68]. In short: It is a royalty free format for transmitting and loading 3D scenes and models, while still keeping the data itself as small as possible, both in-memory and when stored on disk. It is also very extensible, something that can make it suitable for a number of use-cases it was not originally intended for, as is explored more thoroughly in the next section.

The glTF format uses a number of different entities to accomplish different tasks: At the higher levels, there are meshes, cameras, and other objects that constitute the scene itself, while the lower levels describe where the individual bytes are found.

The current glTF 2.0 standard defines the following high-level entities:

Scene	A container for a scene similar to one described in section 3.
Node	A container for a spatial coordinate and orientation, and optionally a link to some geometry. It may also contain links to other nodes, thus forming a hierarchy known as a *scene-graph*.
Mesh	Container for the scene geometry, typically as described in section 3.1. Can be found inside a *Node*.
Camera	Container for the cameras such as those described in section 3.2. Can be found inside a *Node*.
Material	Container for material descriptions, as is described in section 3.4. Used by the *Mesh* to find material and texture properties.
Skin	Container for mesh deformations used in so-called skeletal animations.
Animation	Container for animations across all scenes.

Found at the lower level, the next set of entities define how the individual bytes are found:

Buffer	The lowest level of data storage: An unstructured array of bytes.
BufferView	Metadata describing how to extract the individual bytes from a *Buffer*.
Accessor	Descriptor over the amount and type of data a *BufferView* refers to.
Texture	Descriptor with pointers on how to find *Images* and *Samplers*.
Image	Pointer to a *BufferView* on where to find the bytes making up the image or a `URI` link to it.
Sampler	Describes how pixels in an *Image* are intended to be extracted.

An overview of these entities and the relationship between them can be seen in the diagram in figure 42.

This format is sufficient for describing a large variety of scenes and is the primary format used for the research carried out in Papers II, III and IV with the help of the extensions described in the next section.

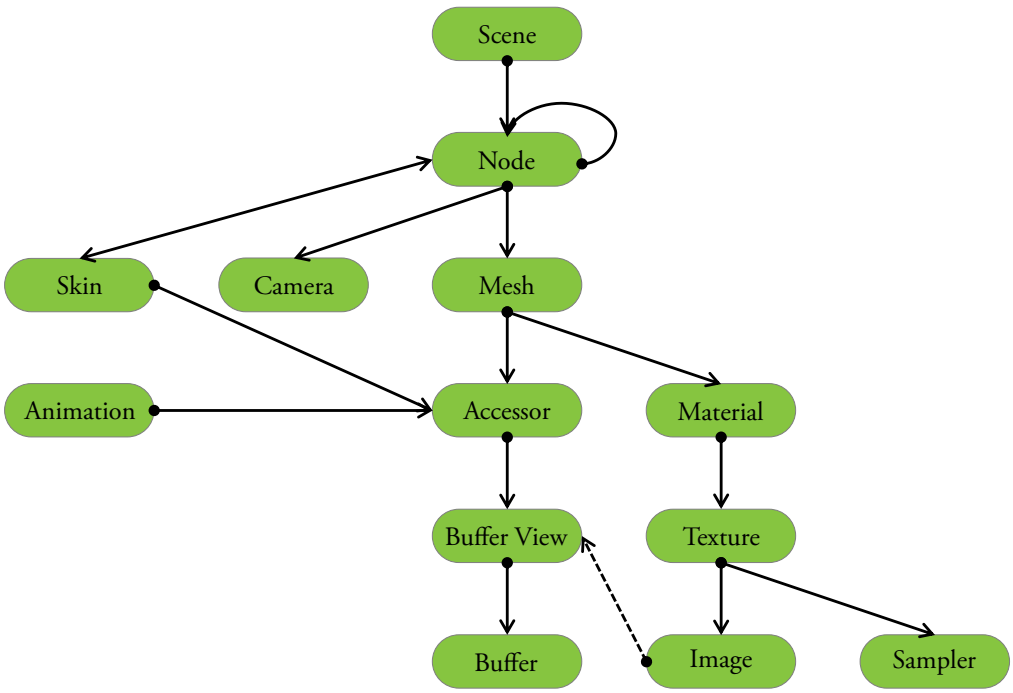


Figure 42: An overview of the entities defined by the `glTF` format and the relationship between them. A scene uses a set of *nodes* to organize *cameras* and geometry such as *meshes* and *skins* in 3D space, possibly with *animations*. *Materials* are used to give meshes their surface characteristics, perhaps with *textures* applied on them; i.e., *images* wrapped over the surface with pixel data extracted according to a *sampler*. The *accessors*, *buffer views* and *buffers* jointly cooperate to describe how the individual bytes for the other entities should be extracted.

Extending glTF

The glTF format is able to describe a large variety of scenes, but in computer graphics it is quite common to use specialized data for various rendering algorithms, and even more so in the research leading up to the development of these methods. Consequently, the format was designed to be extensible to allow developers to add almost arbitrary amounts of customized data into it.

In Blender [58, 69], this extensibility is exposed to developers through a series of hooks: Specialized functions that run and modify the glTF entities as they are created during the conversion and export process of a Blender scene, such as those imported from the PBRT files described in section 6.

These functions can be used to perform arbitrary modifications to the exported glTF files, enabling a large number of use-cases. One such example is to create a hook that watermarks all textures belonging to geometry that is still under development, as illustrated in figure 43.

Another possibility is to derive additional data from the textures, such as *micromaps*, which are described in more detail in section 4.7, and Papers III and IV. However, in contrast to the earlier watermarking process that only modifies the textures, extra data derived in such a way needs to *stored* somewhere. This is typically done by creating new *buffers* and *views* for that data, and then adding pointers to these in the entities for later retrieval during rendering. One such process that creates *opacity-micromaps* from alpha textures can be seen in figure 44, and the results of rendering that data is shown in figure 45.

Note that this kind of processing relies on details from both the glTF specification [68] and Blender internals [69] that are out of scope for this background summary. A complete overview on the development of these details can be found in the presentation and reference material related to this subject [85].



Figure 43: One example of a plugin that can be added to the Blender glTF exporter interface is a watermarking tool that can mark all textures belonging to assets that are still under development.

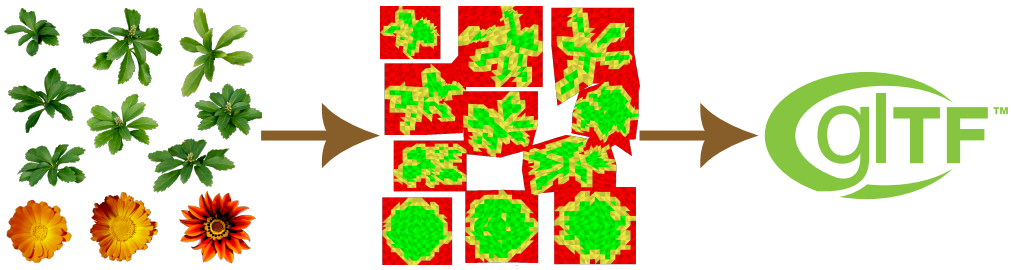


Figure 44: The extensibility of the `glTF` format makes it relatively easy to add new things to it, such as *opacity micromaps*. An object that similar to *alpha-masking* can selectively mask out objects, but at a much lower cost, something that is explored more thoroughly in Paper III. In this example, *opacity micromaps* (**center**) are derived from alpha textures from the Sponza [92] scene (**left**) and then stored in a `glTF` file, here symbolized with its logo (**right**). The rendered results from this process can be seen in figure 45.

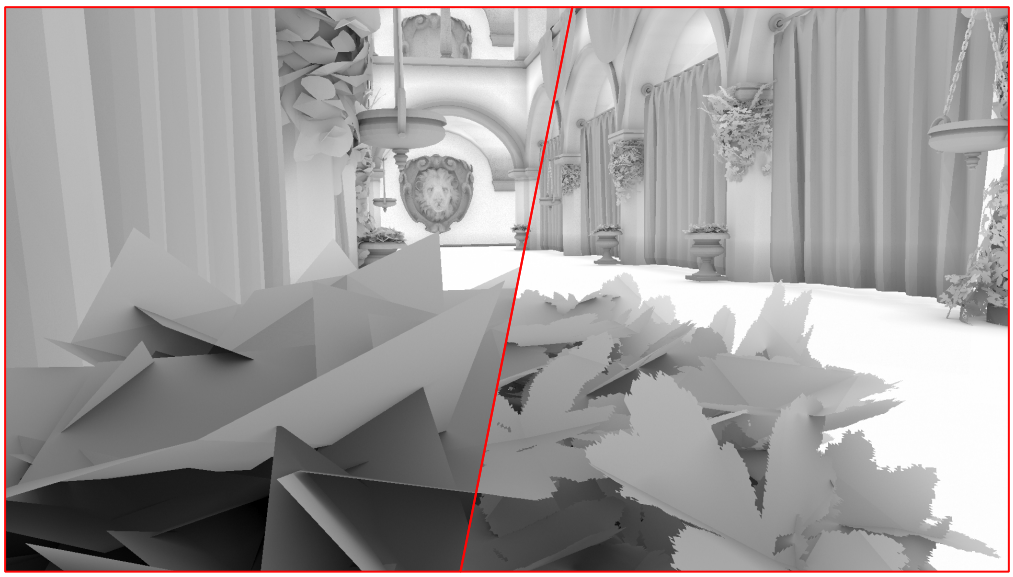


Figure 45: *Opacity micromaps* are similar to *alpha-masks* insofar as they make geometry more granular, but they do this at a much lower cost, albeit in a more limited way. The left side of this image shows a rendering of the Sponza [92] scene base geometry, and the right side shows the same scene with *opacity micromaps* applied to it.

8 Scientific Contributions

This section briefly summarizes the scientific contributions of the included articles.

Paper I: Photon Mapping Superluminal Particles

Cherenkov radiation is a thoroughly studied and well understood lighting phenomenon from particle physics [8, 13]. It has however, *not* been studied in the context of computer graphics, and the matter of rendering the phenomenon on digital images in a graphically correct scene have been largely neglected. This paper demonstrates a novel technique to capture it, including theory for performing statistical improvements with Russian roulette by estimating the emitted photon distributions. Furthermore, it demonstrates that ray-tracing techniques can be leveraged to assist the simulation process itself: Charged particles *can* be simulated as rays, even if the underlying *straight-line* assumption might be too simplistic for real use-cases.

Additionally, second order effects, such as caustics are trivially created with the presented algorithms, which to our knowledge have not really been considered in real scenarios, even when they can have a substantial impact, particularly for specular surface interactions.

Paper II: Parallel Axis Split Tasks for BVH Construction with OpenMP

Historically, acceleration structures have been extremely important for improving the run-time performance of ray-tracing applications. However, it has not been as important to reduce the build-time for the acceleration structures itself; primarily since it is usually only done once, but also due to the inherent complexity, particularly when the build is parallelized over multiple threads or co-processors.

In contrast, this paper demonstrated the feasibility of using OpenMP® to parallelize the implementation of one particularly important acceleration structure in a new way, achieving a modest build-time improvement while still maintaining the simplicity of the original serial implementation.

Paper III: Succinct Opacity Micromaps

Even before this paper, the opacity micromap concept had proven itself to be a simple but useful method for accelerating ray-tracing in software [19], and this publication cemented this claim, particularly for hardware implementations.

The paper additionally introduced a new algorithm for mapping barycentric coordinates to micromap sub-triangle indices, which is arguably easier to understand than the reference algorithm supplied by the ray-tracing APIs yet proven to be equivalent.

Finally, the paper used this algorithm to implement a new compression algorithm for opacity micromaps that can compress them by up to 110 times, or equivalently, to less than 1 % of their original size.

Paper IV: Color and Attribute Micromaps

Opacity micromaps primarily boost ray-tracing performance by helping the hardware avoid interruptions in the acceleration structure traversal. Similarly, it could be beneficial to have the hardware perform even more operations to avoid other kinds of interruptions as well.

If the hardware ray-tracing pipeline is to perform more work however, it first needs data to work with. Thankfully, the micromap format is already prepared for such a scenario. In fact, the displacement micromaps concept is such an attempt at using the format for other purposes, even if that particular format was arguably not able to justify the extra complexity it added to the acceleration structure.

In contrast, this paper showed that it may be more feasible to store surface attributes in the micromap format instead: They do not affect the acceleration structure, and can even be used in place of textures, and they are even particularly useful for meshes with densely packed vertices where they can substantially reduce the overall memory use, with only a small loss of image quality.

The paper also presented extensions to the micromap concept to allow filtering of multiple adjacent micromap values, and demonstrated how these concepts could be used to enable the hardware pipeline to execute a limited set of operations without an `AnyHit` shader, such as the color-blending operations needed to implement partial transparency.

9 Conclusions and Looking Forward

Computer graphics is not a solved problem by any stretch of the imagination, and thanks to the ever evolving demand and need to process increasingly large quantities of data, it is likely that research in this space will continue for many years to come. This thesis in particular covers a relatively broad swathe of this field, but is primarily focused on various aspects of ray-tracing:

Paper I clearly demonstrated that ray-tracing approaches can be used to simulate and render relatively exotic phenomena from physics. In fact, it is arguably *underused* in practice; many of the ray-tracing techniques used by computer graphics are heavily optimized to work with billions of rays, something that probably could be beneficial for many physics simulations. Strictly speaking though, the techniques from Paper I may not be physically *correct*, but if nothing else, they make a strong case for the communication and understanding of these phenomena. These techniques are also arguably a stepping stone: The presented renders only represent a single frozen frame of superluminal effects, in reality, they interact much more with objects, and could benefit from a more volumetrically correct simulation. The phenomenon is also incredibly *transient*; they only occur for a split-second before it is gone. Consequently, it would be very interesting to add this phenomenon to a so-called *transient rendering framework* [25], that creates an animation of how light propagates in the scene over time.

On the other hand, research into ray-tracing performance optimizations, such as the work on acceleration structures in Paper II currently occupy a somewhat odd position:

- It is desirable to use the established hardware to render images as fast as possible, but,
- The same APIs also abstract away the details of the acceleration structures.

This makes it hard to use new methods to improve the structures, but also prevents investigations into what approaches are currently being used and how to improve them. Consequently, this lets the hardware vendors be the primary driver for innovations in this area. However, ray-tracing is not the only use case for acceleration structures; equivalent structures are frequently used for other applications such as collision detection [35] and volumetric data storage [39]. Thus, methods that simplify the construction of these structures while maintaining a fast build-time is still of interest.

Researching such structures properly requires access to a wide variety of scenes however. The already available `PBRT` scenes can be very useful in that regard, as a large number high-quality assets have been made available in that format over the years. So far though, they have only really been usable by the *Physically Based Ray-Tracer* framework itself. Thus, using those assets for research elsewhere often required a substantial time-investment. Consequently, the `PBRT` scene importer that was developed over the course of this thesis can make it a

lot easier for researchers to load existing scenes into Blender, modifying them as necessary, and then exporting them into any of the formats that Blender already supports. Further, this also demonstrates that while the `PBRT` format might not have been *intended* as a scene interchange format, it is arguably flexible enough to be used in that way. It has limitations though; most notable is the single camera-per-scene restriction, but it also lacks native support of most modern animation types, making it unsuitable for many real-time rendering use-cases.

On the other hand, the `glTF` scene format arguably fills a large part of that niche instead, although only for the more common triangle-based mesh objects and materials. It is a continually evolving standard however, so it is possible that support for more exotic objects and materials will be added over time. In the interim, there will always be a need to organically extend the format to support special use-cases, particularly when researching novel methods. To that end, the tooling presented in this thesis to do just that will likely remain useful for many years to come.

Similarly, opacity micromaps are extremely likely to remain as an important optimization for the ray-tracing pipeline. In terms of research however, it occupies a similar position as that of acceleration structures: It is mostly of interest to hardware vendors. That said, the opacity micromap format is substantially easier to work with than the acceleration structure, and is relatively easy to replicate in software, as was demonstrated in Paper III. The compression algorithm presented in the same paper appears to be a bit more of a mixed bag: The compression result itself is excellent. It shrinks micromaps to almost their theoretical limit, but the run-time look-up cost must be improved for it to be usable in a real-time rendering context. Even so, the approach still has merit for other use-cases, such as efficient on-disk storage.

As for the other types of micromaps; the displacement micromaps and the presented attribute micromaps, the future is less certain. As outlined in Paper IV, displacement micromaps have now been practically deprecated, and cannot be used on modern hardware any longer. We can only speculate as to why, but the likely reason for this is due to the algorithmic complexity the approach added to the acceleration structures while not providing any substantial run-time related performance improvements [36]. It is possible to re-implement it in software, as demonstrated by the recent work of Gruen et al. [20], but it is still unclear whether the complexity for these approaches are warranted.

Theoretically, the attribute micromaps presented in Paper IV does not have this kind of limitation, as all attributes are tied to the underlying triangle-surface, and consequently does not impact the acceleration structure. This allows properties to be stored in a compact format, particularly for very dense triangle meshes. However, such a *surface parameterization* represent a substantial break from the already established methods in computer graphics, (e.g., *texture mapping*), and at this time, the possible benefits of this representation does

not justify the required re-engineering work necessary to support the feature in many 3D modeling packages.

The fixed-function operations provided by the format does present an interesting mechanism to address some common ray-tracing pain-points however, such as that of efficient partial-transparency, but these use-cases are so-far too niche to warrant modifications to the existing hardware pipeline. That said, micromaps do represent a good mechanism for providing the hardware pipeline with more information, and it seems likely that it could be useful for similar endeavors in the future.

Ultimately, it will be very interesting to see how ray-tracing will evolve in the future and if these computer graphics methods will find their way into physics somehow, and particularly, if these insights in the use of micromaps will eventually find some use in the hardware accelerated ray-tracing pipeline.

Bibliography

If I have seen further, it is by standing on the shoulders of giants.

- Sir Isaac Newton

Academic References

[1] John Amanatides and Andrew Woo. 'A Fast Voxel Traversal Algorithm for Ray Tracing'. In: *EG 1987 - Technical Papers*. Eurographics Association, 1987. DOI: 10.2312/egtp.19871000.

[2] Pontus Andersson et al. 'ꟻLIP: A Difference Evaluator for Alternating Images'. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), 15:1–15:23. DOI: 10.1145/3406183.

[3] Aristotle. *Problems: v. I, Bk. 1-19*. Ed. by Robert Mayhew. Loeb Classical Library. Cambridge, MA: Loeb Classical Library, Nov. 2011.

[4] Eduard Ayguadé et al. 'The Design of OpenMP Tasks'. In: *IEEE Trans. Parallel Distrib. Syst.* 20.3 (Mar. 2009), pp. 404–418. ISSN: 1045-9219. DOI: 10.1109/TPDS.2008.105. URL: https://doi.org/10.1109/TPDS.2008.105.

[5] Jon Louis Bentley. 'Multidimensional binary search trees used for associative searching'. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[6] Adam Brumm et al. 'Oldest cave art found in Sulawesi'. In: *Science Advances* 7 (2021), eabd4648. DOI: 10.1126/sciadv.abd4648. URL: https://advances.sciencemag.org/content/7/3/eabd4648.

[7] Edwin Earl Catmull. 'A subdivision algorithm for computer display of curved surfaces.' AAI7504786. PhD thesis. The University of Utah, 1974.

[8] P. A. Čerenkov. 'Visible Radiation Produced by Electrons Moving in a Medium with Velocities Exceeding that of Light'. In: *Phys. Rev.* 52 (4 Aug. 1937), pp. 378–379. DOI: 10.1103/PhysRev.52.378.

[9] Subrahmanyan Chandrasekhar. *Radiative Transfer*. Dover Publications, 1960.

[10] John G. Cleary and Geoff Wyvill. 'Analysis of an algorithm for fast ray tracing using uniform space subdivision'. In: *The Visual Computer* 4 (1988), pp. 65–83. URL: https://api.semanticscholar.org/CorpusID:6599825.

[11] Robert L. Cook, Thomas Porter and Loren Carpenter. 'Distributed ray tracing'. In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 137–145. ISSN: 0097-8930. DOI: 10.1145/964965.808590. URL: https://doi.org/10.1145/964965.808590.

[12] James Essinger. *Jacquard's Web: How a Hand-Loom Led to the Birth of the Information Age*. Oxford, UK: Oxford University Press, 2004. ISBN: 9780192805782.

[13] I. Frank and Ig. Tamm. 'Coherent Visible Radiation of Fast Electrons Passing Through Matter'. In: *Selected Papers*. Ed. by Boris M. Bolotovskii, Victor Ya. Frenkel and Rudolf Peierls. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 29–35. ISBN: 978-3-642-74626-0. DOI: 10.1007/978-3-642-74626-0_2.

[14] Erik Ivar Fredholm. 'Sur une classe d'équations fonctionnelles'. In: *Acta Mathematica* 27 (1903), pp. 365–390. DOI: 10.1007/BF02421317.

[15] Akira Fujimoto, Takayuki Tanaka and Kansei Iwata. 'ARTS: Accelerated Ray-Tracing System'. In: *IEEE Computer Graphics and Applications* 6.4 (1986), pp. 16–26. DOI: 10.1109/MCG.1986.276715.

[16] Andrew S. Glassner. 'Space Subdivision for Fast Ray Tracing'. In: *IEEE Computer Graphics and Applications* 4.10 (1984), pp. 15–24. DOI: 10.1109/MCG.1984.6429331.

[17] Jeffrey Goldsmith and John Salmon. 'Automatic Creation of Object Hierarchies for Ray Tracing'. In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20. DOI: 10.1109/MCG.1987.276983.

[18] Cindy M. Goral et al. 'Modeling the interaction of light between diffuse surfaces'. In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 213–222. ISSN: 0097-8930. DOI: 10.1145/964965.808601. URL: https://doi.org/10.1145/964965.808601.

[19] Holger Gruen, Carsten Benthin and Sven Woop. 'Sub-Triangle Opacity Masks for Faster Ray Tracing of Transparent Objects'. In: *Proc. ACM Comput. Graph. Interact. Tech.* 3.2 (Aug. 2020). DOI: 10.1145/3406180. URL: https://doi.org/10.1145/3406180.

[20] Holger Gruen et al. 'Ray Tracing Animated Displaced Micro-Meshes'. In: *Computer Graphics Forum* 43.7 (2024), e15225. DOI: https://doi.org/10.1111/cgf.15225. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.15225. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.15225.

[21] Toshiya Hachisuka and Henrik Wann Jensen. 'Stochastic progressive photon mapping'. In: *ACM Trans. Graph.* 28.5 (Dec. 2009), pp. 1–8. ISSN: 0730-0301. DOI: 10.1145/1618452.1618487. URL: https://doi.org/10.1145/1618452.1618487.

[22] Toshiya Hachisuka, Shinji Ogaki and Henrik Wann Jensen. 'Progressive photon mapping'. In: *ACM Trans. Graph.* 27.5 (Dec. 2008). ISSN: 0730-0301. DOI: 10.1145/1409060.1409083. URL: https://doi.org/10.1145/1409060.1409083.

[23] Ibn al-Haytham. *Kitāb al-Manāẓir (Book of Optics)*. Written ca. 1011–1021 CE.

[24] Christopher S. Henshilwood et al. 'Emergence of modern human behavior: Middle Stone Age engravings from South Africa'. In: *Science* 295.5558 (2002), pp. 1278–1280. DOI: 10.1126/science.1067575.

[25] Adrian Jarabo et al. 'A framework for transient rendering'. In: *ACM Trans. Graph.* 33.6 (Nov. 2014). ISSN: 0730-0301. DOI: 10.1145/2661229.2661251. URL: https://doi.org/10.1145/2661229.2661251.

[26] Henrik Wann Jensen. 'Global illumination using photon maps'. In: *Proceedings of the Eurographics Workshop on Rendering Techniques '96*. Porto, Portugal: Springer-Verlag, 1996, pp. 21–30. ISBN: 3211828834.

[27] Ian Johnston. *The Mozi: A Complete Translation*. The Chinese University of Hong Kong Press, 2010. ISBN: 9789629962708. URL: http://www.jstor.org/stable/j.ctt1pb61v3 (visited on 13/02/2026).

[28] James T. Kajiya. 'The rendering equation'. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, pp. 143–150. ISBN: 0897911962. DOI: 10.1145/15922.15902. URL: https://doi.org/10.1145/15922.15902.

[29] Michael R Kaplan. 'Space-tracing: A constant time ray-tracer'. In: *SIGGRAPH'85 State of the Art in Image Synthesis seminar notes*. Vol. 18. 3. 1985, pp. 149–158.

[30] Tero Karras and Timo Aila. 'Fast Parallel Construction of High-Quality Bounding Volume Hierarchies'. In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99. ISBN: 9781450321358. DOI: 10.1145/2492045.2492055. URL: https://doi.org/10.1145/2492045.2492055.

[31] Timothy L. Kay and James T. Kajiya. 'Ray tracing complex scenes'. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916. URL: https://doi.org/10.1145/15886.15916.

[32] Samuli Laine and Tero Karras. 'Two methods for fast ray-cast ambient occlusion'. In: *Proceedings of the 21st Eurographics Conference on Rendering*. EGSR'10. Saarbrücken, Germany: Eurographics Association, 2010, pp. 1325–1333. DOI: 10.1111/j.1467-8659.2010.01728.x. URL: https://doi.org/10.1111/j.1467-8659.2010.01728.x.

[33] Johann Heinrich Lambert. *Photometria sive de Mensura et Gradibus Luminis, Colorum et Umbrae*. Latin. Augsburg: Eberhard Klett, 1760.

[34] Ben F. Laposky. *Oscillons: Electronic Abstractions*. Electronic images created using analog oscilloscopes; exhibited as early computer art. 1952.

[35] Thomas Larsson and Tomas Akenine-Möller. 'A dynamic bounding volume hierarchy for generalized collision detection'. In: *Comput. Graph.* 30.3 (June 2006), pp. 450–459. ISSN: 0097-8493. DOI: 10.1016/j.cag.2006.02.011. URL: https://doi.org/10.1016/j.cag.2006.02.011.

[36] Andrea Maggiordomo, Henry Moreton and Marco Tarini. 'Micro-Mesh Construction'. In: *ACM Trans. Graph.* 42.4 (July 2023). ISSN: 0730-0301. DOI: 10.1145/3592440. URL: https://doi.org/10.1145/3592440.

[37] Morgan McGuire. *The Graphics Codex*. 2.17. Casual Effects, 2024. URL: https://graphicscodex.com.

[38] Daniel Meister et al. 'A Survey on Bounding Volume Hierarchies for Ray Tracing'. In: *Computer Graphics Forum* (2021). ISSN: 1467-8659. DOI: 10.1111/cgf.142662.

[39] Ken Museth. 'VDB: High-resolution sparse volumes with dynamic topology'. In: *ACM Trans. Graph.* 32.3 (July 2013). ISSN: 0730-0301. DOI: 10.1145/2487228.2487235. URL: https://doi.org/10.1145/2487228.2487235.

[40] Matt Pharr, Greg Humphreys and Wenzel Jakob. *Physically Based Rendering: From Theory to Implementation*. 1st ed. 2004; 2nd ed. 2010; 3rd ed. 2016; 4th ed. 2023. Online 3rd and 4th ed. at https://pbr-book.org/3ed-2018 and https://pbr-book.org/4ed. Morgan Kaufmann and MIT Press, 2023. URL: https://pbr-book.org.

[41] Steven M. Rubin and Turner Whitted. 'A 3-dimensional representation for fast rendering of complex scenes'. In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '80. Seattle, Washington, USA: Association for Computing Machinery, 1980, pp. 110–116. ISBN: 0897910214. DOI: 10.1145/800250.807479. URL: https://doi.org/10.1145/800250.807479.

[42] Hanan Samet. 'The Quadtree and Related Hierarchical Data Structures'. In: *ACM Comput. Surv.* 16.2 (June 1984), pp. 187–260. ISSN: 0360-0300. DOI: 10.1145/356924.356930. URL: https://doi.org/10.1145/356924.356930.

[43] Martin Stich, Heiko Friedrich and Andreas Dietrich. 'Spatial splits in bounding volume hierarchies'. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 7–13. ISBN: 9781605586038. DOI: 10.1145/1572769.1572771. URL: https://doi.org/10.1145/1572769.1572771.

[44] Ivan Edward Sutherland. 'Sketchpad: A Man-Machine Graphical Communication System'. PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 1963. URL: http://hdl.handle.net/1721.1/14979.

[45] T. S. Trowbridge and K. P. Reitz. 'Average irregularity representation of a rough surface for ray reflection'. In: *J. Opt. Soc. Am.* 65.5 (May 1975), pp. 531–536. DOI: 10.1364/JOSA.65.000531. URL: https://opg.optica.org/abstract.cfm?URI=josa-65-5-531.

[46] H. Valladas et al. 'Evolution of prehistoric cave art'. In: *Nature* 413.6855 (Oct. 2001), pp. 479–479. ISSN: 1476-4687. DOI: 10.1038/35097160. URL: https://doi.org/10.1038/35097160.

[47] Ingo Wald. 'On fast Construction of SAH-based Bounding Volume Hierarchies'. In: *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. RT '07. USA: IEEE Computer Society, 2007, pp. 33–40. ISBN: 9781424416295. DOI: 10.1109/RT.2007.4342588. URL: https://doi.org/10.1109/RT.2007.4342588.

[48] Ingo Wald et al. 'State of the Art in Ray Tracing Animated Scenes'. In: *Computer Graphics Forum* 28.6 (2009), pp. 1691–1722. DOI: https://doi.org/10.1111/j.1467-8659.2008.01313.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2008.01313.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01313.x.

