Master of Science in Engineering: Game and Software Engineering
June 2020

Real-Time Ray Tracing With Polarization Parameters

Viktor Enfeldt

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The author declares that they are the sole author of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author:
Viktor Enfeldt
E-mail: vien15@student.bth.se
viktor.enfeldt@gmail.com

University advisor:
Dr. Prashant Goswami
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE−371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. The real-time renderers used in video games and similar graphics applications do not model the polarization aspect of light. Polarization parameters have previously been incorporated in some offline ray-traced renderers to simulate polarizing filters and various optical effects. As ray tracing is becoming more and more prevalent in real-time renderers, these polarization techniques could potentially be used to simulate polarization and its optical effects in real-time applications as well.
Objectives. This thesis aims to determine if an existing polarization technique from offline renderers is, from a performance standpoint, viable to use in real-time ray-traced applications to simulate polarizing filters, or if further optimizations and simplifications would be needed.
Methods. Three ray-traced renderers were implemented using the DirectX RayTracing API: one polarization-less Baseline version; one Polarization version using an existing polarization technique; and one optimized Hybrid version, which is a combination of the other two. Their performance was measured and compared in terms of frametimes and VRAM usage in three different scenes and with five different ray counts.
Results. The Polarization renderer is ca. 30% slower than the Baseline in the two more complex scenes, and the Hybrid version is around 5–15% slower than the Baseline in all tested scenes. The VRAM usage of the Polarization version was higher than the Baseline one in the tests with higher ray counts, but only by negligible amounts.
Conclusions. The Hybrid version has the potential to be used in real-time applications where high frame rates are important, but not paramount (such as the commonly featured photo modes in video games). The performance impact of the Polarization renderer's implementation is greater, but it could potentially be used as well. Due to limitations in the measurement process and the scale of the test application, no conclusions could be made about the implementations' impact on VRAM usage.

Keywords: ray tracing, polarizing filters, Mueller calculus, real-time, rendering

Sammanfattning

Bakgrund. Realtidsrenderarna som används i videospel och liknande grafikapplikationer simulerar inte ljusets polarisering. Polariseringsinformation har tidigare implementerats i vissa stålföljningsbaserade (*ray-traced*) offline-renderare för att simulera polariseringsfilter och diverse optiska effekter. Eftersom strålföljning har blivit allt vanligare i realtidsrenderare så kan dessa polariseringstekniker potentiellt också användas för att simulera polarisering och dess optiska effekter i sådana program.
Syfte. Syftet med denna rapport är att avgöra om en befintlig polariseringsteknik från offline-renderare, från en prestandasynpunkt, är lämplig att använda för att simulera polariseringsfilter i stålföljningsbaserade realtidsapplikationer, eller om ytterligare optimeringar och förenklingar behövs.
Metod. DirectX RayTracing API:et har använts för att implementera tre stålföljningsbaserade realtidsrenderare: en polarisationsfri *Baseline*-version; en *Polarization*-version med en befintlig polariseringsteknik; och en optimerad *Hybrid*-version, som är en kombination av de andra två. Deras prestanda mättes och jämfördes med avseende på *frametime* och VRAM-användning i tre olika scener och med fem olika antal strålar per pixel.
Resultat. *Polarization*-versionen är ca 30% långsammare än *Baseline*-versionen i de två mest komplexa scenerna, och *Hybrid*-versionen är ca 5–15% långsammare än *Baseline*-versionen i alla testade scener. *Polarization*-versionens VRAM-användningen var högre än *Baseline*-versions i testerna med högre strålantal, men endast med försumbara mängder.
Slutsatser. *Hybrid*-versionen har potential att användas i realtidsapplikationer där höga bildhastigheter är viktiga, men inte absolut nödvändiga (exempelvis de vanligt förekommande fotolägena i videospel). *Polarization*-versionens implementation hade sämre prestanda, men även den skulle potentiellt kunna användas i sådana applikationer. På grund av mätprocessens begränsningar och testapplikationens omfattning så kunde inga slutsatser dras gällande implementeringarnas påverkan på VRAM-användning.