[49] Gustaf Waldemarson. *Real-Time Path-Tracing of Ray-Tracing benchmarks of Yore*. Poster presented during the High-Performance Graphics conference 2025 student competetition: https://highperformancegraphics.org/2025/student-competition. 2025. URL: https://gustafwaldemarson.com/misc/hpg25/benchmarks.pdf.

[50] Gustaf Waldemarson and Michael Doggett. 'Color and Attribute Micromaps'. In: *Submission*. 2026.

[51] Gustaf Waldemarson and Michael Doggett. 'Parallel Axis Split Tasks for Bounding Volume Construction with OpenMP'. In: *Proceedings of the 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - GRAPP*. INSTICC. SciTePress, 2025, pp. 347–354. ISBN: 978-989-758-728-3. DOI: 10.5220/0013317100003912.

[52] Gustaf Waldemarson and Michael Doggett. 'Photon Mapping Superluminal Particles'. In: *Eurographics 2020 - Short Papers*. Ed. by Alexander Wilkie and Francesco Banterle. The Eurographics Association, 2020. ISBN: 978-3-03868-101-4. DOI: 10.2312/egs.20201004.

[53] Gustaf Waldemarson and Michael Doggett. 'Succinct Opacity Micromaps'. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7.3 (Aug. 2024). DOI: 10.1145/3675385. URL: https://doi.org/10.1145/3675385.

[54] Bruce Walter et al. 'Microfacet models for refraction through rough surfaces'. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR'07. Grenoble, France: Eurographics Association, 2007, pp. 195–206. ISBN: 9783905673524.

[55] Zhou Wang et al. 'Image quality assessment: from error visibility to structural similarity'. In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861.

[56] Turner Whitted. 'An improved illumination model for shaded display'. In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.

Software and Application Programming Interfaces

[57] AMD. *RDNA4 Shader Instruction Set Architecture*. Tech. rep. AMD, 2025. URL: https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna4-instruction-set-architecture.pdf.

[58] Blender Online Community. *Blender - A 3D Modelling and Rendering Package*. Blender Foundation. Blender Institute, Amsterdam, 2026. URL: http://www.blender.org.

[59] Tim Cheblokov. *Advanced API Performance: Pipeline State Objects*. NVIDIA Developer Technical Blog. July 2023. URL: https://developer.nvidia.com/blog/advanced-api-performance-pipeline-state-objects/.

[60] RenderToolbox Contributors. *RenderToolbox: MATLAB toolbox for PBRT scenes*. GitHub repository. Matlab toolbox for working with PBRT scenes. 2025. URL: https://github.com/RenderToolbox/mPbrt.

[61] Cycles Developers. *Cycles: Physically Based Rendering Engine*. Version latest. Part of the Blender project; Apache License v2. 2026. URL: https://www.cycles-renderer.org/.

[62] *DirectX Engineering Specifications*. Engineering and functional specification repository for DirectX (including Direct3D) maintained by Microsoft. Microsoft. 2026. URL: https://microsoft.github.io/DirectX-Specs/.

[63] Haarm-Pieter Duiker. *PBRTForMaya: Maya plugin for exporting to PBRT*. GitHub repository. Maya plugin to export scenes in PBRT v3 format. 2017. URL: https://github.com/hpd/PBRTForMaya.

[64] Epic Games. *Hardware Ray Tracing in Unreal Engine*. Official Unreal Engine documentation on hardware-accelerated ray tracing. Epic Developer Community. 2026. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/hardware-ray-tracing-in-unreal-engine.

[65] Intel. *Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games*. Tech. rep. Intel, 2023. URL: https://cdrdv2-public.intel.com/766679/intel-rtrt%5C%20-applications-developer-guide-v4.pdf.

[66] Giulio Jiang. *pbrt-v3 Blender exporter*. GitHub repository. Exporter for PBRT version 3 scenes; see GitHub. 2017. URL: https://github.com/giuliojiang/pbrt-v3-blender-exporter.

[67] Burak Kahraman and Timm Dapper. *Cinema 4D exporter for PBRT v3*. Included in the PBRT-v3 distribution. Exporter from Cinema 4D to PBRT v3. 2016. URL: https://github.com/mmp/pbrt-v3/tree/master/exporters/cinema4d.

[68] Khronos Group. *glTF 2.0 Specification*. https://www.khronos.org/gltf/. Accessed: 2026-01-22. 2017.

[69] Khronos Group. *glTF-Blender-IO: Blender glTF 2.0 importer and exporter*. https://github.com/KhronosGroup/glTF-Blender-IO. GitHub repository, Apache-2.0 licensed. Khronos Group, 2026.

[70] Khronos Group. *VK_EXT_opacity_micromap - Vulkan Extension Specification*. 2022. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html.

[71] Khronos Group. *VK_NV_displacement_micromap - Vulkan Extension Specification*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_NV_displacement_micromap.html.

[72] Khronos OpenGL Working Group. *OpenGL 4.6 Core Profile Specification*. https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf. Version 4.6. July 2017.

[73] Nic Nel. *bpbrt4: PBRT-v4 render engine/exporter add-on for Blender*. GitHub repository. Blender add-on exporting to PBRT-v4; based on Blender Mitsuba add-on code. 2025. URL: https://github.com/NicNel/bpbrt4.

[74] NVIDIA. *NVIDIA Turing GPU Architecture Whitepaper*. Tech. rep. NVIDIA, 2023. URL: https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[75] NVIDIA GameWorks. *Displacement MicroMap Toolkit*. 2025. URL: https://github.com/NVIDIAGameWorks/Displacement-MicroMap-Toolkit.

[76] *NVIDIA OptiX Ray Tracing Engine: Programming Guide*. Version 9.1. NVIDIA Corporation. 2025. URL: https://raytracing-docs.nvidia.com/optix9/.

[77] Jim Price. *Houdini PBRT exporters*. GitHub repositories. Exporters from SideFX Houdini to PBRT v3 and PBRT v4. 2023. URL: https://github.com/shadeops/houdini-pbrt-v4.

[78] Juha Sjöholm. *Best Practices for Using NVIDIA RTX Ray Tracing (Updated)*. Nvidia. 25th July 2022. URL: https://developer.nvidia.com/blog/best-practices-for-using-nvidia-rtx-ray-tracing-updated/.

[79] Stig Atle Steffensen. *io_scene_pbrt: Blender exporter for PBRT*. GitHub repository. Exporter for PBRT v4 (under development). 2023. URL: https://github.com/stig-atle/io_scene_pbrt.

[80] *The OpenCL™ Specification, Version 3.0.19*. Latest unified OpenCL API specification (v3.0) published by Khronos Group. The Khronos Group Inc. 2025. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

[81] Unity Technologies. *Unity Real-Time Ray Tracing*. High-level overview of hardware-accelerated ray tracing features in Unity's HDRP. 2026. URL: https://unity.com/ray-tracing.

[82] *Vulkan API Specification*. Version 1.3, with KHR ray tracing extensions. Khronos Group. 2024. URL: https://registry.khronos.org/vulkan/.

[83] Ingo Wald. *pbrt-parser: A simple parser for the PBRT file format*. GitHub repository. Version 1.1. 2019. URL: https://github.com/ingowald/pbrt-parser.

[84] Gustaf Waldemarson. 'Graphics Processing'. Patent Application US20250157145A1. Priority date: 2023-11-15. May 2025. URL: https://patents.google.com/patent/US20250157145A1/en.

[85] Gustaf Waldemarson. *Handling Custom Data in glTF Files with Exporter/Importer Plugins*. Accompanying page: `https://gustafwaldemarson.com/pages/publications/custom-gltf-data`. The Blender Foundation, Youtube. 2023. URL: `https://youtu.be/4fBGM8qc21M`.

[86] Gustaf Waldemarson. *PBRT: Create your own Importers and Exporters*. The Blender Foundation, Youtube. 2024. URL: `https://youtu.be/BEbscsBRIx0`.

Scenes and Assets

[87] Epic Games Amazon NVIDIA. *Open Research Content Archive (ORCA) Assets*. Collection of NVIDIA ORCA 3D assets for graphics research. Includes Amazon Lumberyard Bistro, NVIDIA Emerald Square, SpeedTree Trees and Plants, UE4 Sun Temple, and BEEPLE Zero-Day. 2017. URL: https://developer.nvidia.com/orca.

[88] Amazon Lumberyard. *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)*. http://developer.nvidia.com/orca/amazon-lumberyard-bistro. July 2017. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro.

[89] Cornell Program of Computer Graphics. *Cornell Box Data*. https://bowers.cornell.edu/cornell-box. Accessed: 2026-02-17.

[90] Keenan Crane. *"Yeah Right" 3D Model*. Published as a PBRT scene. Model courtesy of Keenan Crane. URL: https://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/.

[91] Marko Dabrovic. *Sponza Atrium*. 3D scene model. Originally created by Marko Dabrovic; Updated by Morgan McGuire. 2002. URL: https://casual-effects.com/g3d/data10/index.html.

[92] Morgan McGuire Frank Meinl Marko Dabrovic. *CryTek Sponza*. https://www.cryengine.com/asset-db/product/crytek/sponza-sample-scene. 2011.

[93] GAMMA Group, University of North Carolina. *Power Plant 3D Model*. Available online. Coal-fired power plant 3D model (12,748,510 triangles) used in graphics research. 2001. URL: http://gamma.cs.unc.edu/POWERPLANT/.

[94] Timm Dapper Jan-Walter Schliep Burak Kahraman. *Landscape*. https://www.laubwerk.com. 2016.

[95] Khronos Group. *glTF Sample Assets (v2.0)*. https://github.com/KhronosGroup/glTF-Sample-Assets/tree/Models. Accessed: 2025-09-16. 2023.

[96] Guillermo M. Leal Llaguno. *San Miguel*. https://www.pbrt.org/scenes-v3/. 2010.

[97] Morgan McGuire. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data.

[98] Frank Meinl et al. *Intel Sample Library*. https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-processing-research/samples.html. 2022.

[99] *The Stanford 3D Scanning Repository*. https://graphics.stanford.edu/data/3Dscanrep/. Accessed: 2026-02-21.

Scientific Publications

If you always know exactly what to do, is it really worth doing?

Included Papers

This chapter includes the papers from the following publications:

I **Photon Mapping Superluminal Particles**

Gustaf Waldemarson and Michael Doggett
Eurographics 2020 - Short Papers

II **Parallel Axis Split Tasks for Bounding Volume Construction with OpenMP**

Gustaf Waldemarson and Michael Doggett
Proceedings of the 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - GRAPP

III **Succinct Opacity Micromaps**

Gustaf Waldemarson and Michael Doggett
Proceedings of the ACM on Computer Graphics and Interactive Techniques

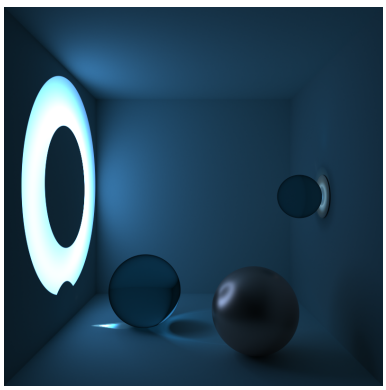
IV **Color and Attribute Micromaps**

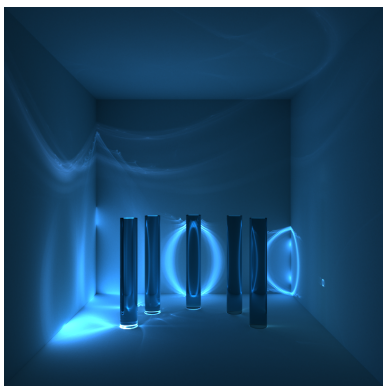
Gustaf Waldemarson and Michael Doggett
In submission

All papers are reproduced with permission of their respective publishers.

Paper I

Photon Mapping Superluminal Particles





I

Summary

When we think about light sources in graphics, we often associate these with a point or surface that emits light with a certain fixed color, or a energy intensity.

Almost all types of light sources are significantly more complex in practice however, and sometimes we need to simulate the light-generating process a bit closer to faithfully mimic the phenomena. In this work we use ray-tracing to both simulate the emission of very energetic particles and the photons that these generate, creating images of scenes entirely lit by *Cherenkov radiation*.

Conference poster in Appendix A.

Photon Mapping Superluminal Particles

G. Waldemarson[1,2] and M. Doggett[2]

[1] Arm Ltd, Sweden
[2] Lund University, Sweden

Abstract

One type of light source that remains largely unexplored in the field of light transport rendering is the light generated by superluminal particles, a phenomenon more commonly known as Cherenkov radiation [Č37]. By re-purposing the Frank-Tamm equation [FT91] for rendering, the energy output of these particles can be estimated and consequently mapped to photons, making it possible to visualize the brilliant blue light characteristic of the effect. In this paper we extend a stochastic progressive photon mapper and simulate the emission of superluminal particles from a source object close to a medium with a high index of refraction. In practice, the source is treated as a new kind of light source, allowing us to efficiently reuse existing photon mapping methods.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;***

1. Introduction

In the late 19th and early 20th century, the phenomenon today known as Cherenkov radiation were predicted and observed a number of times [Wat11] but it was first properly investigated in 1934 by Pavel Cherenkov under the supervision of Sergey Vavilov at the Lebedev Institute [Č37]. A few years later, Cherenkov's colleagues Igor Tamm and Ilya Frank developed the theory for the effect in 1937 within the framework of Einstein's special relativity, effectively summarizing the phenomenon in the so-called Frank-Tamm equation [FT91].

Normally, the phenomenon is observed in nuclear reactors or in the proximity of highly radioactive materials, see e.g., figure 6. Thus, the effect can also be used in the reverse order: the detected light from the Cherenkov effect can give a hint about the remaining radioactivity of the substance. For this reason, the effect is often used in detectors for radioactive substances [Wat11]. Additionally, there is an increasing number of attempts to use the phenomenon in the field of medicine, either as a complement to other methods or as a completely novel imaging technique for in vivo imaging [CB17].

1.1. Cherenkov Radiation

Physically, as a charged particle (e.g., an electron) moves through a medium it will interact with the particles of that medium, temporarily exciting its electrons and causing them to emit electromagnetic waves moving at the speed of light for the current medium (formally this is referred to as the phase speed of the medium). According to Huygen's principle, these waves travel outwards spherically from the point interaction. Normally, these spheres will accumulate in the direction of the particle but will not otherwise cross one another, as depicted in figure 1. However, if the particle travels faster than these spheres the waves will constructively interfere, generating coherent photons at an angle proportional to the velocity of the particle, similar to the propagation of a sonic boom from supersonic aircraft. Formally, this occurs when the criterion $\frac{c_0}{n} = c_m < v_p < c_0$ is fulfilled, where c_0 is the speed of light in vacuum and n is the refractive index of the medium. That is, Cherenkov radiation occurs when the particle travels faster than light inside the current medium. Additionally, the angle at which the coherent photons are emitted can be computed from figure 2 using trigonometry:

$$cos(\theta) = \frac{c_0}{v_p n}$$

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

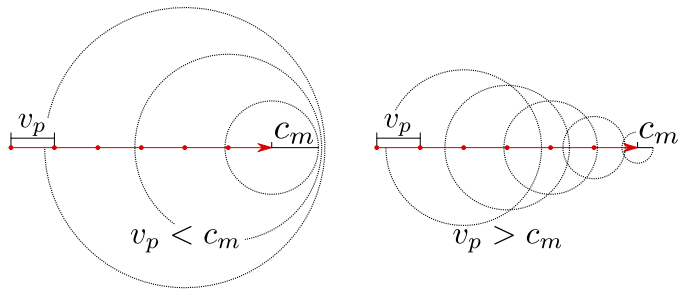


Figure 1: *Schematic view of the creation of Cherenkov photons where c_m represents the speed of light in the medium and v_p is the speed of the particle. Structurally, this is similar to a sonic boom for sound waves.*

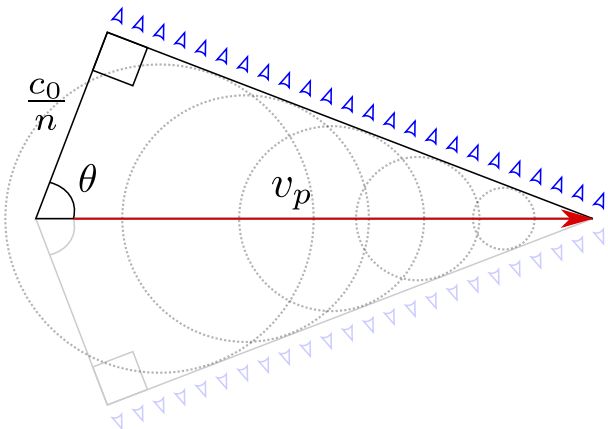


Figure 2: *The angle at which Cherenkov radiation will be emitted at each unit distance. Note that the particle in this figure is significantly faster than the other ones to better illustrate the concept.*

1.2. The Frank-Tamm Equation

The Frank-Tamm equation [FT91] describes how the photon energy gets distributed by charged particles that travel faster than the speed of light in the current medium. Typically, it is given in the form:

$$\frac{d^2E}{dxd\omega} = \frac{q^2}{4\pi}\mu(\omega)\omega\left(1 - \frac{c_0^2}{v^2n^2(\omega)}\right) \tag{1}$$

Where:

ω Is the angular frequency of the photon.
$\mu(\omega)$ The permeability of the medium.
q The electric charge of the particle.
$n(\omega)$ The refractive index of the medium.
v The speed of the particle.
c_0 The speed of light in a vacuum.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

$\frac{d^2E}{dxd\omega}$ Energy emitted per unit length x traveled and frequency ω.

For our purposes, it is more practical to use wavelengths (λ) and number of emitted photons (N). Thus, the equation is rewritten to the following:

$$\frac{d^2N}{dxd\lambda} = -\frac{2\pi\alpha\mu(\lambda)}{\lambda^2}\left(1 - \frac{c_0^2}{v^2n^2(\lambda)}\right) \tag{2}$$

Where α is the free structure constant.

If we additionally assume that $\mu(\lambda) \approx 1$ the Cherenkov radiation spectra is straight-forward to visualize, as shown in figure 3. Note how

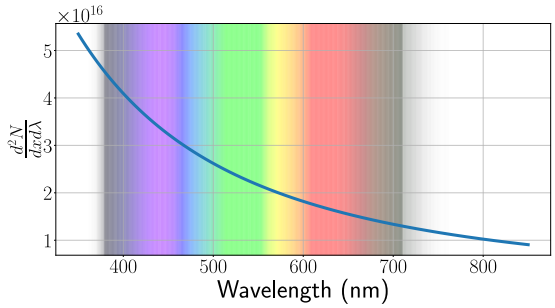


Figure 3: *Example of how equation 2 maps particle energy to photons. Depicted here for an electron moving through heavy water ($n \approx 1.328$) at a speed of $0.8c_0$, assuming that $\mu(\lambda) = 1$.*

the curve rises rapidly for shorter wavelengths and especially in the ultraviolet range which is one of the reasons that give Cherenkov its brilliant blue glow. As given, this equation would simply continue to rise for shorter and shorter wavelengths but in the region of anomalous dispersion the refractive index drops below unity, effectively stopping the effect. Similarly, at longer wavelengths in the infrared spectra and beyond, the effect stops due to material self-absorption [CB17].

2. Related Work

While the Cherenkov effect is a widely understood physical phenomenon with various avenues of active research within medicine [CB17], nuclear physics and astronomy. To our knowledge, the effect has not been studied in a ray tracing or computer graphics context. That said, there exists a few frameworks for simulating the Cherenkov effect such as Geant4, but to our knowledge, they are not based on ray tracing techniques. Additionally, a few other phenomena involving charged particles have been studied within this context, one of which being the rendering of the formation of auroras [LG11], but they are fundamentally different from the effect discussed here.

3. Photon Mapping Algorithm

The idea is to extend the progressive photon mapping algorithm [HOJ08] with an additional pass: Trace M charged particles from predetermined sources in the scene and store the path that they traveled. This paper modifies a traditional photon mapping algorithm to use particle paths as light sources that can be fed into the remaining passes of the algorithm with a different sampling strategy:

1. Randomly choose a point along the path that the charged particle traveled.
2. Determine the index of refraction at the location.
3. If the particle is not superluminal at that point, emit a photon in a random direction.
4. Otherwise, compute the Cherenkov emission angle (see figure 2) and randomly choose a direction perpendicular to the cone surface to emit a photon towards.
5. Evaluate the Frank-Tamm equation for the particle and use the resulting spectra as photon color.
6. Trace the photon as in [HOJ08].

Intuitively, the algorithm can be summarized as follows: As a charged particle travels through matter, it will excite the atoms of the material causing it to emit photons uniformly in a sphere. However, if the particle is in a medium where it is superluminal the photons will instead constructively interfere and primarily be emitted in the cone defined by the Cherenkov angle. The sampling scheme is also summarized in figure 4.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

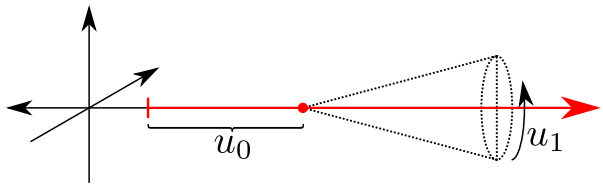


Figure 4: *Illustration of how uniform random variables are used to generate new photons. u_0 selects a point along the particle path, u_1 retrieves a value on the disk defined by the cherenkov angle.*

3.1. Photon Density Distributions

Photon mapping is a Monte Carlo simulation and as such, it is often desirable to use statistical Russian roulette to improve rendering performance. To do so however, the photon probability density function $p(\mathrm{o}, \omega)$ must be estimated. Given the particle light source defined by our algorithm, it can be derived as follows:

$$p(\mathrm{o}, \omega) = p(\mathrm{o}) \cdot p(\omega)$$

Where $p(\mathrm{o})$ and $p(\omega)$ are the density for the photon origin and direction respectively. Of these, $p(\mathrm{o})$ is straightforward to compute: The particle origin will always be somewhere along the path of the particle, hence the density is simply:

$$p(\mathrm{o}) = \frac{1}{\text{total particle length}}$$

As the ray direction in our algorithm can be either uniformly distributed on a sphere or along the Cherenkov angle, $p(\omega)$ is slightly trickier to estimate. To do so, we first define a probability mixture model as follows:

$$p(\omega) = p(S)p(\omega_\mathrm{c}) + (1 - p(S))p(\omega_\mathrm{s})$$

Where $p(S)$ is the probability of the particle being superluminal and $p(\omega_\mathrm{c})$ and $p(\omega_\mathrm{s})$ are the probability densities of emitting light along the Cherenkov angle and uniformly in a sphere respectively. As $p(\omega_\mathrm{c})$ and $p(\omega_\mathrm{s})$ are well known, this can be reduced to:

$$\begin{aligned} p(\omega) &= p(S)p(\omega_\mathrm{c}) + (1 - p(S))p(\omega_\mathrm{s}) \\ &= \frac{p(S)}{2\pi} + \frac{1 - p(S)}{4\pi} = \frac{1 + p(S)}{4\pi} \end{aligned}$$

Also, if it is possible to find all intervals where a particle is superluminal, $p(S)$ can be estimated as:

$$p(S) = \frac{\text{superluminal path lengths}}{\text{total path length}}$$

4. Results

The algorithm itself is implemented as a new kind of light source in the ray tracing framework PBRT [PJH16], the code of which will be made available on GitHub. Note that no other part of the framework had to be changed to add this feature.

To demonstrate the effect in a simple setting, a variant of the Cornell box is used where at least one of the objects emits charged particles. Additionally, as Cherenkov radiation is most commonly observed in the context of nuclear reactions, an additional scene of a nuclear reactor is created based on photographs of the Reed Research Reactor. By replicating the phenomenon in such a setting the output image from the photon mapper will be comparable to photographs of the same reactor when it is in operation. The resulting images from these scenes can be seen in figures 5 and 6.

5. Discussion

The Cornell box scenes clearly visualizes the cone shape of the emitted Cherenkov photons and how they are affected by varying refractive indices. Further, the Reed model shows how the sheer mass of photons eventually diffuses to something akin to a single source of light. Naturally though, the render differs from the photograph but that is to be expected for a number of reasons, chief of which is the inability to closely model particle absorption properties in the scene.

Note that the brilliant blue light that Cherenkov radiation creates can be faked using conventional virtual light-sources, but as we demonstrated with PBRT the effect can be implemented succinctly as a custom light-source without changing the ray tracer itself. Furthermore,

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

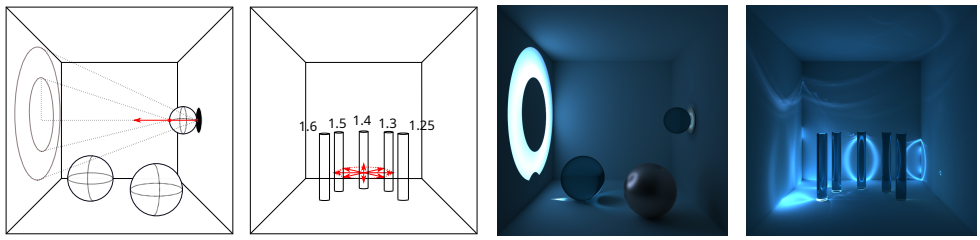


Figure 5: *Schematic and renderings of the Cornell box. From left to right: (a) Schematic drawing with the particle and Cherenkov angle highlighted. (b) Schematic with a particle source emitting uniformly from a small cylinder in the box center towards cylinders with varying refractive indices. (c, d) Rendering of (a) and (b) respectively.*

there may be cases where simulating the effect might be useful, such as generating reference images that artists can base their work on. Additionally, this kind of simulation could be used as a reference for how a reactor vessel should look like during operation, prior to it even being built. Further, since the phenomenon is utilized in detectors, this kind of simulation could give an idea of where such devices should be placed in a reactor environment.

5.1. Future Work

In this work, it was assumed that the emitted particles traveled in straight lines similar to normal rays. In reality, particles interact with matter to a larger degree and often change propagation direction afterwards. Thus it may be of interest to model the particle path as more of a random walk instead. It is also worth noting that the current algorithm only works for forward and bidirectional ray tracers. It is not clear how this kind of light source would be sampled in backward rendering frameworks, such as those based on path tracing [Kaj86]. Also, this work primarily focused on surface photon-mapping but as Cherenkov radiation is normally seen as a volumetric effect it should be analyzed in a setting where this can be taken into account, such as in [JC98], [JNSJ11] or [NNDJ12]. Additionally, the passage of a charged particle is an inherently transient event. For this reason, it may be interesting to integrate the effect in a transient photon rendering framework, such as the one described by [MGJ*19].

6. Conclusions

This paper describes an extension to the stochastic progressive photon mapper algorithm [HOJ08] which allows it to visualize a phenomenon from nuclear physics that typically only occurs under specific conditions, namely the Cherenkov effect. We implemented the effect in an existing physically based rendering framework (PBRT), which produced the expected results. It also shows that similar ray tracing techniques may be applicable to other phenomena from nuclear physics. Additionally, with increasing focus on using the Cherenkov in medicine it may be of interest to use ray tracing algorithms to accelerate their applications or improve their methods.

6.1. Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We would also like to thank Arm Sweden AB for letting Gustaf pursue a PhD as one of their employees. Additionally, we would like to thank Pierre Moreau and Simone Pellegrini for their valuable input during this work.

References

[CB17] CIARROCCHI E., BELCARI N.: Cerenkov luminescence imaging: physics principles and potential applications in biomedical sciences. *EJNMMI Phys 4*, 1 (Dec 2017), 14. doi:10.1186/s40658-017-0181-8. 1, 3

[FT91] FRANK I., TAMM I.: *Coherent Visible Radiation of Fast Electrons Passing Through Matter*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 29–35. doi:10.1007/978-3-642-74626-0_2. 1, 2

[HOJ08] HACHISUKA T., OGAKI S., JENSEN H. W.: Progressive photon mapping. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 130:1–130:8. doi:10.1145/1409060.1409083. 3, 5

[JC98] JENSEN H. W., CHRISTENSEN P. H.: Efficient simulation of light transport in scenes with participating media using photon maps. SIGGRAPH 1998, ACM, pp. 311–320. doi:10.1145/280814.280925. 5

[JNSJ11] JAROSZ W., NOWROUZEZAHRAI D., SADEGHI I., JENSEN H. W.: A comprehensive theory of volumetric radiance estimation using photon points and beams. *ACM Transactions on Graphics (Presented at SIGGRAPH) 30*, 1 (Jan. 2011), 5:1–5:19. doi:10/fcdh2f. 5

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

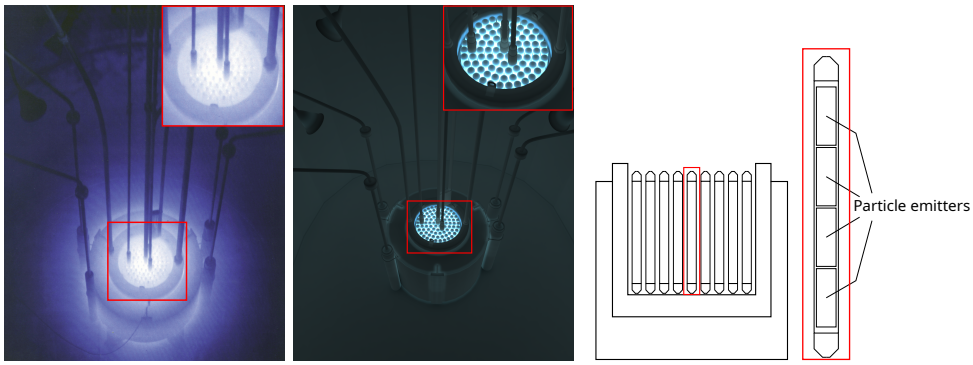


Figure 6: *From left to right: (a) A photograph of the Reed research reactor. (b) Rendering of a reactor model based on the Reed reactor using our algorithm with a large number of particle emitters. Note how the large amount of Cherenkov radiation coalesce into something resembling a single light source. (c) Schematic of the reactor interior and particle emitter structure in the model.*

[Kaj86] KAJIYA J. T.: The rendering equation. *SIGGRAPH '86 20*, 4 (Aug. 1986), 143–150. doi:10.1145/15886.15902. 5

[LG11] LAWLOR O., GENETTI J.: Interactive volume rendering aurora on the gpu. *Journal of WSCG 19* (01 2011), 25–32. 3

[MGJ*19] MARCO J., GUILLÉN I., JAROSZ W., GUTIERREZ D., JARABO A.: Progressive transient photon beams. *Computer Graphics Forum 38*, 6 (2019). 5

[NNDJ12] NOVÁK J., NOWROUZEZAHRAI D., DACHSBACHER C., JAROSZ W.: Virtual ray lights for rendering scenes with participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 31*, 4 (July 2012). doi:10/gbbwk2. 5

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation (3rd ed.)*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Oct. 2016. 4

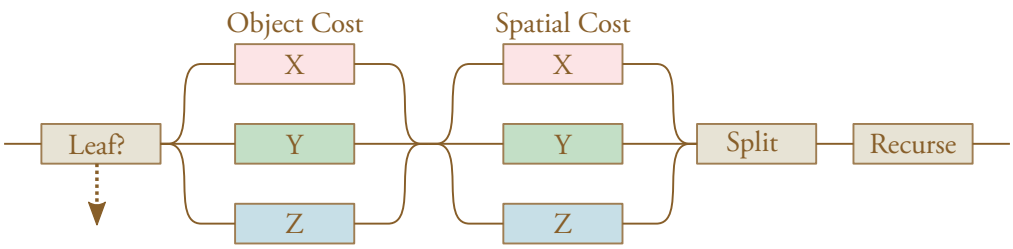
[Č37] ČERENKOV P. A.: Visible radiation produced by electrons moving in a medium with velocities exceeding that of light. *Phys. Rev. 52* (Aug 1937), 378–379. doi:10.1103/PhysRev.52.378. 1

[Wat11] WATSON A. A.: The discovery of cherenkov radiation and its use in the detection of extensive air showers. *Nuclear Physics B - Proceedings Supplements 212-213* (Mar 2011), 13–19. doi:10.1016/j.nuclphysbps.2011.03.003. 1

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

Paper II

Parallel Axis Split Tasks for Bounding Volume Construction with OpenMP



Summary

Ray-tracing large or complex scenes can be very time-consuming. Thus, it is important to leverage acceleration structures to speed up this process as much as possible, and among these structures, the *spatial split bounding volume hierarchy* is among the best at doing so.

It is however one of the slowest to build, something that must be done before the ray-tracing process can even start. Thus, in this work we investigate various methods of parallelizing the build process of this structure using the OpenMP multiprocessing framework.

Conference poster in Appendix D.

Parallel Axis Split Tasks for Bounding Volume Construction with OpenMP®

Gustaf Waldemarson[a,1,2] and Michael Doggett[b,1]
[1]*Department of Computer Science, Lund University, Sweden*
[2]*Arm, Lund, Sweden*
gustaf.waldemarson@cs.lth.se, michael.doggett@cs.lth.se

Keywords: Ray-tracing, Bounding Volume Hierarchy, OpenMP, Parallelization

Abstract: Many algorithms in computer graphics make use of acceleration structures such as Bounding Volume Hierarchies (BVHs) to speed up performance critical tasks, such as collision detection or ray-tracing. However, while the typical algorithms for constructing BVHs are relatively simple, actually implementing them for performance critical systems is still challenging. Further, to construct them as quickly as possible, it is also desirable to parallelize the process. To that end, parallelization APIs such as OpenMP® can be leveraged to greatly simplify this matter. However, BVH construction is not a trivially parallelizable problem. Thus, in this paper we propose a method of using OpenMP® tasking to further parallelize the spatial splitting algorithm and thus improve construction performance. We evaluate the proposed way and compare it with other ways of using OpenMP®, finding that some of these work well to improve the construction time by between 3 and 5 times on an 8-core machine with a minimal amount of work and negligible quality reduction of the final BVH.

1 INTRODUCTION

Bounding volume hierarchies are arguably one of the most important data-structures currently in widespread use in the field of computer graphics, and it is often prominently used for ray-tracing during image synthesis. However, it is also used for various other tasks, such as collision detection or occlusion based audio mixing (Fowler et al., 2014). As such, it is often important to be able to create these hierarchies with as high quality as possible, thus ensuring that when the structure is used to accelerate some task, that *query* operation is as fast as possible.

Typically, the recursive algorithms used to build these structures are relatively simple, but rewriting them for maximum throughput in a parallelized context can be challenging. Thus, modern versions of parallelization APIs such as OpenACC or OpenMP® (Dagum and Menon, 1998) can be leveraged to trial various paralellization strategies before committing to a particular approach, or in other cases, be used directly in the original algorithm. However, there are often multiple ways to apply these APIs. As such, finding the most performant way to use them can be a beneficial endeavor.

[a] https://orcid.org/0000-0003-2524-0329
[b] https://orcid.org/0000-0002-4848-3481

2 RELATED WORK

Over the years, many variations of acceleration structures have been invented, notable examples being octrees (Meagher, 1980), kd-trees (Bentley, 1975) and bounding volume hierarchies, or BVHs. Lately however, BVHs have become the more popular category for four main reasons: They have a predictable memory footprint, queries are robust and efficient, they easily adapt to dynamic geometry, and most crucially: The build itself is scalable; allowing users to either quickly create a hierarchy with possibly slow queries, or, to spend more time upfront to yield potentially faster ones (Meister et al., 2021).

Thus, as it is assumed that construction of an optimal BVH is an NP-hard problem (Karras, 2012), numerous heuristics and algorithms have been developed for generating these hierarchies, and depending on the target application, one of the above approaches are typically preferred:

1. For interactive applications, such as real-time ray-tracing, fast builders running on the GPU are predominantly used, such as the LBVH (Lauterbach et al., 2009), HLBVH (Pantaleoni and Luebke, 2010), and more recently, the H-PLOC algorithm by (Benthin et al., 2024).
2. For offline ray-tracing applications, such as those described by (Pharr, 2018), slower builders, such the SBVH (Stich et al., 2009) or PRBVH (Meister and Bittner, 2018) may be preferable, where any improvement in the quality of the BVH is often recovered during the actual ray-tracing phase (Aila and Laine, 2009).

No matter the application however, it is always desirable to be able to create these structures as quickly as possible. To that end, these build processes are usually parallelized as much as possible. Fast builders typically do this by relaxing some spatial constraints to expose more parallelism, making them amenable to fast GPU implementations. In contrast, quality focused builders typically create their hierarchies with a CPU implementation, as that often provides a bit more flexibility when analyzing the input geometry (Ganestam et al., 2015; Wald et al., 2014). However, many of these algorithms build the hierarchy from the top-down, thus initially suffering from poor scaling in the first few splitting tasks. To that end, a number of parallelization schemes have been proposed to extract additional parallelism from these early splits (Wald, 2007; Wald, 2012; Fuetterling et al., 2016). These approaches typically attempt to split up the computations over all primitives, thus providing a large amount of potential parallelism, but requires a number of complex synchronization mechanisms to function. In contrast, in this paper we propose an arguably simpler approach by only parallelizing over the split axes, thus losing some opportunities for parallelization, but in turn only requiring a relatively simple synchronization method.

3 BACKGROUND

This section provides relevant background information about the SBVH algorithm (Stich et al., 2009) targeted for parallelization with OpenMP® in this work.

3.1 Spatial Split BVH

While BVHs have many great qualities, they can perform poorly in scenes with many overlapping primitives, as is often the case with triangle meshes. In those cases, Kd-trees (Bentley, 1975) are typically able to achieve higher ray-tracing performance. To that end, (Stich et al., 2009) developed a variation of the BVH construction algorithm that drew inspiration from the spatial splits used by kd-trees, thus creating one of the highest performing triangle based BVH algorithms in terms of query-time, which is typically referred to as the *SBVH* algorithm. However, while the query-times are fast, its major drawback is the construction time: It is typically the slowest BVH construction algorithm in widespread use. Furthermore, this algorithm is challenging to parallelize, as part of its operation depend on being able to dynamically create new references to triangles with subdivided bounding boxes.

This particular aspect was improved by (Ganestam and Doggett, 2016), who noted that only around 10 % extra references are needed in most scenes. Thus, by pre-allocating memory for these and distributing them in each split task, parallelization gets a bit easier. Thus, in this paper we further simplify this matter, showing how the SBVH algorithm can be easily parallelized with the help of OpenMP®.

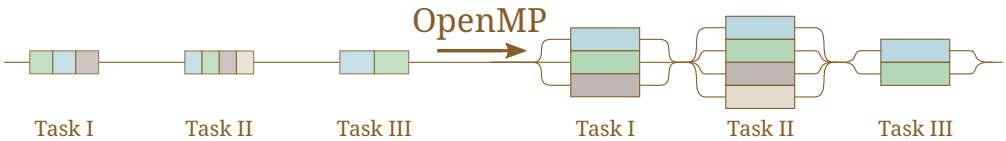


Figure 1: Graphical visualization of the typical OpenMP® usage for splitting up tasks into parallel regions.

```
size_t n = 8;
#pragma omp parallel for num_threads(n)
for (size_t i = 0; i < height; ++i)
{
    for (size_t j = 0; j < width; ++j)
    {
        im[i][j] = ray_trace(i, j);
    }
}
```

Figure 2: A simple `for`-loop parallelized using an OpenMP®compiler directive.

3.2 OpenACC and OpenMP®

OpenACC and OpenMP® are APIs for performing numerous types of multi-processing tasks in a convenient and portable fashion in the `C`, `C++` and `Fortran` programming languages through the use of compiler directives and library routines. Both of these are managed by non-profit organizations: The OpenMP Architecture Review Board, and the OpenACC organization, jointly governed by all major compiler and hardware developers with collaboration from their user communities.

As illustrated in figure 1, these APIs provide a relatively simple way of successively parallelizing portions of a program, and the simplest application of it is typically through the use of the `parallel for` compiler `pragma` from the OpenMP® API, as shown in figure 2.

OpenMP® 3.0 and onwards also enables the manual creation of parallel tasks that may be submitted to a thread-pool, a process typically referred to as *tasking*. Further, tasks may even recursively create more tasks, as seen in figure 3, thus enabling complex algorithms to be parallelized in simple fashion (Ayguadé et al., 2009). However, some care is still needed to ensure that each task is able to perform a suitable amount of work to account for the overhead of its creation. Beyond this, OpenMP® also provide directives for automatically converting the iterations of loops to tasks with the `taskloop` directive and even performing guided `SIMD` vectorization of loops.

In contrast, OpenACC was originally intended for offloading tasks to discrete accelerator devices, thus providing a simple interface to program coprocessors such as GPUs. However, modern versions of this API can also parallelize on the host CPU when required. Additionally, as of OpenMP®4.0, similar device offloading capabilities are available for that API as well, even if the performance of these features may be a bit worse (Usha et al., 2020).

Still, applying device offloading correctly often requires significantly more effort to ensure that the data can be transferred correctly to the coprocessor, often forcing a major restructuring of the original algorithms. As such, these types of approaches are out of scope for this paper.

4 ALGORITHM

This section provides a high-level overview of how the SBVH algorithm recursively constructs a hierarchy from a collection of primitive references, i.e., a set of triangles with potentially subdivided bounding boxes. Our contribution for parallelizing this algorithm with OpenMP® is also described here.

4.1 SBVH Splitting Tasks

In algorithm 1 each SBVH splitting task can recursively create two more work-packets up to the point that it decides to create a leaf-node instead, in a fashion that is very similar to the OpenMP® tasking example in figure 3. This also means that the algorithm is not able to run at full capacity until enough tasks have been spawned to

```
int fibonacci(int n)
{
    int fn1, fn2;
    if (n == 0 || n == 1)
        return n;
#pragma omp task shared(fn1)
    fn1 = fibonacci(n - 1);
#pragma omp task shared(fn2)
    fn2 = fibonacci(n - 2);
#pragma omp taskwait
    return fn1 + fn2;
}
```

Figure 3: A more complex parallelization example to demonstrate the use of OpenMP® tasking. Note that, while illustrative, this particular example would likely not benefit much from parallelization as the overhead of creating tasks likely outweigh the cost of the work itself.

```
Fn build(references, bounds):
    if create leaf? then
        return;
    end
    obj ← object_split(references, bounds);
    spt ← spatial_split(references, bounds);
    if spt.cost ≤ obj.cost then
        perform spatial split;
    else
        perform object split;
    end
(1) build(left-references, left-bounds);
(2) build(right-references, right-bounds);
EndFn
```

Algorithm 1: High-level overview of the SBVH construction algorithm. The functions `object_split` and `spatial_split` are described in algorithms 2 and 3 respectively. Further, lines that may be parallelized with tasks are marked with (1) and (2) as is described in section 4.2.

```
Fn object_split(references, bounds):
(3) foreach axis do
        sort references;
        foreach reference do
            estimate split cost;
        end
    end
    return optimal split cost and location;
EndFn
```

Algorithm 2: High level overview of the BVH object split estimation: Find the appropriate splitting axis and segment the objects to the left and right side of it. Note that the axis-loop on line (3) may be parallelized as described in section 4.2.

keep all available threads occupied. To that end, we propose that further subdividing the splitting task itself may expose more beneficial parallelism during the early stages of the SBVH construction. E.g., by creating tasks for searching each splitting plane along each of the primary axes marked in algorithms 2 and 3, and visualized in figure 4. Further, this may allow more expensive splitting heuristics to be evaluated each time, as more of them can be tried in parallel. E.g., more bins can be used for the binned surface area heuristic (SAH) by (Wald, 2007), or a different, more expensive heuristics such as the original SAH variant proposed by (Goldsmith and Salmon, 1987; MacDonald and Booth, 1990) can be used. This could theoretically improve the BVH quality, while still providing some balance between the amount of work being done and the time it takes to execute each task.

```
Fn spatial_split(references, bounds):
(4)   foreach axis do
        foreach reference do
          chop references into bins;
          split references on bin boundaries;
        end
        find axis splitting plane;
      end
      return optimal split axis and plane;
    EndFn
```

Algorithm 3: High level overview of the BVH spatial split estimation: Find the appropriate splitting axis and plane, and segment or create references to the left and right side of it. Note that the axis-loop on line (4) may be parallelized as described in section 4.2.

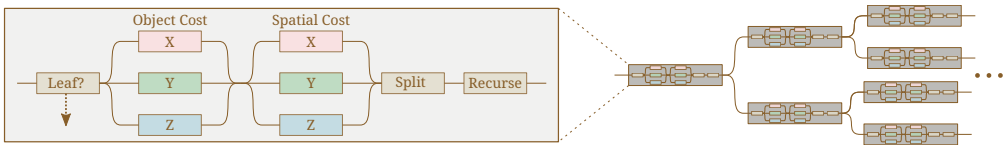


Figure 4: Graphical representation of the task creation process during the construction of an SBVH as well as the further parallelizable regions of a single BVH splitting task.

4.2 Variants

In order to evaluate the potential SBVH build time improvement from multithreading with various OpenMP® constructs, we apply one or more source level patches to a base implementation of the algorithm; effectively inserting the necessary compiler directives (i.e., `#pragma omp ...`) at the correct locations. In total, we evaluate five different variants of this approach:

NoOpenMP Reference implementation without any OpenMP® directives.

TaskingOnly Parallel tasks are created using the `#pragma omp task` directive for each recursive call to `build` in algorithm 1, similar to the tasking example in figure 3.

Tasks Same as *TaskingOnly*, but create additional tasks for each iteration of the object and spatial axis search loops, i.e., for the marked loops in algorithms 2 and 3 and ensure that these tasks are synchronized afterwards using the `#pragma omp taskwait` directive.

Taskloop Same as *Tasks*, but use the `#pragma omp taskloop` directive instead, thus avoiding the need for explicit task synchronization through the `#pragma omp taskwait` directive.

ParallelFor Same as *TaskingOnly*, but use nested parallelism for each object and spatial axis search using the `#pragma omp parallel for` directive, similar to the example in figure 2.

Further, we also investigate *only* parallelizing the object and spatial split search tasks, i.e., we do not parallelize the recursive splits in algorithm 1. However, these variants are not expected to scale beyond three available threads as there are only three axes to search in each task. To that end, we test the following additional variants:

NoTaskingFor Use the `#pragma omp parallel for` directive to search each axis, as in the *ParallelFor* variant.

NoTaskingTasks Same as *NoTaskingFor*, but use OpenMP® task constructs as in the *Tasks* variant.

NoTaskingTaskloop Same as *NoTaskingTasks*, but use the *#pragma omp taskloop* directive instead, same as for the *Taskloop* variant.

5 RESULTS

All BVH construction algorithms and their variations ran on an Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz built by the `gcc` (gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0) and `clang` (Ubuntu clang version 14.0.0-1ubuntu1.1) compilers.



Sponza: 393 meshes, 262 267 triangles.



Hairball: 2 meshes, 2 880 002 triangles.



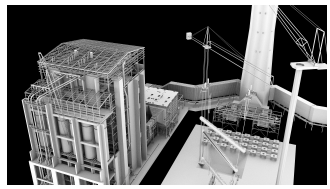
Bistro: 1 mesh, 3 847 246 triangles.



San-Miguel: 287 meshes, 9 980 699 triangles.



Buddah: 1 mesh, 1 087 720 triangles.



Powerplant: 21 meshes, 12 759 246 triangles.

Figure 5: The scenes investigated during this work along with their mesh and triangle counts.

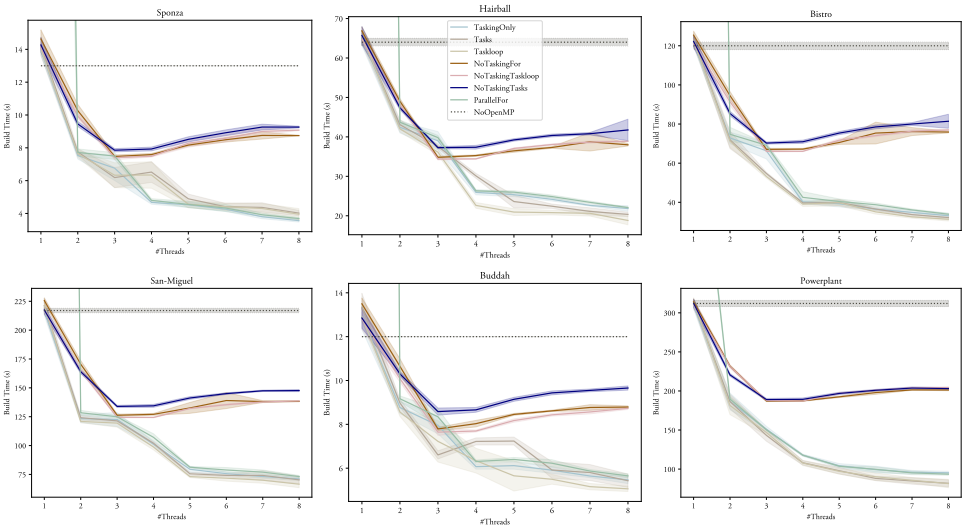


Figure 6: Visualization of how each variant scales with more available threads in the `gcc` implementation.

Each scene was rendered with an `OpenCL` based ray-tracer running on an NVIDIA GeForce RTX 3060 GPU using an ambient occlusion algorithm, example renders of which can be seen in figure 5.

The results depicting how these variants scale with additional threads can be found in figures 6 and 7 for `gcc` and `clang` respectively, clearly demonstrating that OpenMP® is able to provide a substantial improvement to the BVH construction time: Up to 5 times faster than the single thread result on our 8-core setup. Thus, proving that the additional task parallelization of the object and spatial axis searches proposed in section 4.1 is able to provide some additional benefits. However, there is only a non-significant difference between using plain tasks (*Tasks* and *NoTaskingTasks*) or using the `taskloop` directive (*Taskloop* and *NoTaskingTaskloop*), as such, performance-wise, it does not matter which of these are actually used, but the `taskloop` directive is usually a bit easier to read at a glance and does not require explicit synchronization, and may thus be preferred.

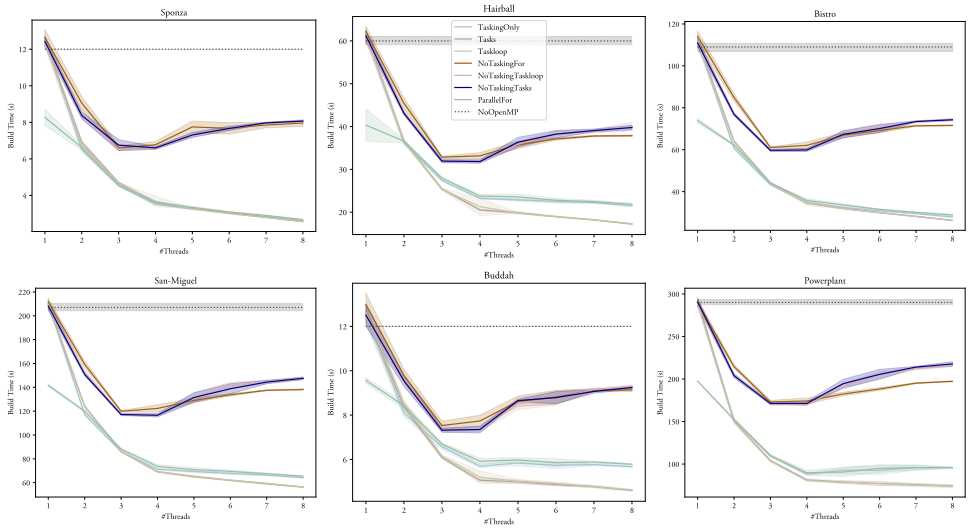


Figure 7: Visualization of how each variant scales with more available threads in the `clang` implementation.

6 DISCUSSION

While figures 6 and 7 demonstrate a notable improvement, it is also evident that the scaling is logarithmic: Each additional thread is able to increase the performance, but each subsequent gain is always diminished. In fact, as can be seen in figure 8, on a heavily multi-threaded machine scaling almost completely stops between 10 and 20 threads. This can be explained by viewing this type of BVH construction as a divide-and-conquer problem: At first, each additional thread greatly reduces the amount of necessary work, but eventually each task becomes too small to benefit from being processed in parallel, thus stopping the scaling.

Furthermore, depending on how the SBVH algorithm is implemented, some synchronization, or critical regions may be necessary. As an example, in our implementation, one such region is used to allocate indices for each of the BVH nodes. Thus, one side effect of the parallelization is that the ordering of the nodes is no longer guaranteed to be deterministic. This is particularly noticeable for the *Tasks* and *Taskloop* variants that may interleave axis searches between the main build tasks. While subtle, this effect is evident in figure 9 where it manifests as a minor increase in the average and variation of the rendering time due to cache-misses from the increased memory fragmentation of the BVH nodes.

Further, it appears that there is a moderate gain from only parallelizing the object and spatial split axis searches, with a minor lead for the *NoTaskingFor* variant, which is likely explained by the `parallel for` directive being more mature and that it has less overhead than a task queue implementation. Thus, this method may be beneficial if there is a strict requirement on a deterministic BVH hierarchy. As expected, none of these approaches scale beyond three threads, and should in fact be locked to that level, as the more threads cause a significant performance overhead.

Additionally, in figure 7, we can see that when using the `clang` compiler, it works well to create nested parallel regions, i.e., when `tasks` and `parallel for` are used inside one-another, as is done in the *ParallelFor* variant. In contrast, the `gcc` implementations in figure 6 experience dramatic regressions by several times the baseline for this variant. This is most likely a consequence of the thread-cache not being used for nested parallel regions[1]. Thus, given that the performance is in-line with the *Tasks* and *Taskloop* variants, it may be prudent to avoid nested regions, unless the targeted compiler is known beforehand.

Finally, note that using OpenMP® is not strictly beneficial: If the code is serialized, i.e., only a single thread is being used, effectively all variants have a small but noteworthy penalty to the construction times.

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108494

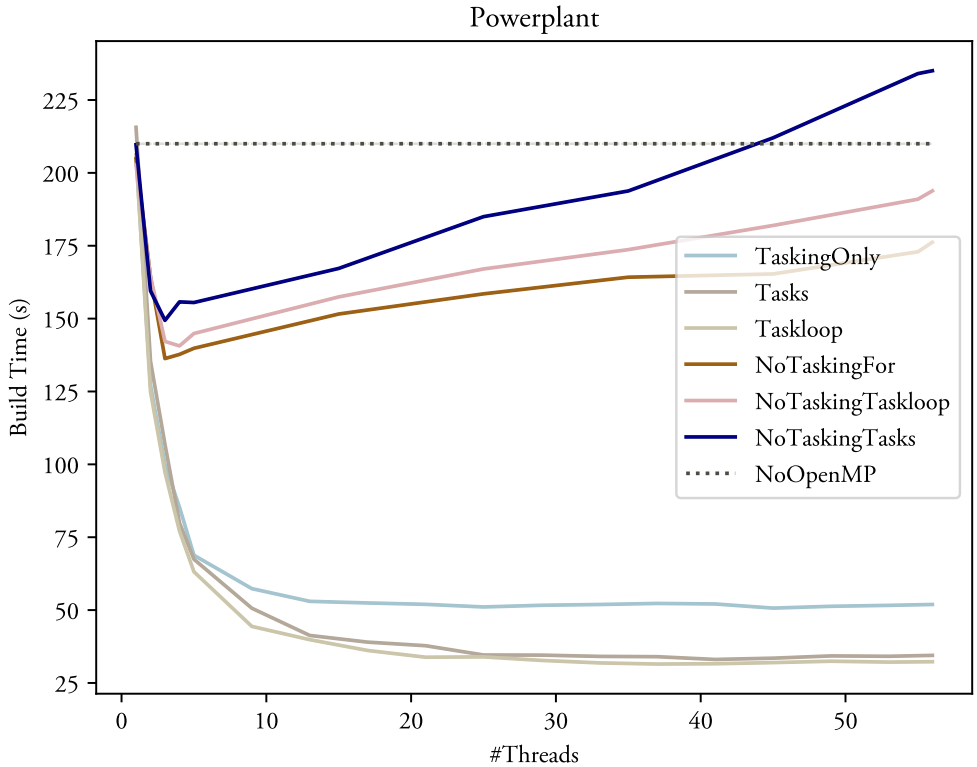


Figure 8: Scaling results for the Powerplant scene on a heavily multithreaded machine (Intel(R) Xeon(R) w7-3465X).

7 FUTURE WORK

Implementation-wise, there are a number of things that could be improved: Currently, the additional tasks for the axis searches are beneficial, particularly when each split contains a lot of primitives. However, smaller tasks are typically less useful, but OpenMP® also has support for conditionally merging or spawning tasks. Thus, finding an appropriate metric that can be used for tuning the task creation process may be a beneficial endeavor. Moreover, as seen in figure 10, it should be possible to restructure the splitting task itself to expose more opportunities for parallelism and thus improve the performance even further. Additionally, this work only considered binary BVHs, but research is currently being done on hierarchies with higher branching factors. While building such structures is more complicated, the additional branching may provide even more opportunities to parallelize the construction.

As noted in section 3.2, modern implementations of OpenMP® and OpenACC have support for offloading computations to coprocessors using e.g., the `target` directive. This was not investigated as a part of this work due to the additional complexity of mapping the input data-structures to the devices. Furthermore, as can be seen in figure 8, this particular algorithm does not appear to scale beyond 30 or 40 threads, thus it is questionable whether it would benefit from a massively parallel architecture.

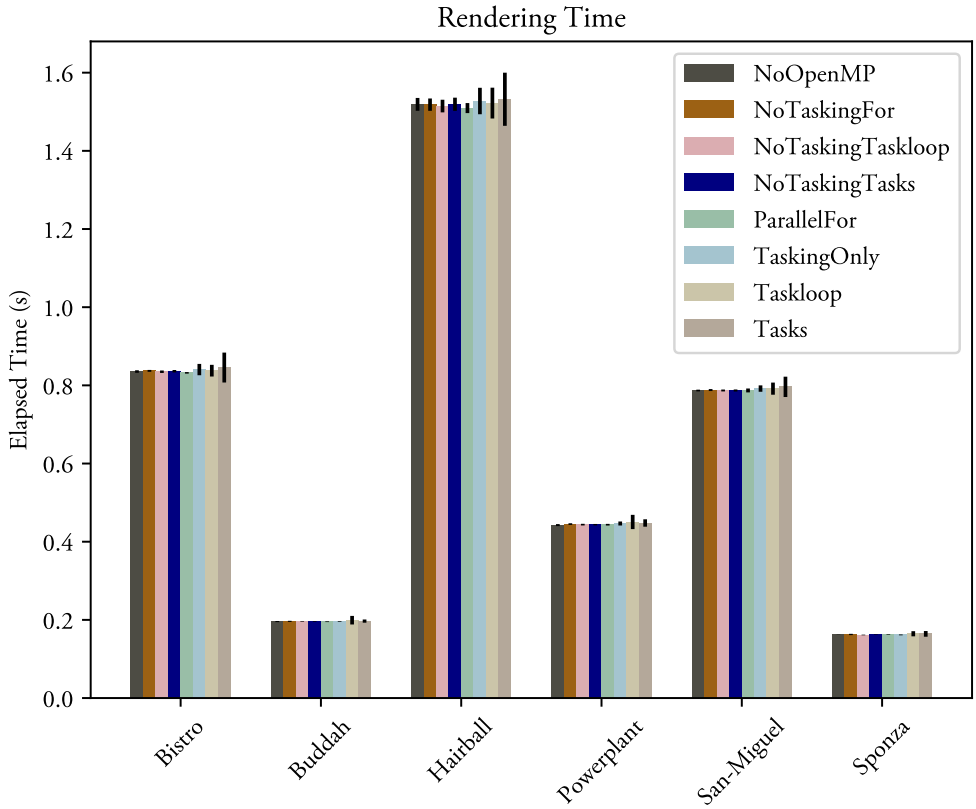


Figure 9: Rendering time using the constructed BVH by each of the `OpenMP` constructs using a `OpenCL` based ray-tracer.

8 CONCLUSIONS

OpenMP® is a very convenient way to drastically reduce the amount of necessary code required to implement many complex algorithms, such as the construction of bounding volume hierarchies (BVHs). In this paper we have both devised a new way of further parallelizing the splitting tasks of the so-called Spatial Split BVH algorithm, and tested numerous ways of applying OpenMP® on it. Thus showing that OpenMP® tasking can be effectively leveraged to keep the construction algorithms simple while still improving the build time by up to 5 times on a modern 8-core consumer workstation.

ACKNOWLEDGEMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We would also like to thank Arm Sweden AB for letting Gustaf pursue a PhD as one of their employees. Additionally, we would like to thank Rikard Olajos and Simone Pellegrini for their valuable input during this work.

Finally, we would also like to thank the authors of the models used in this work: San-Miguel (Guillermo M. Leal Llaguno), Bistro (Amazon Lumberyard), Powerplant (University of North Carolina), Hairball (NVIDIA Research), Sponza (Crytek), and Buddha (Stanford).

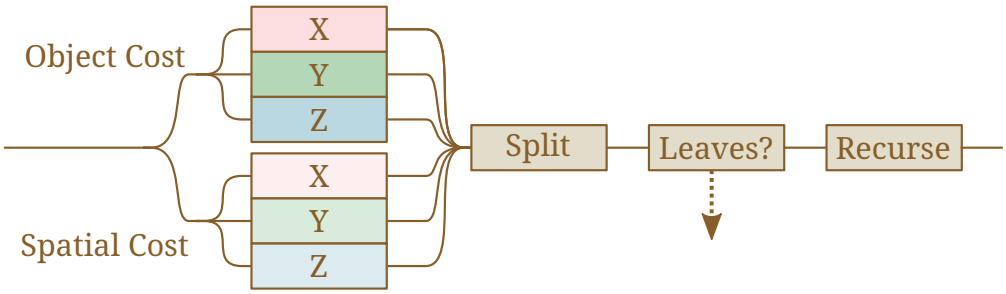


Figure 10: A potentially improved SBVH construction task: The object and spatial cost evaluation are theoretically independent and may thus run in parallel. Further, by determining if a child node would become a leaf before recursing can greatly reduce the number of necessary tasks.

REFERENCES

Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 145–149, New York, NY, USA. Association for Computing Machinery.

Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418.

Benthin, C., Meister, D., Barczak, J., Mehalwal, R., Tsakok, J., and Kensler, A. (2024). H-ploc: Hierarchical parallel locally-ordered clustering for bounding volume hierarchy construction. *Proc. ACM Comput. Graph. Interact. Tech.*, 7(3).

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Fowler, C., Doyle, M. J., and Manzke, M. (2014). Adaptive bvh: an evaluation of an efficient shared data structure for interactive simulation. In *Proceedings of the 30th Spring Conference on Computer Graphics*, SCCG '14, page 37–45, New York, NY, USA. Association for Computing Machinery.

Fuetterling, V., Lojewski, C., Pfreundt, F.-J., and Ebert, A. (2016). Parallel spatial splits in bounding volume hierarchies. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '16, page 21–30, Goslar, DEU. Eurographics Association.

Ganestam, P., Barringer, R., Doggett, M., and Akenine-Möller, T. (2015). Bonsai: Rapid bounding volume hierarchy generation using mini trees. *Journal of Computer Graphics Techniques (JCGT)*, 4(3):23–42.

Ganestam, P. and Doggett, M. (2016). Sah guided spatial split partitioning for fast bvh construction. *Computer Graphics Forum*, 35(2):285–293.

Goldsmith, J. and Salmon, J. (1987). Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20.

Karras, T. (2012). Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, page 33–37, Goslar, DEU. Eurographics Association.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384.

MacDonald, J. D. and Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166.

Meagher, D. (1980). Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Technical report, IPL: Image Processing Laboratory, Electrical and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, New York 12181.

Meister, D. and Bittner, J. (2018). Parallel reinsertion for bounding volume hierarchy optimization. *Computer Graphics Forum*, 37(2):463–473.

Meister, D., Ogaki, S., Benthin, C., Doyle, M. J., Guthe, M., and Bittner, J. (2021). A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum*.

Pantaleoni, J. and Luebke, D. (2010). Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, page 87–95, Goslar, DEU. Eurographics Association.

Pharr, M. (2018). Guest editor's introduction: Special issue on production rendering. *ACM Trans. Graph.*, 37(3).

Stich, M., Friedrich, H., and Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 7–13, New York, NY, USA. Association for Computing Machinery.

Usha, R., Pandey, P., and Mangala, N. (2020). A comprehensive comparison and analysis of openacc and openmp 4.5 for nvidia gpus. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6.

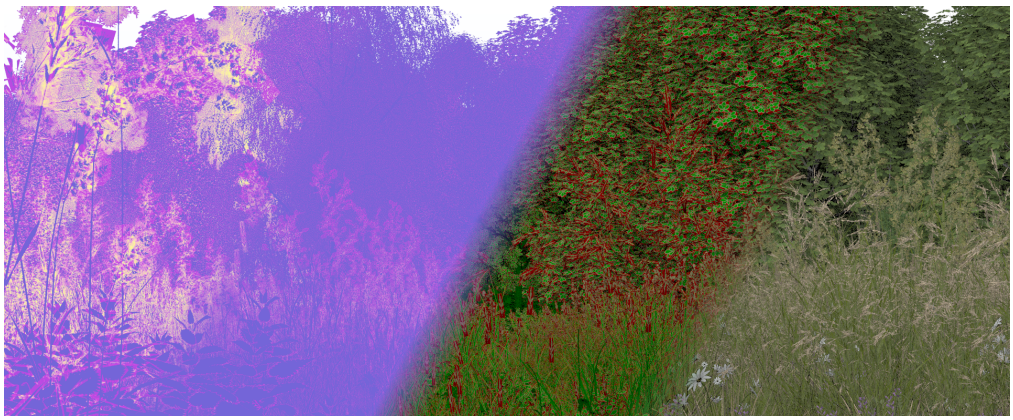
Wald, I. (2007). On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40.

Wald, I. (2012). Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):47–57.

Wald, I., Woop, S., Benthin, C., Johnson, G. S., and Ernst, M. (2014). Embree: a kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 33(4).

Paper III

Succinct Opacity Micromaps



Summary

Ray-tracing typically works well with fully and partially-transparent geometry as each ray intuitively passes through each of the objects that it needs to consider for correct color-blending. The current hardware accelerated methods struggle with this however, as expensive user-defined shaders must be called in one of the inner-most loops to achieve this effect. To combat this, *opacity micromaps* were introduced to quickly find out if these shaders should be called or not.

In some cases however, micromaps themselves can get unduly large, consequently, in this paper we present a novel method for considerably reducing their size.

III

Related conference posters in Appendix B and C.

Succinct Opacity Micromaps

GUSTAF WALDEMARSON, Dept. of Computer Science, Lund University, Sweden and Arm Ltd, Sweden
MICHAEL DOGGETT, Dept. of Computer Science, Lund University, Sweden

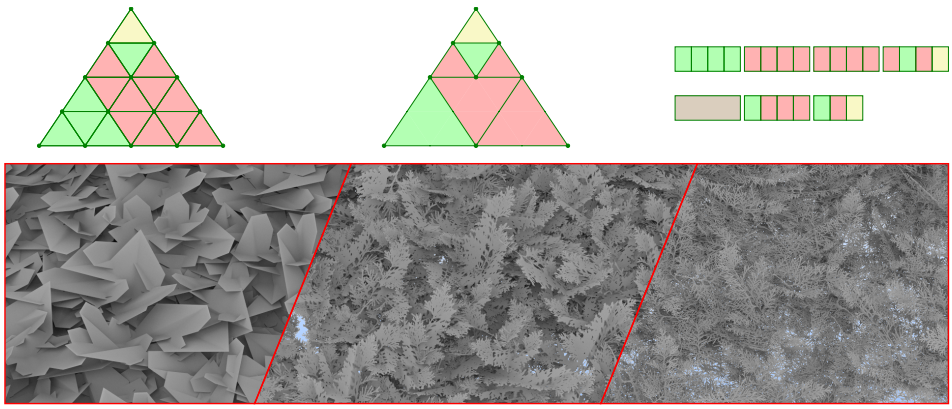


Fig. 1. Depiction of how merging adjacent micromap values into a 4-way tree structure allows it to be compressed to around 75 % of the original size with an example rendering from the New Sponza [2022] scene showing the impact of increasing subdivision levels.

Alpha masked geometry such as foliage has long been one of the trickier things to render efficiently, both for rasterization based approaches and for hardware accelerated ray-tracing. Recently, a new type of primitive was introduced to the Vulkan® and DirectX® ray-tracing APIs that promises to alleviate this issue: Opacity Micromaps, a structure that uses a bit of extra memory as hints to the pipeline when it should *actually* call the AnyHit-shader. In this paper, we extend this primitive with a novel compression method that uses the concept of succinct 4-way trees to reduce the memory footprint by up to 110 times, including an algorithm for looking up micromap values directly from this compressed form. Further, we perform a comprehensive analysis of the generated micromaps to demonstrate their performance in terms of both memory footprint and frame render time compared to a number of similar structures. Finally, we highlight some aspects of the extension that developers and artists should be aware of to make the most out of it.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; *Mesh models*; *Image compression*.

Additional Key Words and Phrases: Ray Tracing, Compression, Opacity Micromaps

ACM Reference Format:
Gustaf Waldemarson and Michael Doggett. 2024. Succinct Opacity Micromaps. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3, Article 45 (July 2024), 18 pages. https://doi.org/10.1145/3675385

Authors' addresses: Gustaf Waldemarson, gustaf.waldemarson@cs.lth.se, Dept. of Computer Science, Lund University, Lund, Sweden and Arm Ltd, Lund, Sweden; Michael Doggett, michael.doggett@cs.lth.se, Dept. of Computer Science, Lund University, Lund, Sweden.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2577-6193/2024/7-ART45
https://doi.org/10.1145/3675385

1 INTRODUCTION

Transparency has always been a challenging effect to apply for computer graphics systems. Particularly for rasterization based methods that frequently have to resort to various tricks to ensure that the effect looks correct [McGuire and Mara 2017]. In contrast, the linear propagation of rays simplifies this matter for ray-tracing methods, typically removing the need for these tricks. Thus, with the advancement in hardware accelerated ray-tracing methods, the hope has been that transparency effects would become much more prevalent. This has not materialized in practice, in part due to the remaining dependence on the rasterization pipeline in many frameworks, but also due to the so called `AnyHit`-shader [Werness 2023] that must be invoked to correctly apply most transparency effects. As this shader is situated in one of the innermost loops of the ray-tracing pipeline it has ended up as a major bottleneck for these effects.

Recently, there has been an effort to improve this matter for one particular transparency effect: Alpha masked geometry, which is frequently used on foliage, leaves and branches in many scenes. This is done by introducing a new kind of abstraction known as an *opacity micromap* that encodes a limited amount of transparency information on a subsection of each triangle and allows this information to be quickly accessed with the ray-triangle intersection data, without having to load any other metadata. And most crucially, without having to call the `AnyHit`-shader during the traversal or even interrupt the hardware traversal [Sjöholm 2022], thus promising an improved ray-tracing performance for these effects.

2 RELATED WORK

The concept of micromaps were first presented by [Gruen et al. 2020]: A relatively simple format that divided a triangle into a regular grid with a simple indexing algorithm. This basic concept has remained in the current Vulkan® and DirectX® extension, but the subdivision scheme has been changed as shown in figure 2 and discussed further in Section 4.3.

Further, [Fenney and Ozkan 2023] were first to present a scheme for compressing a two-dimensional single channel opacity map to the same states as the micromap presented by Gruen et al.. This scheme compresses the maps very well (down to between 50 % to 25 % of the original size), but are fundamentally different from the final micromaps in use by Vulkan® and DirectX®. A direct comparison to our work is difficult to accomplish for two reasons: (1) Their methods are not directly available in any Graphics API, and (2) the encoding scheme uses assets with quads in a way that is not widely used in practice. Interestingly, Fenney and Ozkan also mention investigating an explicit 3-level quad-tree method as well as a `wavelet mod 3` scheme. However, no details regarding this investigation was included in their work.

Coincidentally, work related to compressing displacement data into micromaps are covered by [Maggiordomo et al. 2023]. Notably, this type of micromap is primarily used to reduce the footprint of displacement data rather than to provide any frame-time improvement. However, as this work is not related to *opacity* micromaps, it will not be covered by this paper.



Fig. 2. A comparison of the subdivision schemes used by Gruen et al. (left), Vulkan® and DirectX® (right) at equivalent levels (i.e., $N = 2, 4$ and $n = 1, 2$ respectively). Note in particular the indexing order and the rounding pattern at the edges and corners, here shown with red arrows.

3 BACKGROUND

This section provides a general background to the technologies used to develop our algorithms.

3.1 Micromaps

In brief, a micromap is simply a linear array of values mapped into fixed sub-areas of a triangle specified by a space-filling curve as shown in figures 2 and 3. To date, there are only two kinds of *official* micromaps:

- Opacity Micromaps (`OMM`), and
- Displacement Micromaps (`DMM`).

Of these, only the opacity micromaps is currently available as a generally available Vulkan® and DirectX® extension [Werness 2022]. Further, an opacity micromaps can only contain a very specific set of opacity values depending on whether it is operating in the so-called `2-state` or `4-state` *mode*:

2-State		4-State	
		`0b00`	Fully Transparent
`0b0`	Fully Transparent	`0b01`	Fully Opaque
`0b1`	Fully Opaque	`0b10`	Unknown Transparent
		`0b11`	Unknown Opaque

As these values only occupy either 1 or 2 bits each, the opacity micromap is typically handled as a type of bit-vector. Functionally, the values are mostly self-explanatory:

- Fully transparent and opaque means that that particular sub-triangle is either completely opaque or transparent.
- Unknown values should look up the actual opacity values using some other method, by e.g., looking it up in an alpha texture. Additionally, these values can be converted to an equivalent `2-state` value, e.g., when it is undesirable to perform the alpha texture lookup, such as for shadow-rays in the ray-tracing pipeline.

Thus, this extension is primarily aimed at improving the rendering performance of alpha mapped geometry, such as foliage for the DirectX® and Vulkan® ray-tracing and ray-query extensions by avoiding most, if not all, `AnyHit` calls or returns to the calling shader. A case where ray-tracing has long promised to excel over rasterization based methods, but failed in practice, primarily due to having to call the `AnyHit`-shader inside the ray-tracing traversal pipeline.

Opacity micromaps are intended to solve this: Provide a relatively small amount of extra data with hints to the ray-tracing pipeline when to avoid calling the `AnyHit`-shader, and consequently

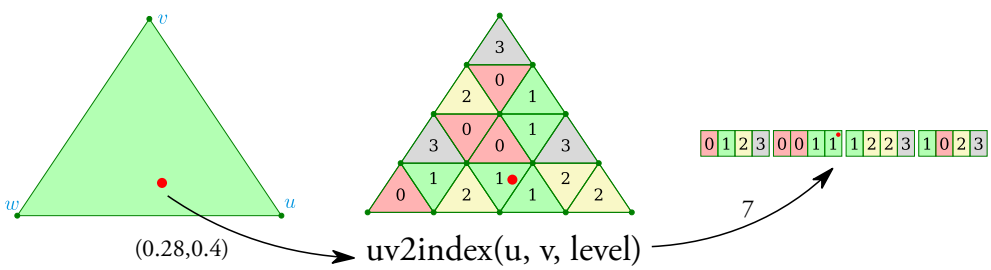


Fig. 3. Description of the barycentric coordinate to opacity micromap indexing process.

reduce the overall texture and memory bandwidth. However, these optimizations are entirely in the hands of the users: It is up to them to generate appropriate micromaps and apply them correctly to see any appreciable improvements.

3.1.1 Highest Useful Subdivision Level. In `4-state` mode, the subdivision level is typically only another tool to dial in performance, effectively trading runtime at the expense of memory. In `2-state` mode, the triangle shape and subdivision level directly influence the final geometry as there are no *unknown* values, and consequently no way of calling an `AnyHit`-shader. This shape may be distinctive, especially compared to low resolution alpha texels, as seen in figures 1 and 6. Thus, if the intent is to conservatively recreate the alpha texture with micromaps, it is useful to estimate the maximum useful subdivision level n by computing when the texture coordinate length of a subtriangle along the longest edge e is less than one pixel, i.e., when the subtriangles are smaller than the pixels. In other words:

$$\frac{\max(|e_0|, |e_1|, |e_2|)}{2^n} \geq 1 \Rightarrow \log_2 \max(|e_0|, |e_1|, |e_2|) \leq n \tag{1}$$

This is valid regardless of any minification or magnification issues introduced by camera motion as long as the texture coordinates are static. However, this becomes more complicated for primitives with coordinate transforms, but if it is possible to estimate the maximum deformation, the same approach can still be used.

3.1.2 Special Indices. In Vulkan®, Opacity micromaps are applied on triangles by providing an array of so-called triangle indices at the creation of a bottom level acceleration structure (`BLAS`). However, it is relatively common for micromaps to contain only a single value, which may be wasteful, especially at high subdivision levels. To alleviate this, Vulkan® provides a set of special index values to map directly to these cases:

`-1`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_TRANSPARENT_EXT`
`-2`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_OPAQUE_EXT`
`-3`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_UNKNOWN_TRANSPARENT_EXT`
`-4`	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_UNKNOWN_OPAQUE_EXT`

These values can be used in some cases to reduce bandwidth or otherwise simplify the micromap processing.

3.2 Succinct Data Structures

One type of data structure that is not widely used throughout the field of computer graphics is the so-called succinct data structure, originally introduced by [Jacobson 1989]. These types of data structure are so called because of their small memory footprint: Typically only a constant factor from the information-theoretical minimum.

As a concrete example, consider all binary trees with n nodes. There are only a finite number of these trees, given by the Catalan number C_n [2023, A000108]. As such, there are approximately 4^n distinct trees for large values of n. Consequently, it is possible to enumerate them and encode the trees using only $\log_2(4^n) = 2n$ bits.

Depending on the desired properties, a number of different encoding schemes could be used: The simplest of which is to perform a depth-first search over the tree, setting a bit to 1 if the node is an internal node and 0 otherwise. Thus representing the entire tree and allowing us to readily count the number of internal and leaf nodes in the tree using population counts on the bits themselves. Further, these counts can be used as indices for retrieving data stored in the abstract tree nodes.

4 ALGORITHMS

In this section we will present the primary algorithmic contributions of this paper.

4.1 Succinct Tree Encoding

There are quite a few ways of encoding an existing micromap as a succinct tree, but one of the simplest can be expressed as follows:

- Construct the *perfect* 4-way tree from the flat micromap from the bottom up by merging adjacent nodes with identical values (algorithm 1), then
- encode the resulting *perfect* tree as a *succinct* tree (algorithm 2).

An example of this is illustrated in figure 4.

```
Data: Micromap map
Result: Perfect-Tree
for level in map.level − 1 to 0 do
    foreach subtriangle at level do
        if all child nodes contain the same micromap value then
            Convert this node to a leaf node
        else
            Mark the node as an internal node
        end
    end
end
```

Algorithm 1: Algorithm used to construct a perfect tree from a flat micromap in a bottom-up fashion.

```
Data: Perfect-Tree
Result: Bit-vectors Succinct-Tree and Data
stack ← root node of Perfect Tree;
while stack is not empty do
    node ← stack.pop();
    if node is an internal node then
        Succinct-Tree.append(1);
        add all child nodes to stack;
    else
        Succinct-Tree.append(0);
        Data.append(node.value);
    end
end
```

Algorithm 2: Algorithm used to encode a perfect tree into its succinct representation using two separate bit-vectors *Succinct-Tree* and *Data*. Note that we represent internal nodes as 1 and leaf nodes as 0.

4.2 Succinct Tree Decoding

Given an encoded tree, it is straightforward to convert this back to a non-encoded tree, and by extension, the original micromap. It is however also possible to traverse this bit-vector to directly

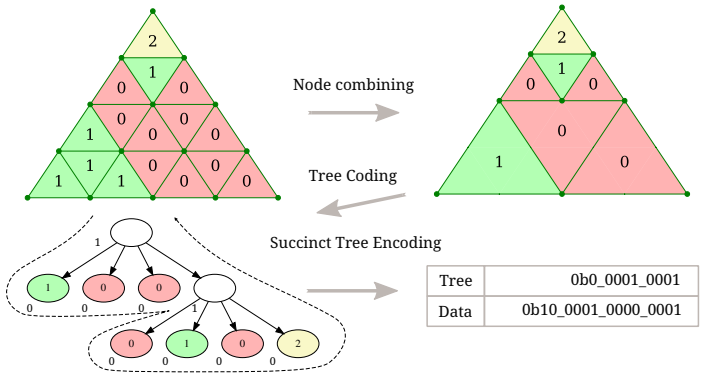


Fig. 4. A simple, but illustrative example on how to encode a flat `4-state` micromap with 2 subdivision levels as a succinct tree.

access the underlying micromap value. This can be done with algorithm 3. Note that the algorithm is strongly tied to the bit-representation of the tree. If the representation is changed, the algorithm must change accordingly.

Data: Succinct-Tree, Data
Result: Micromap value
Fn `tree-lookup`(u, v)**:**
  $t = 0$; $d = 0$;
  while *true* **do**
    if *t is internal node in tree* **then**
      $t = t + 1$;
      $child$ = `step`(u, v);
      t, d = `bitscan`$(tree, data, child, t, d)$;
    else
      return data$[d]$;
    end
  end
End Fn

Fn `bitscan`$(tree, data, child, t, d)$**:**
  while $t < tree.length$ *and* $child > 0$ **do**
    if *tree[t] is internal* **then**
      $child = child + 4$;
    else
      $d = d + 1$;
    end
    $child = child - 1$;
    $t = t + 1$;
  end
  return t, d;
End Fn

Algorithm 3: Algorithm to directly look-up a micromap value from the tree representation. Note that the `step` function is described in algorithm 4 and here returns the index of the child node to visit in the range 0 to 3.

4.3 Micromap Indexing

The Vulkan® micromap extension includes an algorithm for converting barycentric coordinates to an index into the micromap structure, in this paper referred to as the *reference* algorithm [Werness 2022]. However, this algorithm is *arguably* not very easy to understand due to the opaque nature of the numerous bit-wise operations. To that end, we devised an arguably simpler, iterative algorithm (4), that to our knowledge, has not been published elsewhere yet. In brief, it does the following:

- Explicitly compute all barycentric coordinates for the hit point, then use these to determine which of the 4 sub-triangles (*L*eft, *M*iddle, *R*ight or *T*op) is hit by the intersection.

- Increment the index and recompute the barycentric coordinates according to the intersected subtriangle.
- Recurse until the desired subdivison level is reached.

Note that the majority of the conditions and their ordering is done to correspond exactly with the *reference* algorithm, as it has some peculiar rounding behavior in the edge and corner cases, as shown in figure 2. Moreover, a detailed description of this algorithm and how the expressions were derived can be found in Appendix A. Additionally, this new algorithm has been proven to be equivalent to the *reference* up to subdivision level 15 using the ACL2 theorem prover [Kaufmann and Moore 2004] using rational numbers. It is not clear whether the discrepancy at level 16 is an error in our algorithm, a failure in the *reference* due to an overflow condition or some kind of floating point issue. However, it is obvious that the *reference* algorithm **must** be rewritten to handle more than 16 subdivision levels as the final interleaving step cannot handle more than 16-bit values. Although, a micromap of that size (4 GiB) appears to be of little practical use at this time.

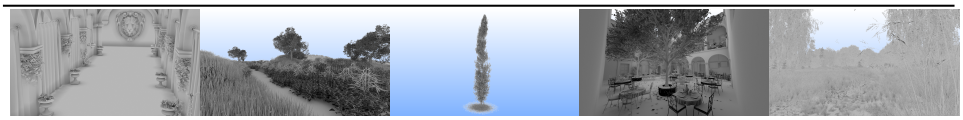
Data: Barycentric coordinates u, v, subdivision *level*
Result: Opacity micromap *index*
Fn `uv2index`$(u, v, level)$**:**
  $w = 1.0 - u - v$;
  return step$(u, v, w, 0,$ **false**, **false**$)$;
End Fn

Fn `step`$(u, v, w,$ *index, mid-flip, top-flip*$)$**:**
  if *depth = level* **then**
    return index;
  if *top-flip* **then**
    $L, M, R, T = 2, 1, 0, 3$
  else
    $L, M, R, T = 0, 1, 2, 3$
  end
  if $w > 0.5$ **then**
    return step$(2u, 2v, (w - u - v), 4 \cdot index + L,$ *mid-flip, top-flip*$)$
  else if $v \geq 0.5$ *and not* $(v = 0.5$ *and* mid-flip*)* **then**
    return step$(2u, (v - u - w), 2w, 4 \cdot index + T,$ *mid-flip,* **not** *top-flip*$)$
  else if $u \geq 0.5$ *and not* $(v = 0.5$ *and* mid-flip*)* **then**
    return step$((u - v - w), 2v, 2w, 4 \cdot index + R,$ *mid-flip, mop-flip*$)$
  else
    return step$((u + v - w), (w + u - v), (v + w - u), 4 \cdot index + M,$ **not** *mid-flip, top-flip*$)$
  end
End Fn

Algorithm 4: Algorithm used to convert a pair of barycentric coordinates to a linear index into an opacity micromap. Note in particular that the *step* function is tail-recursive, and as such can easily be changed to an iterative method.

Table 1. Description over all tested scenes with a number of relevant properties. Note that micromap features may depend on the subdivision level and thus are listed as a range.

Property	*Sponza*	*Ecosys*	*New Sponza*	*San Miguel*	*Landscape*
Instances	25	12 755	4	380 748	407 691
Meshes	25	141	4	808	370
Triangles	262 267	1 171 562	2 023 747	2 503 044	27 885 845
Textures	38	9	5	237	212
Alpha Masks	3	8	1	236	211
Opacity Micromaps	22 to 3792	24 to 128	4 to 40	1409 to 52 118	832 to 17 636
Special Indices	218 to 28 427	1080 to 8280	0 to 1 219 113	147 313 to 394 049	287 534 to 7 498 852



CryTek Sponza [2011] Ecosys [1998] New Sponza [2022] San Miguel [2010] Landscape [2016]

5 RESULTS

The evaluation of the algorithms is split into two parts: One representing the compression potential of the succinct tree encoding, the other, the frame rendertime. In both cases, the algorithms are tested on the scenes described in table 1. Opacity micromaps are generated from the original alpha masks up to subdivision level 6, all of which can be found in the supplemental material. In total, we evaluate six different methods:

Micromap	Micromaps emulated in software.
Tree	Tree encoded micromaps.
Vulkan Fast-Build (FB)	Vulkan Micromaps built with the *Fast-Build* flag.
Vulkan Fast-Trace (FT)	Vulkan Micromaps built with the *Fast-Trace* flag.
Bitmask	A pseudo-texture where every 1 or 2 bits represent the alpha value.
Texture	The original alpha texture.

Every method except the texturing approach can also run in either `2-state` or `4-state` mode as described in Section 3.1.

5.1 Compression

We evaluate each opacity method by estimating their total memory footprint. For micromap approaches, this includes the micromap data itself, the so-called triangle indices, and any metadata structure. Only the micromap data is included for the Vulkan methods however, as they may embed any additional metadata in the acceleration structure. Further, the bitmask and texture approach need to load vertex indices and texture coordinates in addition to the bitmask or texture. This data is listed in table 2 and plotted in figure 5 for increasing subdivision levels. Finally, the tree compression ratio in figure 5 is computed against the original opacity micromap data.

5.2 Runtime

The frame rendertime is evaluated by implementing each method in an `AnyHit` shader, except for the *Vulkan* methods, as they cannot be controlled in such a fashion. All methods are rendered at 1920×1080 with a refining ambient occlusion and stochastic transparency algorithm on an Nvidia RTX 3080 and 4080. Each sample time is gathered from the start- to end-of-pipe as reported by the Vulkan pipeline querying API. Note that the values presented in table 3 and figures 6, 7a and 7b represent the average over all viewpoints in a given scene weighted by the number of rendered frames. A per-pose breakdown is available in the supplemental material, however.

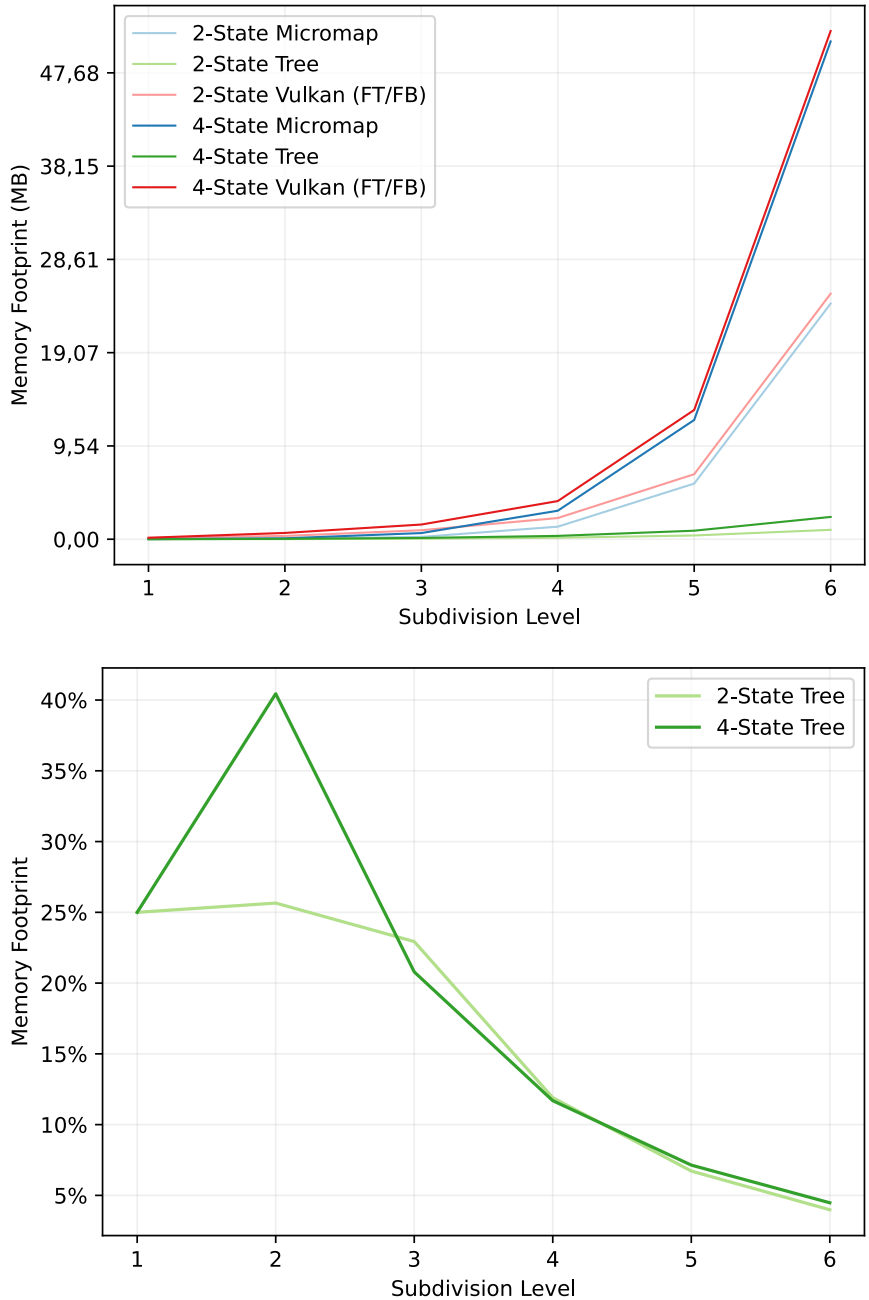


Fig. 5. Plot over the memory footprints and micromap tree compression potential for increasing subdivision levels in the San Miguel [2010] scene. The results are similar for all tested scenes and thus omitted for brevity.

Table 2. Total memory footprint in megabytes (MB) required by the various opacity methods for each scene. Note that the `fast-trace` and `fast-build` has the same footprint and that the micromap and tree method requires the same amount of metadata and that all micromap methods use subdivision level 6.

Mode	*Method*	*Sponza*	*Ecosys*	*New Sponza*	*San Miguel*	*Landscape*
2-State	*Micromap*	1,842	0,062	0,020	24,091	8,192
	Tree	**0,139**	**0,002**	**0,003**	**0,960**	**0,630**
	Vulkan (FT/FB)	1,915	0,067	0,021	25,103	8,574
	Bitmask	0,188	0,320	2,000	5,428	14,402
4-State	*Micromap*	3,703	0,125	0,039	50,896	17,223
	Tree	**0,346**	**0,004**	**0,007**	**2,279**	**1,446**
	Vulkan (FT/FB)	3,777	0,130	0,041	51,962	17,620
	Bitmask	0,375	0,640	4,000	10,852	28,801
N/A	*Texture*	1,500	2,558	16,000	43,399	115,195
N/A	*Vertex Data*	1,258	1,315	52,706	51,910	422,197
N/A	*Triangle Index*	0,133	0,082	5,167	1,923	35,083
N/A	*Micromap/Tree Metadata*	0,029	0,001	0,000	0,376	0,128

6 DISCUSSION

In this section we discuss the implications of our findings and attempt to interpret them.

6.1 Frame-time

Performance-wise, the first notable result is that there appear to be a small frametime improvement between a micromap created with the `fast-build` versus one with the `fast-trace` flag. However, the memory footprint is the same in both cases. Thus, it is likely that at this date the Nvidia driver only implement a single type of opacity micromap, but have a slightly more optimized access algorithm for the `fast-trace` case.

Further, even without official micromaps, we were able to detect a substantial improvement: Up to 16 % lower frame-time compared to using alpha masks directly. However, with only software emulation, we found a number of cases where using micromaps would instead increase the frame-time by up to 30 %. It seems as this is the primary case where the *RTX 40*-series significantly improve matters: Using the official micromap methods the worst recorded frame-time is only increased by 2 % and the best recorded one reduces it by 29 %. However, we also want to highlight the results from figure 6, which clearly shows a case where *not* using micromaps seems preferable if an *RTX 40*-series card is not available. However, we again want to note that the difference between all methods is very small: Only around 0.15 ms.

Moreover, there appears to be only an extremely small improvement to using a bitmask instead of a real texture: Presumably the bandwidth cost from loading the vertex data offsets most of the potential gains. Then again, it would be interesting to see if compressing such a map similar to the approach suggested by [Fenney and Ozkan 2023] would improve matters.

Lastly, accessing values from the tree compression is comparable to the other methods for micromaps of subdivision levels 3 or 4 as seen in figures 7a and 7b. This also appears to be the more reasonable sizes for the scenes tested in this work, as can be seen among the micromap samples in the supplemental material. At higher levels however, it is arguably too slow to be of practical use, at least in its current form. This is likely caused by the `bitscan` function in algorithm 3: Each time the right-most child is accessed, all intervening subtrees for all other children must be scanned, the

number of which roughly quadruples for each subdivision level. However, improvements to these types of data structures that use a bit more memory (about 26.5 % more) to ensure that the data can still be accessed in an efficient manner already exist [Gog et al. 2014]. Investigating these options and finding a structure with a better trade-off seems like a worthwhile endeavor, and even if these structures cannot improve the access times, the tree structure would still be useful as a storage format, particularly for high subdivision levels.

6.2 Compression

The tree method presented in this work is able to compress the opacity micromap data extremely well: Typically down to between 45 and 15 % of the original size, but in some extreme cases, such as for the New Sponza scene [2022] at 12 subdivision levels, to less than 1 % the original size, or by 110 times. A trend we expect to continue at higher subdivision levels.

However, while the compression is good in terms of memory footprint, the same is not necessarily true for the memory *bandwidth*. Similar to other compression methods such as Huffman coding [Huffman 1952], a potentially large portion of memory may need to be decoded before the sought value is found, as is visualized in figure 8. This is also a notable deviation from the recommendations for texture compression given by [Beers et al. 1996] and the most important aspect that needs to be improved with a different succinct structure.

Table 3. Frametimes in milliseconds (ms) for all methods averaged over all camera poses in a scene. All micromap based methods are using subdivision level 6. See figures 7a and 7b for a per-level breakdown.

Platform	*Mode*	*Method*	*Sponza*	*Ecosys*	*New Sponza*	*San Miguel*	*Landscape*
RTX 4080	*2-State*	*Micromap*	4.56875	7.46140	2.11000	7.51584	16.11604
		Tree	**9.75386**	**13.87786**	**33.27646**	**19.16756**	**36.50315**
		Vulkan (FT)	4.50288	7.30962	1.34046	6.98950	13.86996
		Vulkan (FB)	4.57210	7.43183	1.44883	6.95655	14.17963
		Bitmask	4.58546	8.12129	3.52295	7.70161	17.81338
	4-State	*Micromap*	4.64359	7.50737	3.30602	7.86051	16.91202
		Tree	**16.17243**	**15.64397**	**68.65045**	**29.81126**	**56.89248**
		Vulkan (FT)	4.59129	7.35208	2.67456	7.28262	14.53477
		Vulkan (FB)	4.65063	7.47321	2.87122	7.43438	15.00084
		Bitmask	4.65097	8.22064	3.83369	7.93755	18.54452
	N/A	*Texture*	4.64821	8.20801	3.68133	7.96372	18.47607
RTX 3080	*2-State*	*Micromap*	7.63283	13.34334	3.84370	15.00808	30.71679
		Tree	**15.78279**	**23.38782**	**52.33564**	**34.02836**	**62.33444**
		Vulkan (FT)	7.64356	13.63098	3.64963	14.82397	29.88393
		Vulkan (FB)	7.68931	13.94923	3.95979	15.07067	31.06890
		Bitmask	7.70873	14.34579	7.18576	15.47086	34.17612
	4-State	*Micromap*	7.73455	13.37131	6.60013	15.63869	32.02999
		Tree	**25.54302**	**26.25542**	**108.61797**	**50.41895**	**94.30972**
		Vulkan (FT)	7.80105	13.69033	6.38412	15.36736	31.07889
		Vulkan (FB)	7.84215	14.01853	6.80095	15.72004	32.50680
		Bitmask	7.74633	14.14043	6.98767	15.68434	33.99675
	N/A	*Texture*	7.83878	14.23929	7.44211	15.82377	34.42618

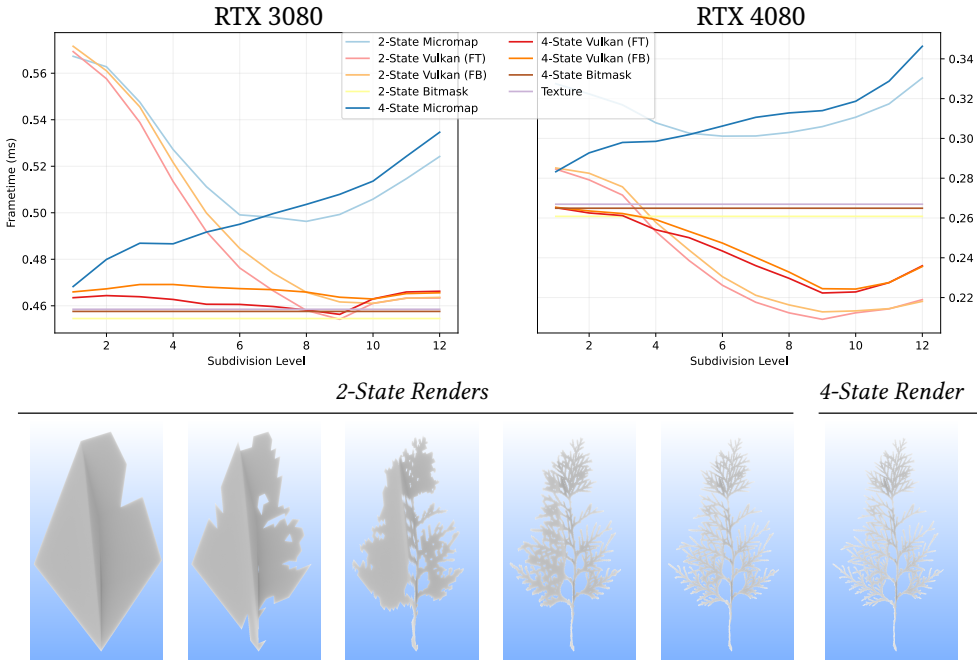


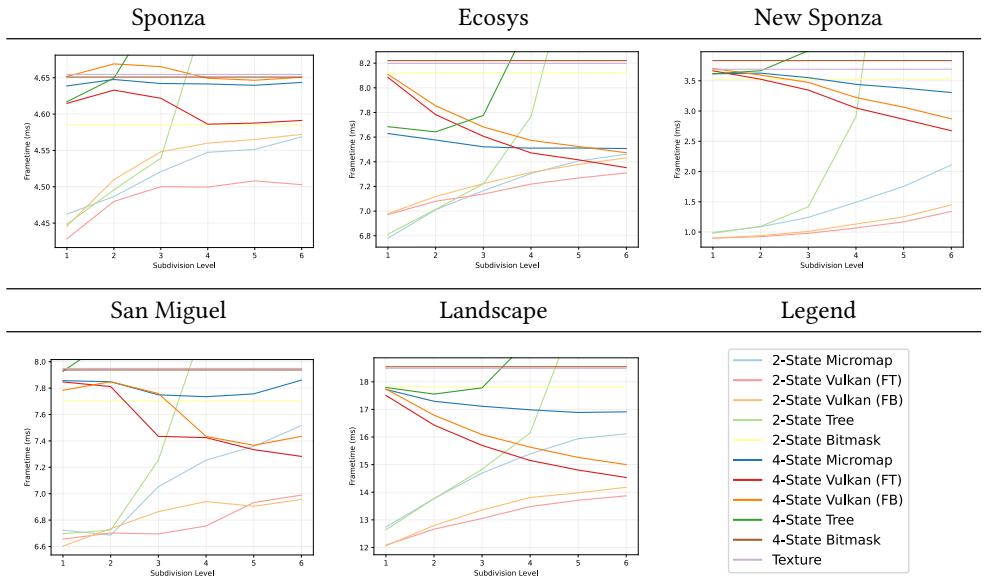
Fig. 6. Frametime plots for subdivision levels 1 to 12 when rendering a single twig from the New Sponza scene [2022]. Note that each `4-state` methods yield the same render, whereas the `2-state` methods is different for each one, with only subdivision 1, 3, 5, 7 and 9 shown here. Be aware of the Y-axis scale: The difference between each method is very small practice.

6.2.1 Degenerate Cases. The tree compression algorithm seems to be good in the presented scenes, but there are a number of cases that cannot be compressed with this method. E.g., a micromap that *always* alternates between the micromap states will create a succinct tree that requires more memory than the original micromap, as depicted in figure 9. Such cases could be handled by extending the encoding to sub-trees rather than just leaf-nodes but as these cases are exceedingly rare in real content this may not be necessary in practice.

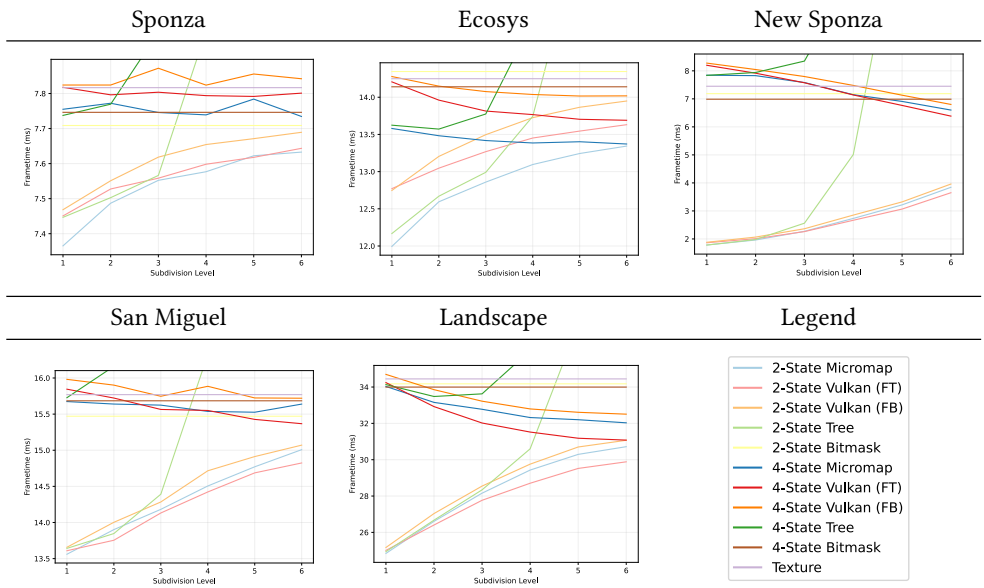
6.3 Comparing Micromaps to Textures

At first glance micromaps may appear to simply be a specialized kind of image texture. However, this is arguably not true, especially not for micromaps generated from alpha mapped foliage. Those kinds of micromaps are strongly tied to both the texture and the (uv) coordinates they were generated from. As such, they are more similar to the textures originally envisioned by [Catmull 1974]. Further, a cursory analysis of our chosen scenes quickly reveal that scenes that only use a single triangle or quad to represent foliage are exceedingly rare. As an example, figure 10 depicts a texture atlas from the CryTek Sponza scene [2011] with all unique triangle texture coordinates overlaid on top of it. This is a representative view of what can happen in practice:

- In some cases, we have a well-structured grid of coordinates, in others,
- we have many overlapping and crossing edges due to the coordinates being slightly misaligned.



(a) RTX 4080



(b) RTX 3080

Fig. 7. Plots over how the average frame-time changes for increasing subdivision levels on an Nvidia RTX 3080 and 4080.

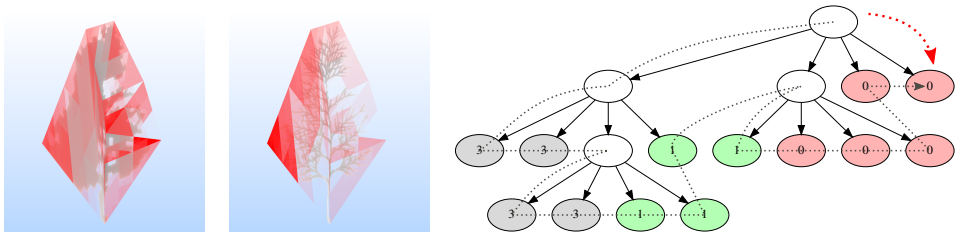


Fig. 8. Visualization of the number of bits of the micromap tree that has to be read by an incoming ray before an opacity value is found. Here shown in two scenes with `2-state` micromaps at levels 4 and 9 along with a schematic view over how these asymmetric read patterns arise when attempting to read the right-most tree nodes.

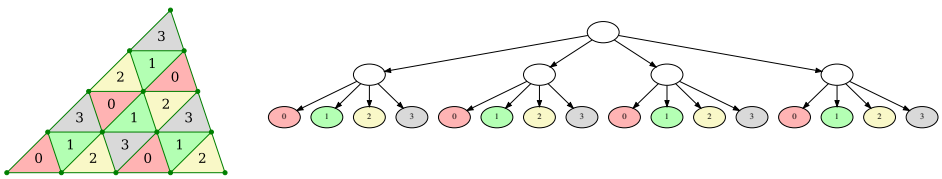


Fig. 9. Example degenerate case for the succinct tree compression: Instead of actually reducing the number of nodes, it is forced to add 5 internal ones.

In such cases, each triangle may create unique micromaps despite sharing a common texture and may consequently increase the acceleration structure bandwidth usage rather than reduce it.

Further, in more modern scenes such as in San Miguel [2010] and Landscape [2016], leaves and branches are often modeled individually and split into many segments to allow artists to give them more depth as seen in figure 11. In these cases, micromaps work well, and relatively small subdivision levels can be used as only small parts of the mesh cover partially-transparent regions. However, in theses cases it is important to use the special triangle indices (see Section 3.1.2) to avoid having to explicitly represent all triangles, most of which will be fully-opaque.

7 FUTURE WORK

The micromap extensions are currently limited in scope but thanks to the extensible nature of the modern graphics APIs, it will be straight-forward to extend them in the future. The concept itself may even extend beyond these APIs:

The indexing algorithm (4) generalizes in multiple aspects: To other dimensions, shapes and subdivision levels. Investigating this further could lead to a number of interesting applications.

The tree accessing algorithm (3) only considered software implementations. As such, investigating the costs and benefits of these algorithms when implemented in hardware remains open, and while the presented version probably does not translate well into hardware, other uses of succinct data structures may be more amenable to this.

In regard to the extension itself: The current `2-state` mode is limited to the `fully-opaque` or `fully-transparent` values. Adding one or more 1-bit modes to the opacity micromap extension that replace either of these values with the `unknown-opaque` or `unknown-transparent` could be

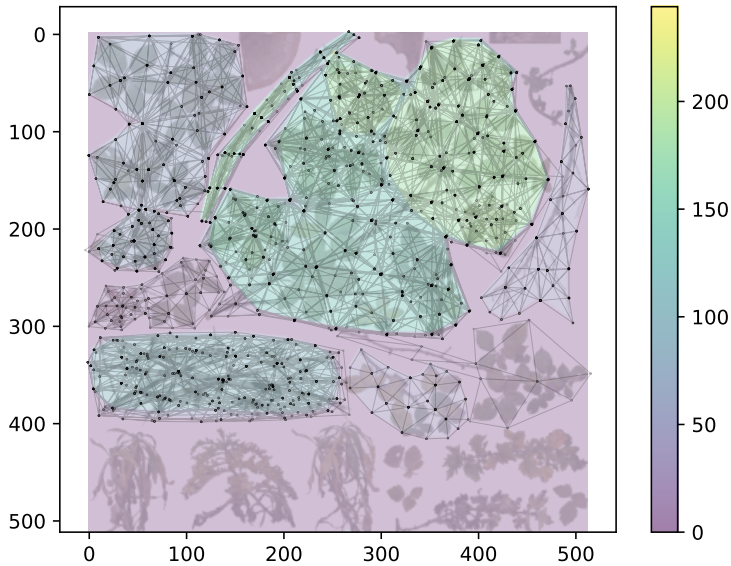


Fig. 10. A representative texture atlas of various plants from the CryTek Sponza [2011] scene with all uniquely unwrapped texture coordinates overlaid on top, along with a histogram showing how many triangles overlap each pixel.

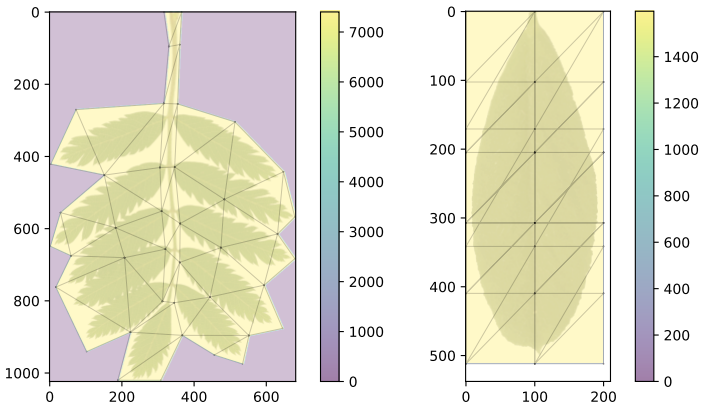


Fig. 11. Two representative textures depicting how modern scenes model leaves with a single texture and a highly tessellated grid to allow artists to give the leaves more depth.

an interesting way to further reduce the memory footprint. We plan to investigate the potential gains from such modes in the future.

Opacity micromaps is not the only available type: As mentioned in Section 3, *displacement* micromaps are already available on some hardware and other types are under development [Bickford 2023]. Some aspects of the presented tree encoding seem to extend to these types but more research

is needed in this regard. Further, the algorithms presented in this paper only considered *lossless* encoding. Allowing lossy encoding of micromap values would open up many new avenues of research.

For the sake of brevity, the subject of *constructing* micromaps were not considered in this paper. Nvidia has published a framework for constructing them either during rendering or as a part of the asset preparation pipeline [GameWorks 2022]. Further, there is ongoing work to make similar tools available in Blender [Waldemarson 2023]. However, the aspect of how to create micromaps in an efficient and continuous manner is arguably still worth investigating.

Finally, the use-case for micromaps, and particularly *opacity* micromaps, is rather limited at the time of writing. Finding new and clever use-cases for them could be very interesting.

8 CONCLUSION

In this paper we introduced several novel algorithms related to the compression of micromaps by converting them to succinct 4-way trees, reducing their memory footprint by up to 110 times in a number of representative scenes that use alpha mapped foliage extensively while remaining competitive at reasonable subdivision levels, but additional work is needed for this method to be practical in a high performance context.

Further, we have evaluated the potential performance gain from using the official Vulkan® and DirectX® opacity micromaps extension applied to alpha mapped geometry, confirming that the feature can provide an increase in frametime by at most 29 % when used in the so-called `4-state` mode. Further, we have documented several important considerations regarding the use of `2-state` micromaps that may be important for designers to be aware of. Moreover, we have introduced an alternative to the reference barycentric-to-index-algorithm that may be easier to understand.

ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We would also like to thank Arm Sweden AB for letting Gustaf pursue a PhD as one of their employees. Additionally, we would like to thank Rikard Olajos, Simone Pellegrini and Mathieu Robart for their valuable input during this work. Finally, we are very grateful for the extensive input from our reviewers that helped us identify some areas that needed a bit of extra work.

REFERENCES

Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. 1996. Rendering from compressed textures. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. Association for Computing Machinery, New York, NY, USA, 373–378. https://doi.org/10.1145/237170.237276

Neil Bickford. 2023. *NVIDIA Micromap Extensions*. Nvidia Inc. https://github.com/NBickford-NV/glTF/tree/micro-mesh

Edwin Earl Catmull. 1974. *A subdivision algorithm for computer display of curved surfaces*. Ph. D. Dissertation. The University of Utah. AAI7504786.

Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. 1998. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/280814.280898

Simon Fenney and Alper Ozkan. 2023. Compressed Opacity Maps for Ray Tracing. In *High-Performance Graphics - Symposium Papers*, Jacco Bikker and Christiaan Gribble (Eds.). The Eurographics Association, EUROGRAPHICS Association Groene Loper 3, 5612AE Eindhoven, 23–31. https://doi.org/10.2312/hpg.20231133

Morgan McGuire Frank Meinl, Marko Dabrovic. 2011. CryTek Sponza. https://www.cryengine.com/asset-db/product/crytek/sponza-sample-scene.

Nvidia GameWorks. 2022. *Opacity Micromap SDK*. Nvidia Inc. https://github.com/NVIDIAGameWorks/Opacity-MicroMap-SDK

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Experimental Algorithms*, Joachim Gudmundsson and Jyrki Katajainen (Eds.). Springer International Publishing, Cham, 326–337.

Holger Gruen, Carsten Benthin, and Sven Woop. 2020. Sub-Triangle Opacity Masks for Faster Ray Tracing of Transparent Objects. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 18 (Aug. 2020), 12 pages. https://doi.org/10.1145/3406180

David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

G. Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS '89)*. IEEE Computer Society, USA, 549–554. https://doi.org/10.1109/SFCS.1989.63533

Timm Dapper Jan-Walter Schliep, Burak Kahraman. 2016. Landscape. https://www.laubwerk.com.

M. Kaufmann and J S. Moore. 2004. The ACL2 Home Page. In *http:// www.cs.utexas.edu/ users/ moore/ acl2/*. Dept. of Computer Sciences, University of Texas at Austin, Main Building (MAI) 110 Inner Campus Drive Austin, TX 78712, 1.

Guillermo M. Leal Llaguno. 2010. San Miguel. https://www.pbrt.org/scenes-v3/.

Andrea Maggiordomo, Henry Moreton, and Marco Tarini. 2023. Micro-Mesh Construction. *ACM Trans. Graph.* 42, 4, Article 121 (July 2023), 18 pages. https://doi.org/10.1145/3592440

Morgan McGuire and Michael Mara. 2017. Phenomenological Transparency. *IEEE Transactions of Visualization and Computer Graphics* 23, 5 (May 2017), 1465–1478. https://casual-effects.com/research/McGuire2017Transparency/index.html#bibtex IEEE Transactions of Visualization and Computer Graphics.

Frank Meinl, Katica Putica, Cristiano Siqueria, Timothy Heath, Justin Prazen, Sebastian Herholz, Bruce Cherniak, and Anton Kaplanyan. 2022. Intel Sample Library. https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-processing-research/samples.html.

OEIS Foundation Inc. 2023. The On-Line Encyclopedia of Integer Sequences. Published electronically at http://oeis.org.

Juha Sjöholm. 2022. *Best Practices for Using NVIDIA RTX Ray Tracing (Updated).* Nvidia. https://developer.nvidia.com/blog/best-practices-for-using-nvidia-rtx-ray-tracing-updated/

Gustaf Waldemarson. 2023. *Handling Custom Data in glTF Files with Exporter/Importer Plugins.* The Blender Foundation, Youtube. https://youtu.be/4fBGM8qc21M?t=1783

Eric Werness. 2022. *VK_EXT_opacity_micromap.* The Khronos Group Inc. https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html

Eric Werness. 2023. *Any-Hit Shaders.* The Khronos Group Inc. https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#shaders-any-hit

A BARYCENTRIC TO MICROMAP INDEX ALGORITHM DETAILS

The algorithm described in Section 4.3 works as follows:

(1) Split the base triangle into 4 equal sized sub-triangles using the midpoint of each edge.
(2) Then, using the vertex to edge mid-point relations;

$$\begin{cases} v_0 = m_{01} + m_{02} - m_{12} \\ v_1 = m_{01} + m_{12} - m_{02} \\ v_2 = m_{02} + m_{12} - m_{01} \end{cases}$$

and the typical formula for barycentric coordinates ($P = wv_0 + uv_1 + vv_2$), it is possible to derive the following relations to update the u, v, w coordinates for each of the sub-triangles L, M, R, T:

$$L \coloneqq \begin{cases} u_L &= u - v - w \\ v_L &= 2v \\ w_L &= 2w \end{cases} \quad M \coloneqq \begin{cases} u_M &= u + v - w \\ v_M &= v + w - u \\ w_M &= u + w - v \end{cases}$$

$$R \coloneqq \begin{cases} u_R &= 2u \\ v_R &= v - u - w \\ w_R &= 2w \end{cases} \quad T \coloneqq \begin{cases} u_T &= 2u \\ v_T &= 2v \\ w_T &= w - u - v \end{cases}$$

(3) Note that the indexing and ordering of the updated coordinates are significant to handle rounding and winding changes for the top and middle triangles as the reference algorithm

rounds *globally* away from the w coordinate. Thus, the rounding direction has to be maintained as we recurse and change winding, requiring special handling for the top and middle triangles as follows:

- Each time we recurse into the top triangle, flip the local index of the left and right subtriangles.
- Each time we recurse into the middle triangle, round u and v tie-breaks to the middle rather than the right subtriangle.

See figures 12 and 2 for examples on how this works in practice.

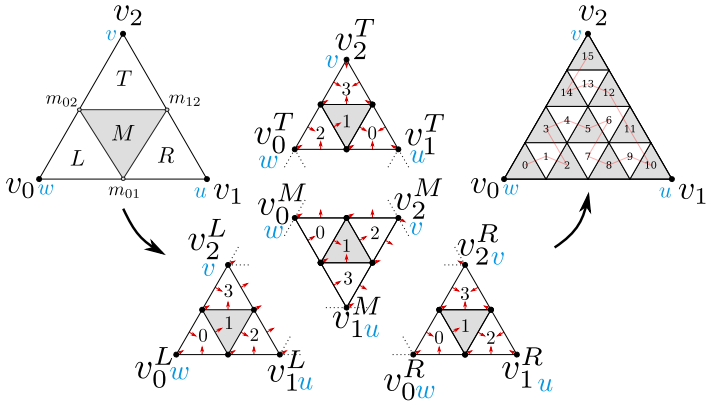


Fig. 12. Description of how a triangle that has already been subdivided once into subtriangles L, M, R, T is further subdivided. Note in particular how the uv coordinates, indices, and rounding change for each of the subtriangles.

Received 28 April 2024; revised 12 June 2024; accepted 17 June 2024

Paper IV

Color and Attribute Micromaps



Summary

Opacity micromaps were primarily intended as a quick transparency testing approach, but *displacement micromaps* provided an illustrative example of how to use the format for other purposes.

Thus, in this work micromaps are further generalized with arbitrary rendering data, creating the so-called *attribute micromap*.

This new micromap is then applied on some typical rendering attributes, such as base colors, normals, and roughness and metalness properties, for which the performance and quality trade-offs are investigated, along with future potential use-cases in an extended hardware ray-tracing pipeline.

Conference poster in Appendix F.

Color and Attribute Micromaps

Gustaf Waldemarson[1,2] and Michael Doggett[2]

[1]Arm, Sweden
[2] Department of Computer Science, Lund University

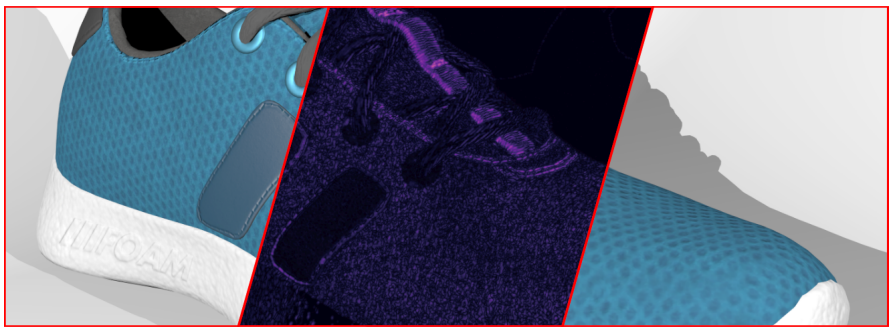


Figure 1: *Micro-vertex micromaps for color, normal and metallic roughness attributes can provide similar rendering quality at* 71 % *the memory footprint when applied to the Shoe model.*

Abstract

The hardware accelerated ray-tracing pipeline presents many opportunities to implement more advanced rendering algorithms than earlier rasterization based approaches. However, one particular aspect of the pipeline still remains a bit too costly to use in practice: The `AnyHit`*-shader. Opacity Micromaps were introduced to alleviate this issue, but the central problem of calling arbitrary shader code on an unknown number of primitives remains prohibitively expensive. Thus, in this paper we take the first step of addressing this by further extending the concept of micromaps to arbitrary attributes; thereby theoretically giving the ray-tracing pipeline access to more generalized types of data. For this work, we focus on shading attributes, such as color, normal, metalness and roughness data, showcasing how their quality changes in a micromap-based setting, and what unique aspects they bring to the ray-tracing pipeline. Finally, we investigate how this could be used in existing shaders, and to implement various fixed-function operations during ray traversal that typically require the use of* `AnyHit`*-shaders.*

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;*** *Mesh models; Image compression;*

1. Introduction

Ever since image textures were originally proposed by Catmull [Cat74] they have been used to great effect to improve the quality of rendered images. At first, they were applied directly on surfaces, but were later extended with per-primitive texture coordinates to improve its flexibility. Eventually, textures were even used to contain more general types of attributes, such as normals, physically based rendering properties such as metalness and roughness, and even transparency values.

Transparency in particular have proven especially challenging to accelerate in hardware, even for the modern ray-tracing hardware where traditional ray-tracing methods have previously worked quite well, typically due to the extra shader code that must be called in one of the innermost traversal loops. This issue was alleviated with the introduction of the so-called *micromaps* [GBW20], a fully independent structure

that encode a fixed set of transparency values to allow the user to systematically reduce the number of times this extra code have to be called, effectively harking back to Catmull's original idea of textures without texture coordinates. So far, this concept has only been applied on two types of attributes: The aforementioned transparency-values, and displacements. Thus, in this paper we investigate how to fill in the gaps between these types of micromaps, and consequently extend the concept to arbitrary attributes. At the same time, we demonstrate what kind of trade-offs are made in terms of memory footprint, frame-time, and quality, and show how these new types of micromaps could be used as operands for a possible fixed-function addition to the hardware ray-tracing pipeline.

2. Related Work

Micromaps were originally introduced by Gruen et al [GBW20] to improve ray-tracing performance for transparent objects by creating fast look-ups for triangular sub-regions that are completely opaque or transparent in an effort to reduce the number of times an `AnyHit`-shader has to be called. This work was later incorporated in the Vulkan® and DirectX® specifications under the name of *Opacity* Micromaps, although with a change to the ordering of the micro-triangles to follow a special space-filling curve, sometimes called the *Bird*-curve [Khr22].

During this time, Fenney et al were first to present a scheme for compressing a micromap-like structure to around 50 % to 25 % of the original size. However, this structure was fundamentally different from the official opacity micromap format [FO23].

Further, Waldemarson et al presented a method for interpreting the official format as a tree-based structure, thus allowing it to be heavily compressed as a succinct data structure. This approach was able to achieve a typical compression ratio of between 45 and 15 %, but in some cases to less than 1 % of the original size. As presented however, this approach was not suitable for real-time usage beyond 3 or 4 subdivision levels due to the high look-up cost [WD24].

Moreover, the concept of displacement micromaps was added to Vulkan® as a provisional extension to render a specific representation of highly compressed and dense triangle meshes by allowing each triangle to displace individual and implicitly created micro-vertices [Khr23b], the construction of which was first covered by Maggiordomo et al [MMT23], and more recently improved with differentiable rendering [DZJ*24], and skinned animations [GBK*24]. However, it would appear that direct use of this particular extension has now been deprecated, and can no longer be used [NVI25].

The concept of storing micromaps and associated metadata for general attributes such as colors or normals has been specified as a `glTF` extension [Bic23], but in this paper we describe in detail how similar micromaps can be used in a rendering engine as well as what the trade-offs are made in terms of memory footprint, frame-time, and quality.

2.1. Prior Surface Parameterizations

The type of micromaps presented here can be viewed as a form of surface parameterization and consequently have a number of features in common with similar concepts, particularly *Ptex* [BL08], *Htex* [BD22] and *mesh colors* [YKH10].

Ptex is a file- and texturing format designed for film quality rendering where each face of a quad-based control mesh receives a unique and independent texture and associated mipmap hierarchy, and whose primary use is to simplify the authoring of assets rather than the rendering itself. Micromaps are similarly assigned on a per-face basis, but only for triangle based-meshes in the real-time ray-tracing pipeline, and is currently a primarily tool-generated format used to assist with very specific shading tasks.

Htex is a per-face texturing solution similar to *Ptex*, but now extended beyond quad-meshes to arbitrary polygons using a half-edge data structure that also allows it to run efficiently on GPUs. However, the primary target for this structure is still only to simplify and optimize the asset authoring process rather than the final rendering.

Mesh colors extend the concept of vertex colors to a more general surface format, if primarily for rasterization based rendering, but with many parallels to the extended micromap concepts presented here: Both formats define implicit micro-vertices on mesh surfaces, but handle various aspects differently: Mesh colors stores values on corner-, edge-, and surface-vertices separately and with an implementation defined ordering, whereas micromaps use an explicit subdivision scheme to distribute and order all vertices. This creates some implications: Micromaps must duplicate values along edges to ensure that each face is independent and thus avoid having to load vertex attributes, but consequently ensures memory coherency for these values. In contrast, the mesh color values for the corners, edges and faces can end up scattered in different places. This also leads to different handling of discontinuities and cracks between faces of different resolution: Micromaps are guaranteed to be crack-free by using a single flag-bit per edge with a fixed decimation scheme, as long as the adjacent face has at most one higher or lower subdivision level. Mesh colors handle this by ensuring that faces with differing resolutions are exact multiples of each other, and that intervening samples from the higher resolution face are consistent with the lower resolution one, e.g., by linearly interpolating them. This does, arguably, mean that these samples are wasted, as they cannot be chosen freely to improve the surface quality. Finally, follow-up work on mesh colors have focused on making the format itself faster and more GPU friendly [Yuk17] or by implementing a version of it in hardware [MSY19].

3. Background

Conceptually, an opacity micromap is simply a linear array of 1 or 2 bit values used to encode 2 or 4 fixed opacity values on implicitly created micro-triangles as described in table 1. These values then in turn allow the ray-tracing pipeline to avoid calling the expensive `AnyHit`-shader for all but the *unknown* cases [WD24].

However, in contrast to opacity micromaps which are defined for micro-triangle *faces*, displacement micromaps operate on micro-triangle *vertices*. Thus, generalized micromaps need to be able to use either of these versions. In either case, micromaps can be viewed as an implicit triangle-mesh on top of each original triangle with distinct values associated with either the micro-triangles, or the micro-vertices respectively, the organization of which we describe in further detail in Section 3.1. Additional requirements that are needed to avoid gaps and discontinuities between triangles with micromaps of different subdivision levels are described further in Section 3.2.

3.1. Micro-triangles and Micro-vertices

There are numerous ways to organize both micro-triangles and micro-vertices inside a micromap: Early approaches organized micro-triangles in a top-to-bottom stripe based way [GBW20], whereas the official extensions eventually settled on ordering them along a special space-filling curve, similar to the Z-order curve [Khr22]. As for micro-vertices: We have identified three distinct approaches to organizing them inside micromaps:

W-Minor Vertices arranged from top-to-bottom from the w barycentric coordinate as inferred by Gruen et al [GBW20].
U-Major Arranged bottom-to-top, starting from the edge between the w and v barycentric coordinates, as implemented in [NVI25].
Bird Arranged hierarchically in triplets along the so-called *Bird*-curve, as implemented in [NVI25].

Each of these methods are summarized in figure 2 for subdivision levels 2 and 3.

3.2. Independence and Edge Handling

Micromaps are always fully independent of each other, which is natural for micro-triangle ones, but for micro-vertices two limitations arise: First, all vertices along the edges of two triangles need to be duplicated, despite technically referring to the same locations. Second, the subdivision level of any neighboring micromap *should* only differ by one: Otherwise, cracks may appear in those regions. This issue is most pronounced for displacement micromaps, but may also surface for general attributes as discontinuities, since the micro-vertex placement become inconsistent.

In this work, this particular issue is avoided by ensuring that all micro-vertex micromaps use the same subdivision level, but in a real application this is handled by finding and decimating the triangles along the affected edges; as shown in figure 3.

4. Rendering Attribute Micromaps

In this section we detail our primary contributions that are not directly derived from existing micromap material.

4.1. Micro-Vertex Lookup

In contrast to the Vulkan® Opacity Micromap extension [Khr22], the micro-vertex based micromaps used in the Displacement Micromap extension did not include an algorithm for looking up micromap values from barycentric coordinates. However, it is possible to infer such an algorithm from the existing implementation [NVI25], but to our knowledge, there is no such algorithm for the **W-Minor** ordering. Thus, in listing 1 we present an algorithm for finding the indices to the closest triplet of micro-vertices from the barycentric coordinates. The corresponding **U-Major**- and **Bird**-order vertex-index algorithms can be found in the supplemental material.

Once these indices are found, the barycentric coordinates can be remapped to the intersected micro-triangle for a final interpolation step. This may be done in a number of ways, but one general approach is as follows:

Table 1: *The opacity values that may be used in the official opacity micromap format in Vulkan® and DirectX®.*

2-State		*4-State*	
0	Fully Transparent	**0**	Fully Transparent
1	Fully Opaque	**1**	Fully Opaque
		2	Unknown Transparent
		3	Unknown Opaque

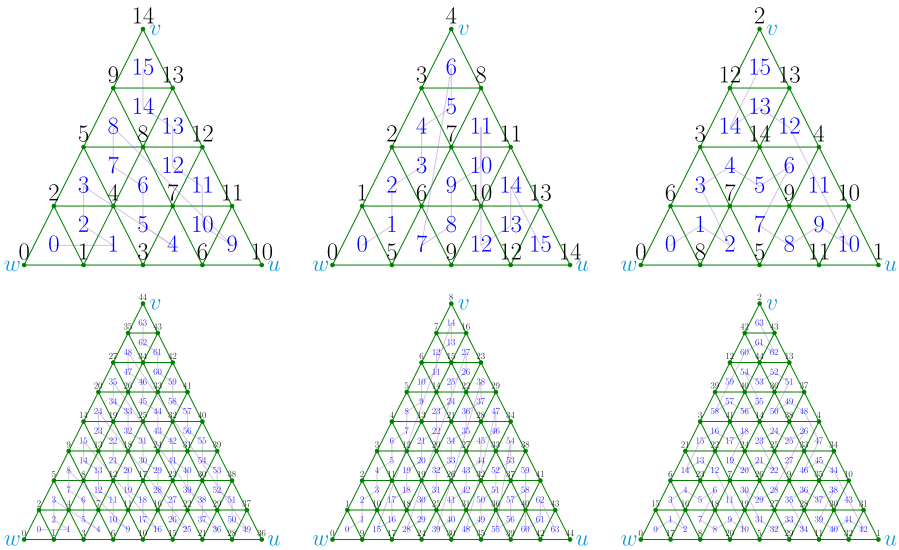


Figure 2: *Visualization of the three identified micromap primitive orderings, showing both the micro-triangle and micro-vertex indices for each scheme.*

1. Express the triangle intersection point (P) with barycentric coordinates for both the primary triangle vertices (V), and the intersected micro-triangle vertices (V^μ):

$$P = wV_0 + uV_1 + vV_2 = w'V_0^\mu + u'V_1^\mu + v'V_2^\mu$$

2. As each of the micro-triangle vertices belong to the same triangle, we can express each of them with barycentric coordinates with respect to the original vertices:

$$\begin{cases} V_0^\mu = w_0V_0 + u_0V_1 + v_0V_2 \\ V_1^\mu = w_1V_0 + u_1V_1 + v_1V_2 \\ V_2^\mu = w_2V_0 + u_2V_1 + v_2V_2 \end{cases}$$

3. Inserting them in the original equation and gathering the terms for each of the original vertices forms the matrix equation:

$$\begin{bmatrix} w_0 & w_1 & w_2 \\ u_0 & u_1 & u_2 \\ v_0 & v_1 & v_2 \end{bmatrix} \begin{bmatrix} w' \\ u' \\ v' \end{bmatrix} = \begin{bmatrix} w \\ u \\ v \end{bmatrix}$$

4. Solving this 3×3 system using e.g., Cramer's method yields the remapped barycentric cooordinates for any kind of sub-triangle.

Simpler approaches based on the over-determined system that only uses two of the barycentric coordinates, or the uniform micromap tessellation pattern can also be used, but are omitted for brevity.

In summary, the following steps are taken to look-up a micro-vertex attribute:

1. Find the intersected micro-triangle.
2. Find the corresponding micro-vertex indices.
3. Fetch the micromap attributes at these indices.
4. Remap the barycentric coordinates to the micro-triangle.
5. Interpolate the fetched attributes with the remapped coordinates.

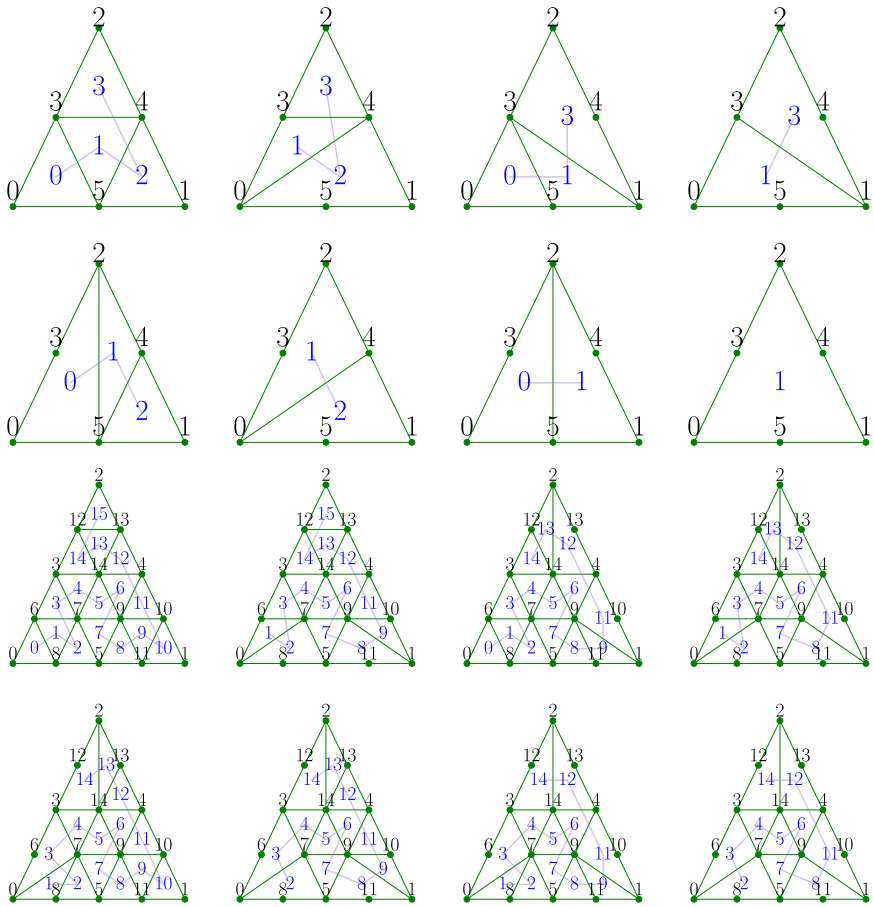


Figure 3: *Exhaustive enumeration of the micromap edge decimations used for the first two levels of micromaps. Note that other strategies can be used, but the rules must be consistent to avoid cracks forming between neighboring micromaps. Observe that vertices are not removed from the mesh: The mapping from the micro-triangle to micro-vertex indices is merely adjusted for the affected edges.*

4.2. GLSL Lookup

The only attribute that is readily accessible to the end-user in the ray-tracing pipeline is the triangle-local barycentric coordinates, and, through an additional extension, the triangle vertex positions[†]. Any other type of attribute must be fetched manually using indexing based on the instance- and primitive-IDs. While these positions and barycentrics are the most commonly used attributes, most shading algorithms typically require more data, and at the very least the surface normal.

[†] `GL_EXT_ray_tracing_position_fetch`

```python
def wminor_uv2triangle_index(u, v, level):
    """Find the micro-triangle index."""
    N = 2**level
    w = 1.0 - (u + v)
    row = min(N - 1, uint32((1.0 - w) * N))
    col = min(row,   uint32(v * N))
    dia = min(N - 1, uint32((1.0 - u) * N))
    return (row * row) + col + (dia - (N - 1 - row))

def wminor_uv2vertex_index(u, v, level):
    """Find the closest triplet of micro-vertex indices."""
    index = wminor_uv2triangle_index(u, v, level)
    row = uint32(sqrt(index))
    col = index - row * row
    def to_index(r, c):
        """Convert a row/column pair to a vertex index."""
        return r * (r + 1) // 2 + c
    if col % 2 == 0:
        v0 = (row, col // 2)
        v1 = (row + 1, col // 2)
        v2 = (row + 1, col // 2 + 1)
    else:
        v0 = (row, col // 2)
        v1 = (row + 1, col // 2 + 1)
        v2 = (row, col // 2 + 1)
    return (to_index(*v0), to_index(*v1), to_index(*v2))
```

Listing 1: *Algorithm for finding the indices of the closest triple of micro-vertices arranged according to the **W-Minor** ordering.*

```
#extension GL_EXT_attribute_micromap : require

layout(binding=0, triangleMicromapEXT) buffer utri
{
    vec4 color;
} uTriAttributes;

layout(binding=1, vertexMicromapEXT) buffer uvtx
{
    vec3 normal;
} uVtxAttributes;
```

Listing 2: *An imagined `GLSL API` for accessing attribute-micromap values for both micro-triangles and micro-vertices.*

Theoretically, micromaps could be extended to provide users with a convenient method for accessing arbitrary attributes in the ray-tracing pipeline, effectively performing the steps outlined in Section 4.1 using the proposed `API` shown in listing 2. Naturally, such a change would require extensions to `GLSL`, `SPIR-V` and `Vulkan` to be usable, but a start at such extensions are available in the supplemental material. For this work however, all attribute fetching is handled manually in software without additional compiler or hardware support.

4.3. Fixed-Functions

In the rasterization pipeline, a number of fixed blending functions can be used to implement various types of transparency. However, it is up to the user to ensure that objects are rendered in the correct order for this blending to look correct. In the ray-tracing pipeline any given ray will similarly pass through all objects that it needs to consider for blending, but as they are also processed in an arbitrary order, complex shader code and buffering of partial results in the ray-payload may be required to achieve the expected result.

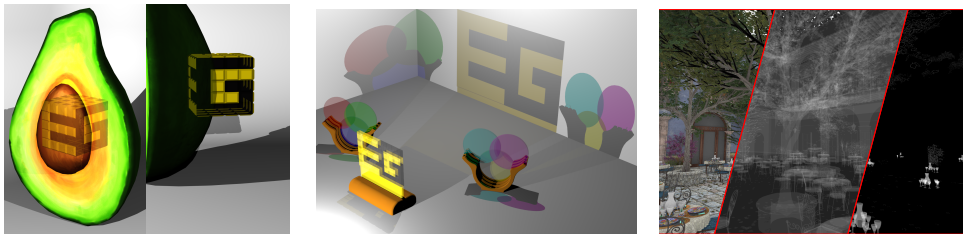
Thus, another potential use case for attribute micromaps would be to allow the ray-tracing pipeline to execute a set of fixed functions on some part of the payload as specified by an offset with the micromap values as operands, thereby potentially avoiding extra shader calls and preventing the hardware traversal from interruptions [Sjö22]. With an appropriate set of these some commonly used operations could be greatly accelerated:

Trivial examples include order independent arithmetic operations, such as `min`, `max`, summations, products, or front-/back-face counters, which can be used for limited forms of volume rendering [WM90, Mat22], estimating alpha- or depth-complexity, or for simple transparency effects such as diffuse transmissions or colored shadows [TZ21] as we demonstrate in figure 4.

Increasing the complexity slightly: A function could estimate the `min` *and* `max` intersections belonging to the same object, thus estimating the object optical depth, which could greatly benefit single-scattering and non-overlapping volume rendering applications. Another highly desirable feature of the same vein would be to find the `k` closest intersections, which could greatly benefit particle based rendering approaches such as Gaussian splats [MLMP*24], but we recognize that such a feature may be prohibitively expensive.

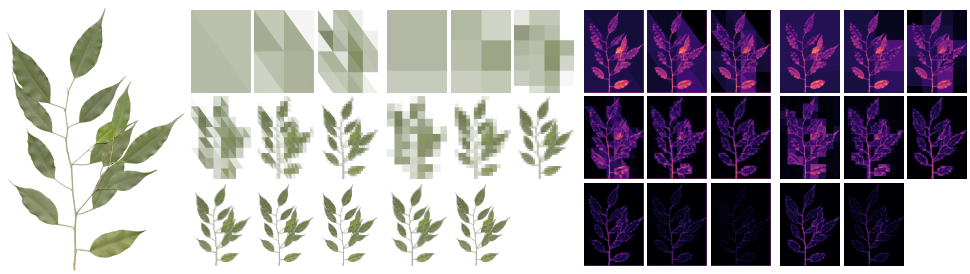
4.4. Mipmapping

Micro-triangle based micromaps can be readily interpreted as a tree-like structure. As such, it is straight-forward to derive a type of mipmap-structure; all that is needed is a method for computing a new micromap value, given the values from the 4 child nodes [WD24]. As an



(a) Maximum Intensity Projection **(b)** Colored Shadows **(c)** Depth and Alpha Complexity

Figure 4: *A small set of use-cases for a possible fixed-function addition to the existing ray-tracing pipeline. In **(a)**, we perform a component-wise `max` on the payload with a special color-micromap and in **(b)** component-wise multiplication is done to form colored shadows. In **(c)**, we compute the depth- and alpha-complexity for each pixel by either counting all primitives, or all those with non-zero alpha along the ray.*



(a) *Original* **(b)** *Micromap-Mipmaps* **(c)** *Image-Mipmaps* **(d)** *Micromap* ꟻLIP **(e)** *Image* ꟻLIP

Figure 5: *Mipmaps created for a micro-triangle based color micromap by averaging the color of the 4 underlying child nodes (**b**), the equivalent image-based mipmaps (**c**) as well as the* ꟻLIP *error maps between each mipmap level and the original texture (**d** and **e**). Note that two triangles are used to create a comparable rectangle which consequently have two colors at the highest mip-level. Thus, to the make the visual comparison fairer to image based mipmaps, we drop the highest mip-level and shift the hierarchy accordingly.*

example: For a color micromap, the corresponding mipmap can be constructed as the average color of the child nodes. This is illustrated in figure 4.4 along with the regular image-base mipmaps and ꟻLIP comparisons to the original texture.

Similar to image-based mipmaps, the additional memory requirement for storing mipmaps is roughly 33 % of the original micromap size as the series: $\sum_{i=1}^{\infty} \frac{1}{4^i}$ converges to $\frac{1}{3}$.

In contrast to image-based mipmaps, those associated with micromaps are always strongly connected to a triangle, thus avoiding some issues such as texture bleeding in atlases but consequently cannot filter across polygon edges.

4.4.1. Micro-Vertex Mipmapping

In contrast to micro-triangles, micro-vertices arguably do not form the same straight-forward hierarchy, but it is still possible to define a mipmapping structure similar to the one used by mesh colors as shown by Cem Yuksel et al [YKH10].

Alternatively, the micro-vertices can be limited to the lowest mipmap level, and face-based micromaps can be used for any higher levels, thus re-creating a simple hierarchy.

4.4.2. Using the Mipmaps

Once created, a corresponding algorithm for finding the appropriate micromap level-of-detail is required to find which mip-levels to sample, preferably without accessing other vertex attributes. For ray-cones, this can be done similar to the anisotropic texturing approach of Akenine-Möller et al [AMCB*21]:

1. Retrieve the world-space triangle positions P_0, P_1, and P_2[†].

2. Compute the triangle normal $\vec{n}$ and area A_T, as well as the projected ray-cone area A_c from the and spread-angle γ and ray hit parameter t:

$$\vec{n} = (P_1 - P_0) \times (P_2 - P_0)$$
$$A_T = 0.5\,\|\vec{n}\| = 0.5\,\|(P_1 - P_0) \times (P_2 - P_0)\|$$
$$A_c = \|\vec{a}_1\|\,\|\vec{a}_2\|\,\pi$$

Where $\vec{a}_1$ and $\vec{a}_2$ are the major axes of the ellipse formed by the projected ray-cone:

$$\vec{a}_1 = \frac{t\tan(\gamma/2)}{\|\vec{h}_1 - (\vec{d}\cdot\vec{h}_1)\vec{d}\|}\vec{h}_1 \qquad \vec{h}_1 = \vec{d} - (\vec{n}\cdot\vec{d})\vec{n}$$
$$\vec{a}_2 = \frac{t\tan(\gamma/2)}{\|\vec{h}_2 - (\vec{d}\cdot\vec{h}_2)\vec{d}\|}\vec{h}_2 \qquad \vec{h}_2 = \vec{n}\times\vec{h}_1$$

3. If the cone-area A_c exceeds the triangle area A_t, use the highest available mip-level.
4. Otherwise, the appropriate starting mip-level is the first subdivision level where the cone-spread is greater than the micro-triangle area ($A_{\mu t} = \frac{A_t}{4^n}$):

$$A_c \geq \frac{A_t}{4^n} \implies 2^{2n} \geq \frac{A_t}{A_c} \implies n \geq \frac{1}{2}\log_2(\frac{A_t}{A_c})$$

5. Finally, interpolate between the micromap values found at mip-levels: $\lfloor n \rfloor$ and $\lfloor n+1 \rfloor$.

This is described schematically in figure 6.

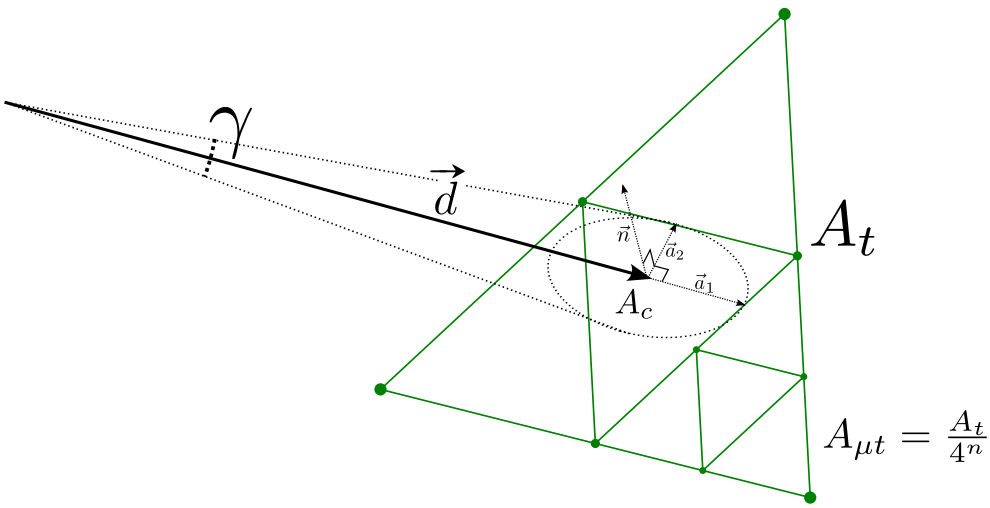


Figure 6: *Schematic view of the variables used to find the micromap level-of-detail given a ray-cone with spread angle γ and projected cone-spread area A_c for the triangle with area A_t and micro-triangle area $A_{\mu t}$.*

5. Results

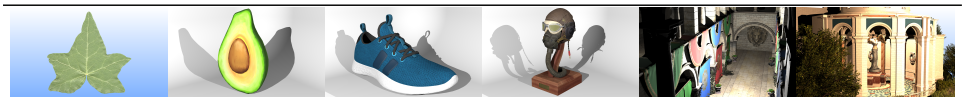
In this work, we have created a software pipeline for evaluating the memory footprint, frame-runtime, and quality of both micro-triangle- and micro-vertex-based micromaps for colors, normals, and metalness-roughness factors to replace the original textures at subdivision levels 0 to 8 for the scenes described in table 2.

5.1. Construction and Footprints

Each micro-triangle-based micromap is constructed by averaging the pixels covered by each sub-triangle projected onto each of the textures. Similarly, each micro-vertex-based micromap is created by performing bilinear sampling at each projected micro-vertex location. Once constructed, the total uncompressed memory footprint of the original textures and the resulting micromaps are recorded for each subdivision level, as shown in figure 7.

Table 2: *Description of all tested scenes with a number of relevant properties. Note that all micromap features depend on the subdivision level and are thus listed as a range and that ORCA assets were authored with a specular workflow, and thus lack metalness and roughness.*

Scenes	*Quad*	*Avocado*	*Shoe*	*Flight-Helmet*	*Sponza*	*Sun-Temple*
Instances	1	2	2	7	1	517
Meshes	1	2	2	7	1	517
Triangles	2	1115	23 133	95 155	262 267	606 376
Textures	2	3	3	18	73	148
Opacity Micromaps	0 to 2	0	0	0	0 to 3801	0 to 569
Special Indices (Opacity)	0 to 2	0	0	0	19 to 3820	334 038 to 338 783
Micro-Triangle Color Micromaps	2	587 to 682	2103 to 14 555	8426 to 85 570	26 321 to 73 542	11 691 to 43 778
Micro-Triangle Normal Micromaps	1 to 2	372 to 682	8438 to 21 323	20 260 to 88 897	11 905 to 45 470	23 196 to 45 263
Micro-Triangle Metal/Roughness Micromaps	0	5 to 682	12 163 to 22 272	31 760 to 91 074	4725 to 40 539	0
Micro-Vertex Color Micromaps	1 to 2	682	11 879 to 14 669	69 561 to 86 844	72 682 to 73 539	41 967 to 44 010
Micro-Vertex Normal Micromaps	1 to 2	682	19 513 to 21 647	86 232 to 88 886	44 287 to 45 027	43 602 to 45 298
Micro-Vertex Metal/Roughness Micromaps	0	22 to 382	21 144 to 22 310	90 174 to 91 090	34 443 to 40 179	0



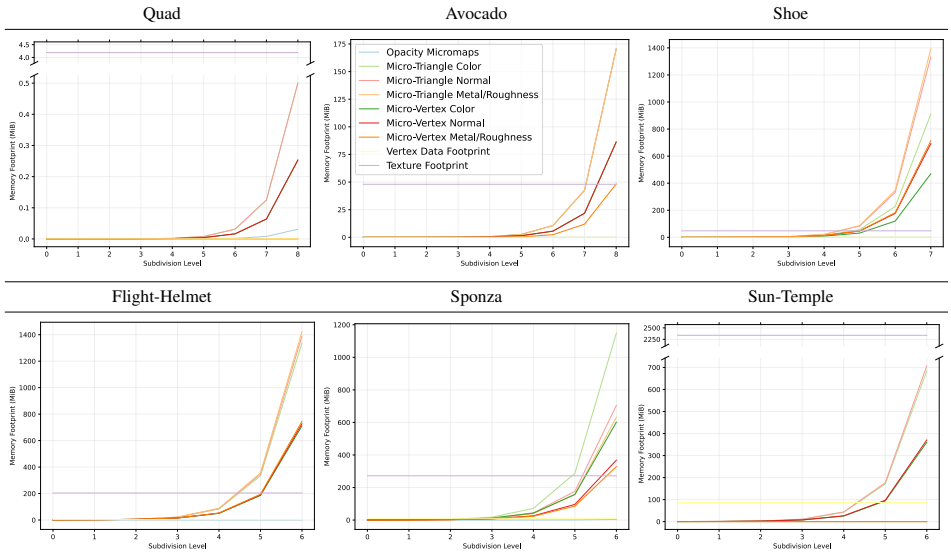


Figure 7: *Memory footprint for each scene and subdivision level in megabytes (*MB*). Axis lines additionally show the footprint required by textures (purple) and vertex attributes (yellow).*

5.2. Rendering Frame-time and Quality

Each scene is rendered on an NVIDIA GeForce RTX 3080 at 1920×1080 with a small set of point-lights in a single-bounce ray-tracer, where all surfaces are shaded with the Trowbridge-Reitz (GGX) [TR75, WMLT07] BRDF model, once with the original textures, and then again for the micro-triangle- and the micro-vertex-based micromaps respectively, at each subdivision level.

The frame-time of each render is estimated by averaging the start- to end-of-pipe time as reported by the Vulkan pipeline querying API over many frames, as can be seen in figure 8.

Each micromap render is compared to the corresponding texture-based rendering using the ꟻLIP image metric [ANA*20], thumbnails of which can be seen in table 3. Plots of the mean ꟻLIP value can be seen in figure 9.

Full-scale images with accompanying error maps can be found in the supplemental material, but results from additional rendering methods, such as an *unlit* shading model, (i.e., base color with an excessively sampled ambient occlusion), and world-space normal visualizations are included to spread out the runtime results and to capture how it changes over subdivision levels in a few different scenarios.

6. Discussion

In this section we discuss the potential trade-offs from using micromaps to represent general attributes.

6.1. Memory Footprints

Ideally, micromaps should be constructed as a part of the retopologizing process of high-resolution models, similar to the process suggested by Maggiordomo et al [MMT23] or even be made as a part of the asset creation itself, as suggest for mesh colors [YKH10], rather than sampled from textures as is done in this work. However, by using existing textured assets as a base effectively turns the micromaps into downsampled versions of the textures, making it straight-forward to evaluate the quality and memory footprint. Thus, we can easily see that despite creating numerous micromaps, the total memory footprint in the scene remains consistently low for subdivision levels 0 to 5, which are the levels we would consider reasonable for general models. Although, in some corner cases it can be feasible to use many more subdivision levels, such as for the Quad model, which can go beyond level 8 before it reaches parity with the original textures.

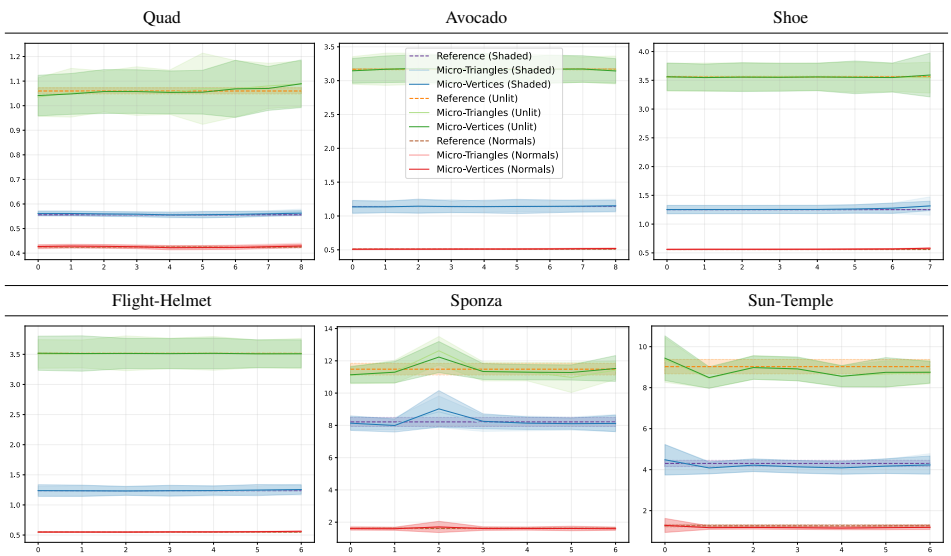
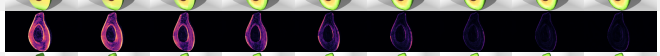
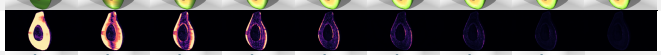
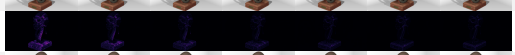
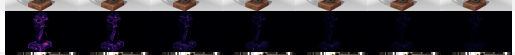
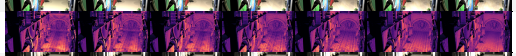
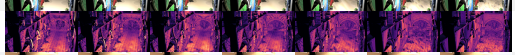


Figure 8: *Frame runtime results in milliseconds* (ms).

Table 3: *Rendered quality comparison for each scene and subdivision level with associated* ꟻLIP *error map.*

Scene	*Mode*	*Attribute*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*Reference*
Quad	*Shaded*	*Face*										
	ꟻLIP											
		Vertex										
	ꟻLIP											
Avocado	*Shaded*	*Face*										
	ꟻLIP											
		Vertex										
	ꟻLIP											
Shoe	*Shaded*	*Face*										
	ꟻLIP											
		Vertex										
	ꟻLIP											
Flight-Helmet	*Shaded*	*Face*										
	ꟻLIP											
		Vertex										
	ꟻLIP											
Sponza	*Shaded*	*Face*										
	ꟻLIP											
		Vertex										
	ꟻLIP											
Sun-Temple	*Shaded*	*Face*										
	ꟻLIP											
		Vertex										
	ꟻLIP											

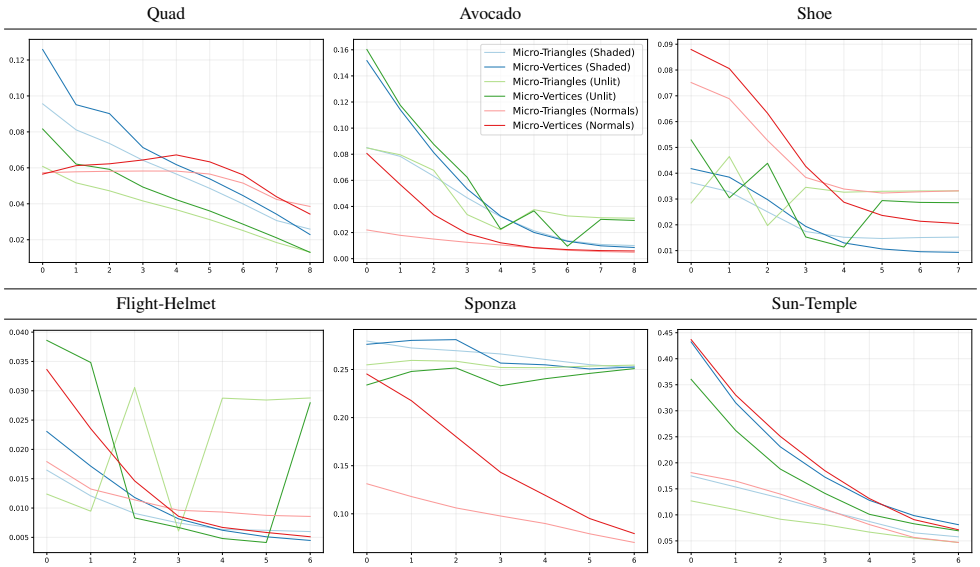


Figure 9: *Mean* ꟻLIP *values for each subdivision level.*

6.2. Frame-time

The frame-time performance was mostly disregarded in this work as it is not possible make a fair comparison between the hardware accelerated texturing pipeline and our software implementation for looking-up attribute micromaps. However, as can be seen in figure 8, the micromap look-up algorithms are all in-line with the references for all shading methods despite this limitation. As such, it stands to reason that a hardware implementation of the look-up algorithms described in this work may actually create a noticeable improvement as it can do for mesh colors [MSY19]. Furthermore, the micromap look-up algorithm coupled with the ray-cone-based level-of-detail approach that was outlined in Section 4.4.2 may be a suitable target for such an implementation as it is simple to implement, has immediate access to all ancillary data, and no additional memory overhead beyond the two floats needed for the ray-cone itself.

6.3. Quality

As seen in figure 9, the overall rendering quality improves as higher subdivision micromaps are used, regardless of which rendering model is used. This is to be expected, as the micromaps gradually captures more features from the original textures, but in turn become prohibitively more expensive memory-wise, as can be seen in figure 7. Micromaps based on micro-triangles also frequently receive a lower mean ꟻLIP value than the corresponding micro-vertex ones at the same level, but are subjectively of worse quality due to the obvious tessellation pattern. Although, once the subdivision level is high enough, this is no longer noticeable.

It appears as if micro-vertex micromaps are particularly suitable for assets with a dense vertex distribution, as can be seen clearly for the Shoe and Flight-Helmet models in figure 3: In these scenes, it is possible to get by with a very low amount of subdivision levels, typically between 2 to 4, without noticeably impacting the visual quality, yet only requiring between 7 % to 71 % of the original texture footprint. In contrast, in models with a very sparse vertex distribution, such as in most of the Sponza scene and some parts of the Sun-Temple scenes, these types of micromaps cannot be used in practice due to the high subdivision level needed to reach an acceptable quality. The same issue can also be seen in figure 1 along the tongue of the shoe where the texture detail is high, but vertex density is relatively low, although to a much lower extent.

One defining feature of micromaps are that they should be independent of the triangle attributes, thus reducing the memory bandwidth by not having to load any unnecessary vertex data. This works well for most shading attributes, but can cause issues for micromaps containing *normals*. In those cases, the normal gets coupled to the vertex-normals and tangents in addition to the texture coordinates and normal-map.

It is possible to decouple all of them, effectively storing micromap normals in object-space rather than tangent-space, but this may lead to various shading artifacts for non-uniformly scaled geometry.

Furthermore, some texture-filtering approaches, such as ray-differentials, and curvature approximations still require the vertex normals [Ige99, AMCB*21]. Thus, we would consider these types of micromaps to only be practical for static geometry.

7. Future Work

While it appears that displacement micromaps have been abandoned in favor of a more dynamic level-of-detail system inside the acceleration structure [Khr25], there are still a lot of potential use-cases for more generalized types of micromaps.

In this paper we only briefly touched upon how these generalized micromaps are actually authored: A complete system for generating and working with micromaps would need to handle a number of corner cases that was glossed over in this work. Additionally, finding algorithms to efficiently update or modify micromaps during runtime and for animations may also be an interesting avenue of research, similar to the work done by Gruen et al [GBK*24], which may in turn benefit from having fast access to arbitrary animation parameters.

One major limitation with the micromap format, is the lack of dedicated design software that support them: Existing image textures can easily be modified in any number of image-editors, but micromaps are limited to be generated from a handful of specialized tools. Thus, adding support for visualizing the underlying micromap-meshes, or even painting or editing individual micro-primitive values similar to what was proposed for *mesh colors* [YKH10] could prove very useful. However, unlike micromaps, mesh colors work on both triangular- and quadrilateral faces, something that is essential for 3D modeling. Thankfully, in the work by Waldemarson et al [WD24] they concluded that the micromap subdivision scheme may generalize to other shapes which could consequently be used to similarly extend micromaps.

Interestingly, given that micromaps can be defined for both micro-triangles and micro-vertices, it stands to reason that one can create micromaps to store data for other aspects of the triangle as well, e.g., the micro-*edges*. This may be practical for some things, such as marking sharp edges, although, it remains to be seen if something like that is useful in practice.

Given the more generalized data stored in these micromaps, it would make sense to invent a new scheme for compressing and looking up the attribute data; as approaches for compressing vertex attributes may be applicable. Additionally, in this work we did not use *special-indices* for anything besides opacity micromaps. It is possible that something similar could be used for general micromaps, e.g., as a palette of fixed values.

Moreover, several rendering algorithms operate directly on bare barycentric coordinates; such as the recently presented "Surface-Stable Fractal Dithering" [Joh25], as such, it may be possible to directly replace these algorithms with appropriate micromaps, particularly if coupled with appropriate micromap mipmapping hierarchies.

It is worth pointing out though, that recent advances in filtering, namely *stochastic texture filtering* [PWSF24], may make mipmapping unnecessary, particularly when coupled with temporal anti-aliasing. However, a thorough analysis of the performance and quality ramifications under such conditions would be necessary.

8. Conclusions

In this paper we have presented micromaps with more generalized attributes, such as colors and normals, with all the necessary algorithms needed for them to be used as shading primitives in a rendering engine, allowing them to be used in place of the corresponding textures. Thus allowing assets with dense vertex distributions to use micromaps to substantially reduce the memory footprint with a minimal loss of quality, and no perceptible loss of frame-time performance; even without proper hardware acceleration.

We also further extend the micromap concept with an analogue to texture mipmapping, a `GLSL API` proposal to make accessing micromap values more convenient in ray-tracing hit-shaders, and show how micromaps can be used for a variety of fixed-function operations in the ray-tracing pipeline, potentially allowing the implementation of effects such as partial transparency without the need for `AnyHit`-shaders.

9. Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We would also like to thank Arm Sweden AB for letting Gustaf pursue a PhD as one of their employees. Additionally, we would like to thank Rikard Olajos for his valuable input during this work. We are also very grateful for the extensive input from our reviewers that helped us identify some areas that needed a bit of extra work.

Finally, we would like to thank the authors of the assets and scenes used in this work: Guillermo M. Leal Llaguno for the San-Miguel scene [Lla10], Microsoft and Shopify for the Avocado and Shoe models, and Gary Hsu for the Flight-Helmet model, all provided by the Khronos Group among their `glTF` sample assets [Khr23a], Frank Meinl, Marko Dabrovic and CryTek for the Sponza scene, and Amazon and Unreal Engine through the Open Research Content Archive (ORCA) for the Sun-Temple and Bistro scenes [Epi17, Ama17].

References

[Ama17] AMAZON LUMBERYARD: Amazon Lumberyard Bistro, Open Research Content Archive (ORCA), July 2017. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro. 13

[AMCB*21] AKENINE-MÖLLER T., CRASSIN C., BOKSANSKY J., BELCOUR L., PANTELEEV A., WRIGHT O.: Improved shader and texture level of detail using ray cones. *Journal of Computer Graphics Techniques (JCGT) 10*, 1 (January 2021), 1–24. URL: http://jcgt.org/published/0010/01/01/. 7, 13

[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–15:23. doi:10.1145/3406183. 10

[BD22] BARBIER W., DUPUY J.: Htex: Per-halfedge texturing for arbitrary mesh topologies. *Proc. ACM Comput. Graph. Interact. Tech. 5*, 3 (July 2022). URL: https://doi.org/10.1145/3543868, doi:10.1145/3543868. 2

[Bic23] BICKFORD N.: Nvidia micromap extensions, 2023. URL: https://github.com/NBickford-NV/glTF/tree/micro-mesh. 2

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008* (2008), pp. 1155–1164. 2

[Cat74] CATMULL E. E.: *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, The University of Utah, 1974. AAI7504786. 1

[DZJ*24] DOU Y., ZHENG Z., JIN Q., SHI R., LI Y., NI B.: Differentiable micro-mesh construction. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2024), pp. 4294–4303. doi:10.1109/CVPR52733.2024.00411. 2

[Epi17] EPIC GAMES: Unreal Engine Sun Temple, Open Research Content Archive (ORCA), Oct. 2017. URL: http://developer.nvidia.com/orca/epic-games-sun-temple. 13

[FO23] FENNEY S., OZKAN A.: Compressed Opacity Maps for Ray Tracing. In *High-Performance Graphics - Symposium Papers* (EUROGRAPHICS Association Groene Loper 3, 5612AE Eindhoven, 2023), Bikker J., Gribble C., (Eds.), The Eurographics Association, pp. 23–31. doi:10.2312/hpg.20231133. 2

[GBK*24] GRUEN H., BENTHIN C., KENSLER A., BARCZAK J., MCALLISTER D.: Ray tracing animated displaced micro-meshes. *Computer Graphics Forum 43*, 7 (2024), e15225. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.15225, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.15225, doi:https://doi.org/10.1111/cgf.15225. 2, 13

[GBW20] GRUEN H., BENTHIN C., WOOP S.: Sub-triangle opacity masks for faster ray tracing of transparent objects. *Proc. ACM Comput. Graph. Interact. Tech. 3*, 2 (Aug. 2020). URL: https://doi.org/10.1145/3406180, doi:10.1145/3406180. 1, 2, 3

[Ige99] IGEHY H.: Tracing ray differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., p. 179–186. URL: https://doi.org/10.1145/311535.311555, doi:10.1145/311535.311555. 13

[Joh25] JOHANSEN R. S.: Dither3d, 2025. Accessed: 2025-03-06. URL: https://github.com/runevision/Dither3D. 13

[Khr22] KHRONOS GROUP: `VK_EXT_opacity_micromap` Vulkan Extension Specification, 2022. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html. 2, 3

[Khr23a] KHRONOS GROUP: `glTF` Sample Assets (v2.0). https://github.com/KhronosGroup/glTF-Sample-Assets/tree/Models, 2023. Accessed: 2025-09-16. 13

[Khr23b] KHRONOS GROUP: `VK_NV_displacement_micromap` Vulkan Extension Specification, 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_NV_displacement_micromap.html. 2

[Khr25] KHRONOS GROUP: `VK_NV_cluster_acceleration_structure` Vulkan Extension Specification, 2025. URL: https://registry.khronos.org/vulkan/specs/latest/man/html/VK_NV_cluster_acceleration_structure.html. 13

[Lla10] LLAGUNO G. M. L.: San miguel, 2010. https://www.pbrt.org/scenes-v3/. 13

[Mat22] MATTSSON L.: Simulating optical depth for participating media using ray tracing, 2022. Student Paper. URL: http://lup.lub.lu.se/student-papers/record/9110658. 6

[MLMP*24] MOENNE-LOCCOZ N., MIRZAEI A., PEREL O., DE LUTIO R., MARTINEZ ESTURO J., STATE G., FIDLER S., SHARP N., GOJCIC Z.: 3d gaussian ray tracing: Fast tracing of particle scenes. *ACM Trans. Graph. 43*, 6 (Nov. 2024). URL: https://doi.org/10.1145/3687934, doi:10.1145/3687934. 6

[MMT23] MAGGIORDOMO A., MORETON H., TARINI M.: Micro-mesh construction. *ACM Trans. Graph. 42*, 4 (July 2023). URL: https://doi.org/10.1145/3592440, doi:10.1145/3592440. 2, 10

[MSY19] MALLETT I., SEILER L., YUKSEL C.: Patch Textures: Hardware Implementation of Mesh Colors. In *High-Performance Graphics - Short Papers* (2019), Steinberger M., Foley T., (Eds.), The Eurographics Association. doi:10.2312/hpg.20191194. 2, 12

[NVI25] NVIDIA GAMEWORKS: Displacement micromap toolkit, 2025. Accessed: 2025-03-06. URL: https://github.com/NVIDIAGameWorks/Displacement-MicroMap-Toolkit. 2, 3

[PWSF24] PHARR M., WRONSKI B., SALVI M., FAJARDO M.: Filtering after shading with stochastic texture filtering. *Proc. ACM Comput. Graph. Interact. Tech. 7*, 1 (May 2024). URL: https://doi.org/10.1145/3651293, doi:10.1145/3651293. 13

[Sjö22] SJÖHOLM J.: Best practices for using nvidia rtx ray tracing (updated), 2022. URL: https://developer.nvidia.com/blog/best-practices-for-using-nvidia-rtx-ray-tracing-updated/. 6

[TR75] TROWBRIDGE T. S., REITZ K. P.: Average irregularity representation of a rough surface for ray reflection. *J. Opt. Soc. Am. 65*, 5 (May 1975), 531–536. URL: https://opg.optica.org/abstract.cfm?URI=josa-65-5-531, doi:10.1364/JOSA.65.000531. 10

[TZ21] "TANKI" ZHANG T.: *Handling Translucency with Real-Time Ray Tracing*. Apress, Berkeley, CA, 2021, pp. 127–138. http://raytracinggems.com/rtg2. URL: https://doi.org/10.1007/978-1-4842-7185-8_11, doi:10.1007/978-1-4842-7185-8_11. 6

[WD24] WALDEMARSON G., DOGGETT M.: Succinct opacity micromaps. *Proc. ACM Comput. Graph. Interact. Tech. 7*, 3 (Aug. 2024). URL: https://doi.org/10.1145/3675385, doi:10.1145/3675385. 2, 3, 6, 13

[WM90] WALLIS J. W., MILLER T. R.: Volume rendering in three-dimensional display of spect images. *Journal of Nuclear Medicine 31*, 8 (1990), 1421–1428. URL: https://jnm.snmjournals.org/content/31/8/1421, arXiv:https://jnm.snmjournals.org/content/31/8/1421.full.pdf. 6

[WMLT07] WALTER B., MARSCHNER S. R., LI H., TORRANCE K. E.: Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Goslar, DEU, 2007), EGSR'07, Eurographics Association, p. 195–206. 10

[YKH10] YUKSEL C., KEYSER J., HOUSE D. H.: Mesh colors. *ACM Trans. Graph. 29*, 2 (Apr. 2010). URL: https://doi.org/10.1145/1731047.1731053, doi:10.1145/1731047.1731053. 2, 7, 10, 13

[Yuk17] YUKSEL C.: Mesh color textures. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, Association for Computing Machinery. URL: https://doi.org/10.1145/3105762.3105780, doi:10.1145/3105762.3105780. 2

Appendix

Conference Posters

Gravitas prima, levitas postea. — Seriousness first, frivolity afterwards.

1. **Photon Mapping Superluminal Particles**
   Presented at the *WASP Winter Conference 2020* in Linköping, Sweden. For further details refer to Paper I and section 4.

2. **Hardware Based Ray-Traced Transparency with Opacity Micromaps in Vulkan & DirectX**
   Presented at the *Arm Global Engineering Conference 2024* in Birmingham, Great Britain. For further details refer to Paper III and section 4.7.

3. **Succinct Opacity Micromaps**
   Presented at the *WASP Winter Conference 2025* in Norrköping, Sweden. For further details refer to Paper III and section 4.7.

4. **Parallel Axis Split Tasks for BVH Construction with OpenMP**
   Presented at the *GRAPP Conference 2025* in Porto, Portugal. For further details refer to Paper II and section 4.4.

5. **Real-Time Path-Tracing of Ray-Tracing Benchmarks of Yore**
   Presented at the *High-Performance Graphics Conference 2025* in Copenhagen, Denmark. For further details refer to section 4.6.

6. **Color and Attribute Micromaps**
   Presented at the *High-Performance Graphics Conference 2025* in Copenhagen, Denmark. For further details refer to Paper IV and section 4.7.

Photon Mapping Superluminal Particles

Gustaf Waldemarson
gustaf.waldemarson@arm.com

Michael Doggett
michael.doggett@cs.lth.se

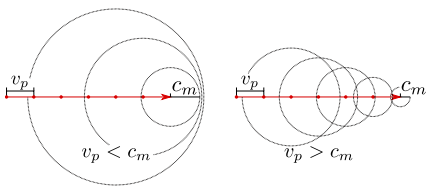
Description

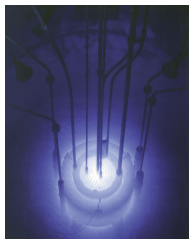
In order to understand how perception works for autonomous systems, a good understanding of the inverse problem is necessary: How does light propagate in a scene? This is the primary question posed in the field of light transport rendering and is commonly solved with ray tracing. In this work a new phenomenon is added to this field: Cherenkov radiation [2]. Light that comes from particles traveling faster than the speed of light for the current medium.

Cherenkov Radiation

As a particle moves through a medium it will excite atoms and cause them to emit electromagnetic waves spherically from the point of interaction.

When the particle exceeds the medium phase speed they constructively interfere, generating coherent photons known as Cherenkov radiation.

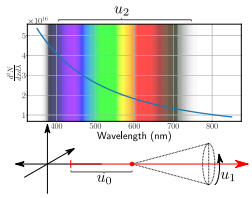




Frank-Tamm Equation

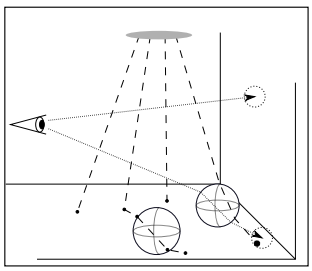
This equation describes how the photon energy gets distributed by a charged particles that travels faster than the speed of light in the medium.

$$\frac{d^2N}{dxd\lambda} = -\frac{2\pi\alpha\mu(\lambda)}{\lambda^2}\left(1 - \frac{c_0^2}{v^2n^2(\lambda)}\right)$$



Photon Mapping

Many different ray tracing algorithms exist but one of the most flexible ones is the Stochastic Progressive Photon Mapping algorithm [1], where paths are traced from both light sources and the camera. Additionally, statistical tricks such as Russian roulette can be utilized to improve rendering performance.



Photon Distributions

Statistical Russian roulette requires prior knowledge of various density distributions, typically denoted as $p(\mathrm{o}, \omega)$ for the photon origin and direction respectively. In this work, it is estimated as follows:

$$\begin{aligned} p(\mathrm{o},\omega) &= p(\mathrm{o}) \cdot p(\omega) \\ p(\mathrm{o}) &= \frac{1}{\text{total particle length}} \\ p(S) &= \frac{\text{superluminal path lengths}}{\text{total path length}} \\ p(\omega) &= p(S)p(\omega_\mathrm{c}) + (1 - p(S))p(\omega_\mathrm{s}) \\ &= \frac{p(S)}{2\pi} + \frac{1-p(S)}{4\pi} = \frac{1+p(S)}{4\pi} \end{aligned}$$

Where the remaining quantities are:

o, ω Photon origin and direction.

$p(S)$ Probability of a particle being superluminal.

$p(\omega_\mathrm{c})$ density for a emitting in a known cone ($\frac{1}{2\pi}$).

$p(\omega_\mathrm{s})$ density for emitting in a sphere ($\frac{1}{4\pi}$).

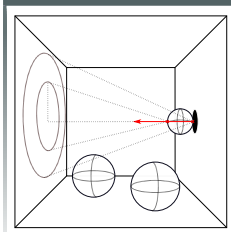
Our Algorithm

1. Choose a point along the particle path
2. Find the refractive index at the location
3. If *superluminal* at the point

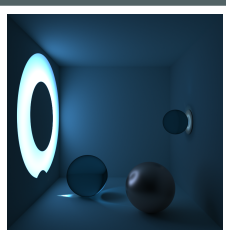
   Find the Cherenkov emission cone and choose a direction along that surface to emit a photon towards
4. Otherwise

   Emit the photon in a random direction
5. Use the Frank-Tamm spectra for the particle as photon color
6. Trace the photon as in [1]

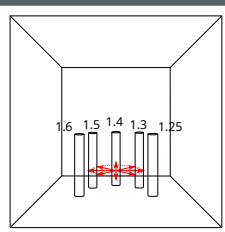
Rendering Results



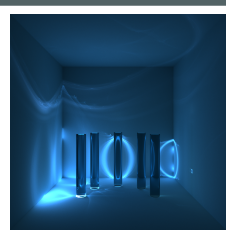
(a) Single particle and Cherenkov cone.



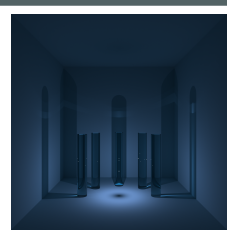
(b) Rendering of (a).



(c) Many particles and varying refractive index.



(d) Rendering of (c).



(e) Rendering (c) with a conventional area light.

References

[1] T. Hachisuka, S. Ogaki, and H. W. Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, Dec. 2008.

[2] P. A. Čerenkov. Visible radiation produced by electrons moving in a medium with velocities exceeding that of light. *Phys. Rev.*, 52:378–379, Aug 1937.

Hardware Based Ray-Traced Transparency with Opacity Micromaps in Vulkan & DirectX

Gustaf Waldemarson, Michael Doggett

Arm Ltd, Lund University, Faculty of Engineering (LTH)

Introduction

Of all the effects in the graphics toolbox, transparency has remained one of the more elusive ones to solve in a general and performant way. Recently, Vulkan® and DirectX® introduced the concept of opacity micromaps [1] which aims to help with this issue by reducing the number of times the ray-tracing pipeline has to invoke the `Any-Hit`-shader, thus improving transparency performance and enabling new ways of manipulating the hardware ray tracing pipeline.

Barycentric Coordinates to Space Filling Curve Index

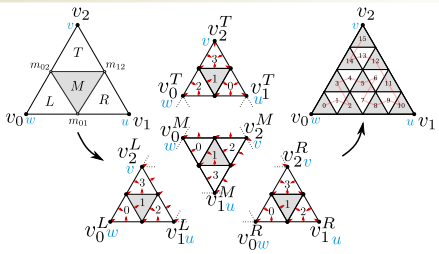


Figure 1: The primitive is recursively split into 4 similar triangles.

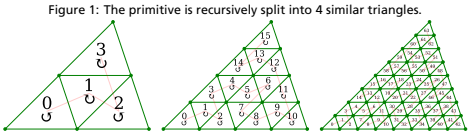


Figure 2: The first three subdivision levels.

Visualization Textures

The micromaps and textures below are all created using an independently developed baking tool that may be made available in the future.

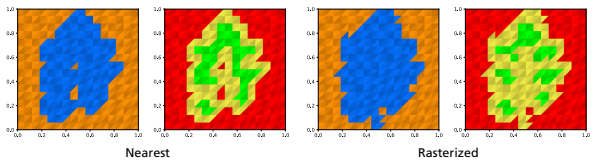


Figure 3: The micromap *quality* is a user parameter: A simple micromap may use the alpha nearest a subtriangle vertex, but to avoid missing small features, each microtriangle should be rasterized. Notice the missing stem in two leftmost images!

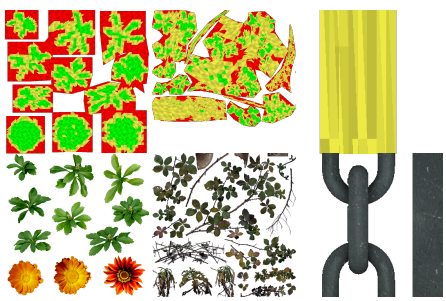


Figure 4: Alpha mapped textures (bottom) and corresponding micromaps (top) from the `Sponza` [2] scene at 3 levels of subdivision. Note that the chain texture (right) is practically useless whereas the other two are more than adequate at despite the low subdivision level.

What is a Micromap?

Micromaps are a user created mapping of subtriangles to one of four transparency values packed into one or two bits of a bit-vector:

4-State (2-bit)	`0b00` Transparent	`0b10` Unknown Transparent (Call `Any-Hit`)
	`0b01` Opaque	`0b11` Unknown Opaque (Call `Any-Hit`)

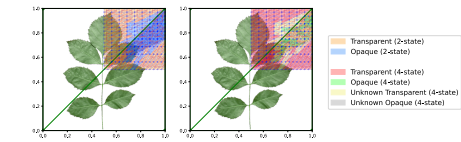
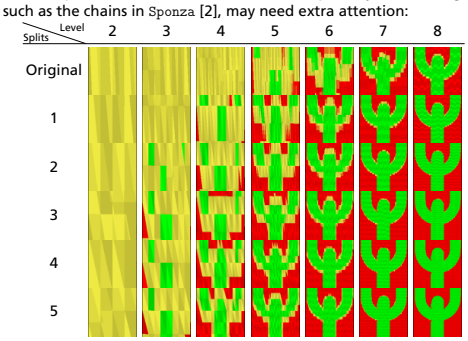


Figure 5: Two examples of a micromapped primitive using the 2-state (left) and 4-state (right) modes.

Not a Panacea for Transparency

Micromaps are not the full solution for transparency. Thin triangles, such as the chains in `Sponza` [2], may need extra attention:



Tab. 1: Visualization of the micromaps for the chandelier chains in the Sponza scene at multiple subdivision levels and length-wise splits

Results

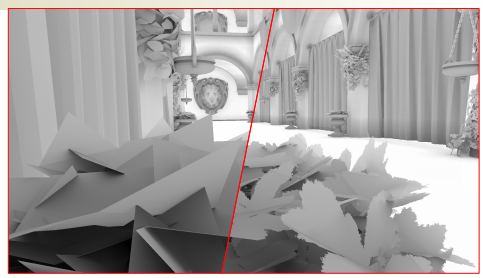


Figure 6: The classic `Sponza` scene with and without micromaps rendered with a simple ambient occlusion renderer.

References and Attributions

[1] E. Werness. "Vk_ext_opacity_micromap," The Khronos Group Inc. (Aug. 24, 2022), [Online]. Available: `https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html` (visited on 05/11/2023)

[2] The Atrium Sponza Palace, Dubrovnik, model courtesy of Frank Meinl (2010). Original model courtesy of Marko Dabrovic (2002).

Succinct Opacity Micromaps

Gustaf Waldemarson & Michael Doggett

Arm Sweden AB, Lund University, Faculty of Engineering (LTH)

Introduction

In Computer Graphics we generate 2D images of 3D scenes primarily from collections of triangles, some of which may be semi-transparent. Thus, in this work we present a new way of representing such translucent triangles efficiently with a novel compression method that reduces the memory footprint by up to 110 times, including an algorithm for looking up values directly from this compressed form.

Further, this method is evaluated in a comprehensive performance analysis in terms of both memory footprint and render-time for a number of scenes along with various other transparency representations, including the so-called opacity micromap representation.

Converting Micromaps to a Tree

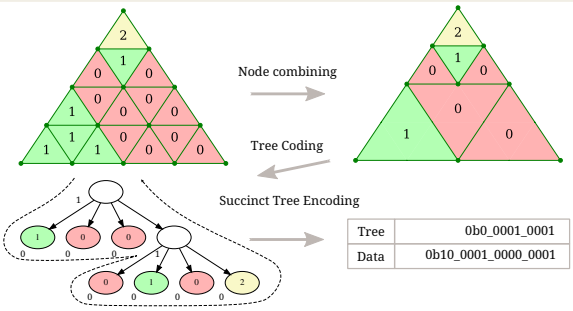


Figure 1: Overview of the opacity micromap to tree conversion process.

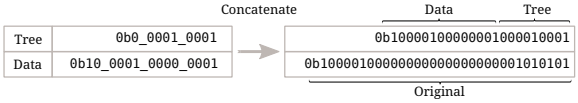


Figure 2: Bitwise comparison of the original opacity micromap and the tree from the above example.

Look-up Algorithm

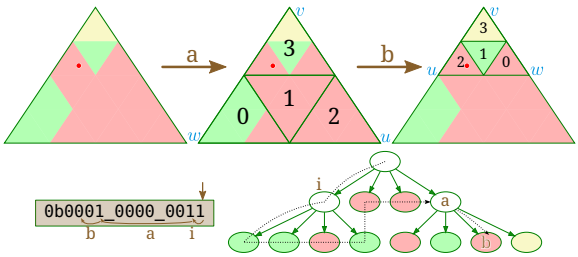


Figure 3: Overview of the look-up algorithm from the micromap tree representation.

a) Determine the tree child index (0-3) and use that as a counter:
- Step at least that number of bits into the tree bitstring.
- However, if another internal node, e.g., i is encountered, increase this counter by 4.

b) When the counter is 0, investigate the bit:
- If the bit is 1: Repeat the above procedure.
- If the bit is 0: The number of passed zeros is the index to the desired opacity value.

What is a Micromap?

A Micromap is a user-created mapping of subtriangles to one of four transparency values packed into one or two bits of a bit-vector:

2-State (1-bit)	`0b0` Transparent `0b1` Opaque
4-State (2-bit)	`0b00` Transparent `0b10` Unknown Transparent (Call `Any-Hit`) `0b01` Opaque `0b11` Unknown Opaque (Call `Any-Hit`)

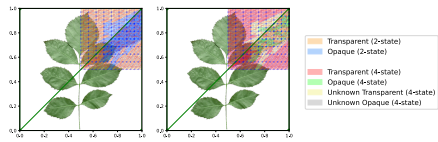


Figure 4: Two examples of a micromapped primitive using the 2-state (left) and 4-state (right) modes.

Footprint Results

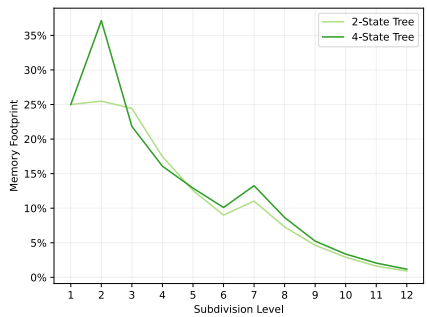


Figure 5: Average footprint reduction over all scenes and subdivision levels.

Frametime Results

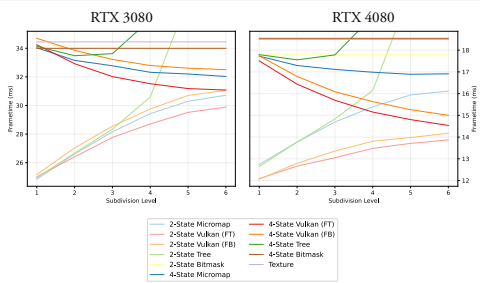
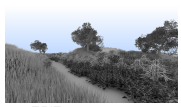


Figure 6: Comparison of the frametime cost for a number of opacity algorithms.

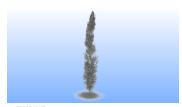
Scenes



CryTek Sponza [2011]



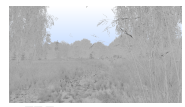
Ecosys [1998]



New Sponza [2022]



San Miguel [2010]



Landscape [2016]

References

[1] Gustaf Waldemarson and Michael Doggett. Aug. 2024. "Succinct Opacity Micromaps." *Proc. ACM Comput. Graph. Interact. Tech.*, 7, 3, (Aug. 2024). DOI: 10.1145/3675385

Parallel Axis Split Tasks for Bounding Volume Construction with OpenMP

Gustaf Waldemarson & Michael Doggett

Arm Sweden AB, Lund University, Faculty of Engineering (LTH)

Introduction

Fast BVH construction is necessary to ensure fast ray-tracing, so a parallel build approach is needed. Thus, in this paper we propose a method of using OpenMP tasking to parallelize the splitting algorithm and thus improve build performance, improving the construction time by between 3 and 5 times on an 8-core machine with a minimal amount of work and negligible quality reduction of the final BVH.

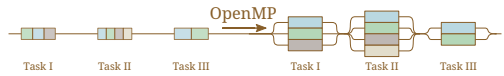


Figure 1: Graphical visualization of the typical OpenMP usage for splitting up tasks into parallel regions.

Tasking Variants

NoOpenMP / NoTasking

```
void sbvh_build(const vector<Reference> &refs)
{
    if (create_leaf_p(refs)) return leaf_node();
    ObjectSplit object = object_cost(refs);
    SpatialSplit spatial = spatial_cost(refs);
    NodeSplit split;
    if (spatial.cost < object.cost)
        split = spatial_split();
    else
        split = object_split();
    sbvh_build(split.left_references);
    sbvh_build(split.right_references);
}
```

Tasking / Default

```
#pragma omp task
    sbvh_build(split.left_references);
#pragma omp task
    sbvh_build(split.right_references);
#pragma omp taskwait
```

Figure 2: High level overview of the SBVH construction algorithm by Stich et al., 2009, and how OpenMP tasking is applied to it.

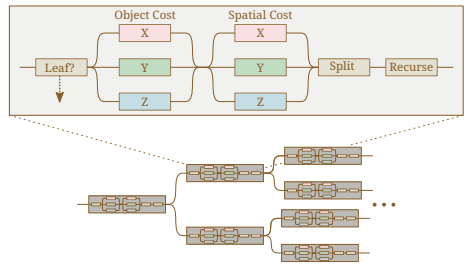


Figure 3: Visualization of how the parallel tasks are structured for our SBVH algorithm.

Axis Search Variants

NoOpenMP

```
SpatialSplit split_spatial()
{
  for (size_t ax = 0; ax < 3; ++ax)
  {
  }
```

ParallelFor

```
#pragma omp parallel for
for (size_t ax = 0; ax < 3; ++ax)
{
}
```

Tasks

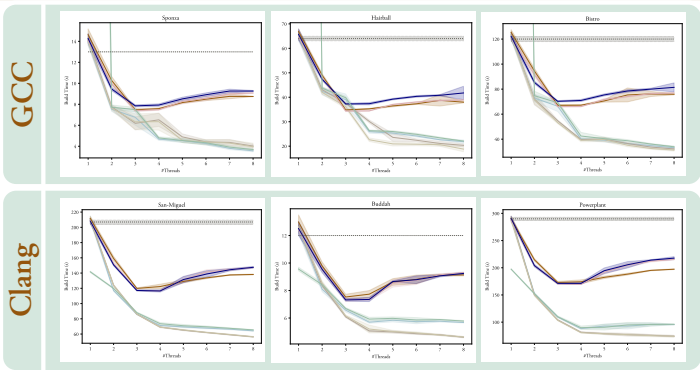
```
for (size_t ax = 0; ax < 3; ++ax)
#pragma omp task
{
}
#pragma omp taskwait
```

Taskloop

```
#pragma omp taskloop
for (size_t ax = 0; ax < 3; ++ax)
{
}
```

Figure 4: Beyond OpenMP tasking from figure 2, the object and spatial axis searches may also be accelerated, as each axis is independent. However, there are numerous ways to use OpenMP when tasking is already being applied elsewhere. Thus, the following OpenMP `#pragma` variants were applied to these to loop to find the best way. Thus, in total we have 8 different variations: With or without tasking, and 4 different ways to accelerate the inner axis `for`-loops.

Scaling Results



Rendering Times

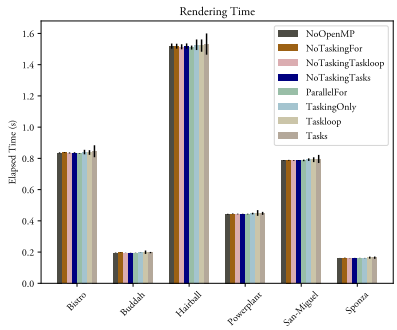


Figure 5: Estimated rendering times for each scene and variant.

Scenes



Sponza: 393 meshes, 262 267 triangles. Bistro: 1 mesh, 3 847 246 triangles. Hairball: 2 meshes, 2 880 002 triangles. Buddah: 1 mesh, 1 087 720 triangles. San-Miguel: 287 meshes, 9 980 699 triangles. Powerplant: 21 meshes, 12 759 246 triangles.

References

Dagum, L., & Menon, R. (1998). Openmp: An industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1), 46–55.

Stich, M., Friedrich, H., & Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. *Proceedings of the Conference on High Performance Graphics 2009*, 7–13. https://doi.org/10.1145/1572769.1572771

Real-Time Path-Tracing Of Ray-Tracing Benchmarks of Yore

Gustaf Waldemarson

Arm Sweden AB & Lund University, Faculty of Engineering (LTH)

Introduction

Ray-tracing has long been used as tool for generating realistic images, and has accumulated a number of different benchmarks over the years, primarily focusing on generating high-quality images. However, we are now moving into the era of real-time *animated* ray-tracing, thus, some old benchmarks that, at the time, did not garner much attention, can be a good starting point for trialing techniques that target animated content. Thus, in this work we take a look back at the *Benchmark for Animated Ray Tracing* [1] (BART) and adapts it to a world of real-time path-tracing with the help of the Spatiotemporal Variance-Guided Filtering (SVGF) [2].

Feature Summary

- A simple, but feature-complete implementation of the paper version of the SVGF [2] filter, with focus on clarity rather than performance.
- A novel (?) way of structuring the filter as a part of the SBT.
- Trowbridge-Reitz (GGX) surface shading model.
- Simple path-tracer for specular reflections, refractions and GI.
- Alternative rendering and filtering of e.g. ambient occlusion.

A Single Shader Pipeline

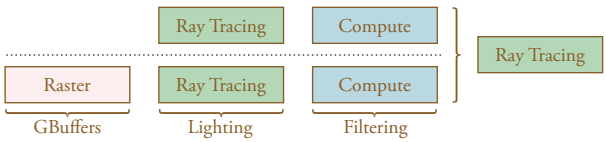


Figure 1: SVGF is typically used with multiple shader stages at once: Either with the rasterization, ray-tracing, and compute pipelines, or just the ray-tracing and compute pipelines. For this work, I wanted to see if I could do *everything* in the ray-tracing pipeline by (ab-)using the shader binding-table and potentially avoid pipeline flushes and related performance issues.

Shader Binding Table

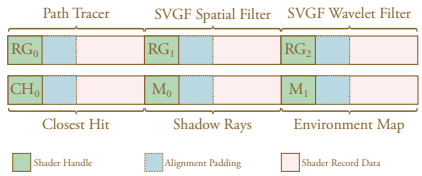


Figure 2: The Shader Binding Table (SBT) control both the rendering and filtering: The first `RayGen`-shader performs the primary rendering, and optionally, the temporal filtering, whereas the remainder perform the actual SVGF passes; the spatial and à trous wavelet filters.

Ray Tracing Pipeline

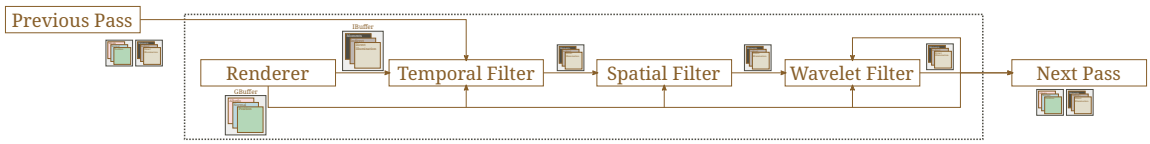
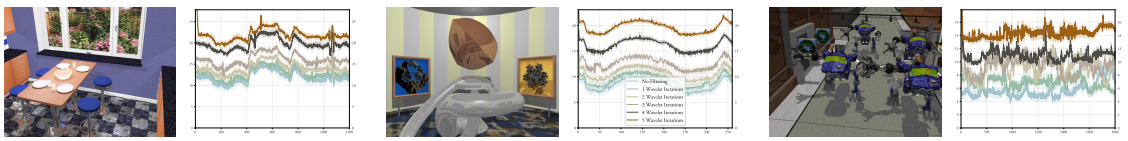


Figure 3: High-level overview of the ray-tracing and filtering pipeline: The renderer generates a Geometry Buffer (`GBuffer`) and Illumination Buffer (`IBuffer`) that is then filtered in the respective SVGF filter components.

Scenes and Performance Results



Kitchen Museum Robots

Figure 4: The BART [1] scenes and associated frametime results in milliseconds (ms) for each frame of the animation with either no filtering, or up to 5 iterations of the à trous wavelet filtering. Additionally, this works includes a `glTF` conversion of the original `AFF` scenes, available through the QR-codes, albeit limited to linear animations for the time being.

Additional Renderers and the History View

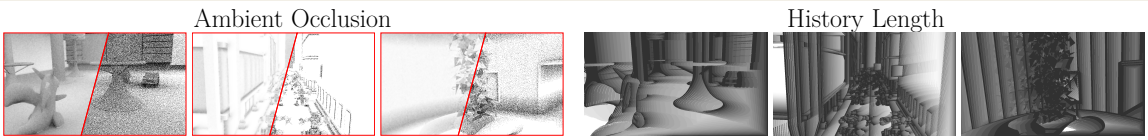


Figure 5: Alternative renderers supported by this framework, such as SVGF filtered Ambient Occlusion with a single sample ray per-pixel (left) and the currently accumulated history length of the SVGF filter (right), the latter of which is a very useful visual milestone while implementing the filter.

References

[1] J. Lext, U. Assarsson, and T. Moller. "A Benchmark for Animated Ray Tracing". In: *IEEE Computer Graphics and Applications* 21.2 (2001), pp. 22–31. DOI: 10.1109/38.909012.

[2] Christoph Schied et al. "Spatiotemporal Variance-Guided Filtering: Real-time Reconstruction for Path Traced Global Illumination". In: *ACM/EG Symposium on High Performance Graphics (HPG)*. July 2017, pp. 23–41. DOI: 10.1145/3105762.3105770.

Videos

LTH Color and Attribute Micromaps

Gustaf Waldemarson & Michael Doggett

Arm Sweden AB & Lund University, Faculty of Engineering (LTH)

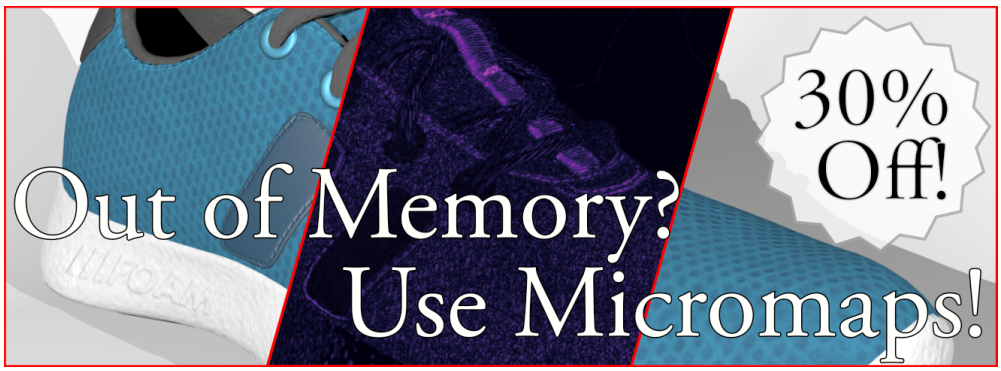


Figure 1: Micro-vertex micromaps for color, normal and metallic roughness attributes can provide similar rendering quality at 71 % the memory footprint at a negligible frame-time cost.

Introduction

In this work, we generalize the concept of opacity and displacement micromaps to arbitrary surface attributes, theoretically giving the hardware accelerated ray-tracing pipeline access to more generalized inputs for either direct use in shaders, or to extend the pipeline itself with entirely new operations, possibly removing the need for `AnyHit`-shaders.

What are Micromaps?

- Fixed micro-*triangle* opacity values through Opacity Micromaps [1].
- Variable micro-*vertex* displacement values through Displacement micromaps [2].

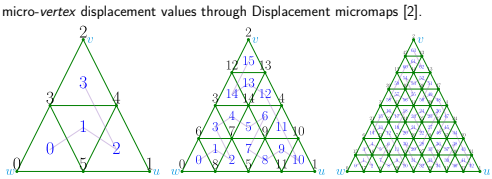


Figure 2: The first three subdivision levels with associated micro-triangles and micro-vertices.

Quality and Footprint

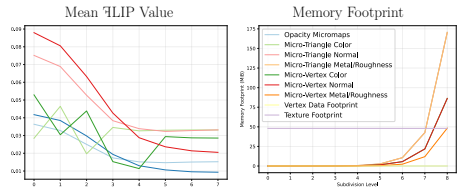


Figure 3: Mean ꟻLIP error and memory footprint (MB) at increasing subdivision levels.

Features and Use Cases

Micromap Mipmaps

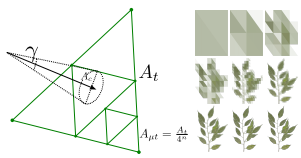


Figure 4: Well defined mipmap constructs for micromaps.

Fixed Ray Operations

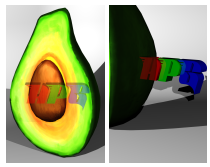


Figure 5: Operate on ray-payloads without `AnyHit` shaders.

GLSL Attributes

```
#extension GL_EXT_attribute_micromap : require

layout(binding=0, triangleMicromapEXT) buffer utri
{
    vec4 color;
} uTriAttributes;

layout(binding=1, vertexMicromapEXT) buffer uvtx
{
    vec3 normal;
} uVtxAttributes;
```

Figure 6: Example GLSL API for retrieving micromap attributes.

Subdivision Level Quality

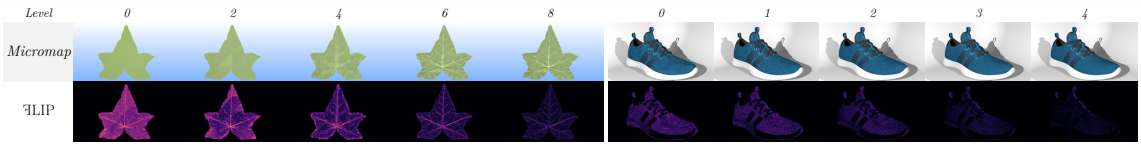


Figure 7: The micromap rendering quality depend heavily on the underlying geometry: Dense meshes can use a low subdivision level (Shoe), where sparse ones may need much higher one (Quad).

Scenes and Models



Quad | Avocado | Shoe | Flight-Helmet | Sponza | Sun-Temple

References

[1] Khronos Group. *VK_EXT_opacity_micromap Vulkan Extension Specification*. 2022. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html.

[2] Khronos Group. *VK_NV_displacement_micromap Vulkan Extension Specification*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_NV_displacement_micromap.html.



Department of Computer Science
Lund University
Box 118, SE-221 00 Lund, Sweden

ISBN (Printed): 978-91-8104-872-8
ISBN (Electronic): 978-91-8104-873-5
ISSN: 1404-1219
Dissertation 84, 2026
LU-CS-DISS 2026-01

Tryckeriet i E-huset, Lund 2026