Nyckelord: strålspårning, polariseringsfilter, Mueller-kalkyl, realtid, rendering

Acknowledgments

I would like to thank the following people for their contributions to this thesis: my supervisor Dr. Prashant Goswami, for his guidance and advice on how to improve the project, the report, and the presentation; everyone who read and gave feedback on the report, for their helpful comments and suggestions; Stefan Petersson, for the initial discussions about the project idea; and lastly, my partner Jessica, for her love and support throughout the course of this project and the past few years of my studies.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Nomenclature

Terms

dielectric Nonconductor of electricity; not a metal. 2, 9–11, 26–28, 41, 44

microfacet Very tiny mirror-like facet of the surface of an object. 19, 20, 24

Monte Carlo Computational method using repeated random sampling. 6, 17, 19, 41, 43

partially polarized Mixture of unpolarized and polarized light. 8, 9, 11

polarization state Direction of the oscillations in a transverse wave or group of waves. 2, 8–12, 14, 15, 17, 23

polarizing filter Optical filter that only lets waves with a specific polarization state pass through. Also called polarizer. 2–4, 10, 11, 13, 14, 17, 23, 24, 26–28, 41–44

primary ray Ray that is cast from the camera into the scene. 5, 6, 19, 22, 24, 25, 40

quasi-monochromatic Consisting of a very limited wavelength spectrum. 9, 11–13, 23

reference frame Plane that is perpendicular to a ray or wave's direction. 15, 24, 25, 41

unpolarized Consisting of randomly polarized waves. 8–11, 13, 15, 22–25, 27, 41

Abbreviations

API Application Programming Interface. 7, 19, 25

BRDF Bidirectional Reflectance Distribution Function. 19–21, 44

CPU Central Processing Unit. 19, 30

DOP Degree of Polarization. 12, 28

DXR DirectX Raytracing. 7, 22, 30

FPS Frames Per Second. 30

GPU Graphics Processing Unit. 3, 6, 19, 26, 29, 30

IOR Index of Refraction. 9, 10, 17, 26, 41

VRAM Video Random Access Memory. 3, 4, 30, 34, 35, 39–41

Symbols

$\hat{\mathbf{a}}$ Normalized vector.

α (*alpha*) The square of the material's roughness value. 20, 21

δ (*delta*) Phase retardation. 9, 10, 14, 17, 41, 44

η (*eta*) Complex refractive index. Defined as $\eta = n - \kappa i$. 9, 26

θ (*theta*) Angle of incidence; the angle between $\hat{\mathbf{v}}$ and $\hat{\mathbf{h}}$. 9, 10, 19, 20, 22, 24

κ (*kappa*) Extinction coefficient of the refractive index. 9, 10, 26

λ (*lambda*) Wavelength. 7, 8

$\mathbf{c_{diff}}$ Diffuse material color. 20

$\mathbf{c_{spec}}$ Specular material color. 10, 11, 26

E_x The x component of the electric field. 7, 8, 12, 24

E_y The y component of the electric field. 7, 8, 12

f Bidirectional reflectance function (BRDF). 19–21

F Fresnel reflectance function. 9, 10, 14, 17, 20–22, 25, 26, 44

$\hat{\mathbf{h}}$ Half vector between $\hat{\mathbf{l}}$ and $\hat{\mathbf{v}}$. 20, 21, 24, 25

I Radiant intensity. 11–13, 23

$K_\mathbf{d}$ Diffuse reflection coefficient. 19, 20, 22, 24

$K_\mathbf{r}$ Mirror reflection coefficient. 19–22, 24

$K_\mathbf{s}$ Specular reflection coefficient. 19–22, 24

$\hat{\mathbf{l}}$ Vector from the intersection point to the light source. 19–21, 25

$\mathbf{M}$ Mueller matrix. 13–15, 25

n Simple refractive index. 9, 10, 26

$\hat{\mathbf{n}}$ Surface normal at the intersection point. 10, 20, 21

p Degree of polarization (DOP). 12, 13, 26

$\vec{\mathbf{S}}$ Stokes vector. $\vec{\mathbf{S}} = (s_0, s_1, s_2, s_3)$. 13–15, 23

$\hat{\mathbf{v}}$ Vector from the intersection point to the viewer. 19–21, 24, 25

$\hat{\mathbf{x}}$ Local x-axis unit vector for a ray's or a Mueller matrix's reference frame. 24, 25

Chapter 1

Introduction

Shading and rendering techniques have developed over the years to simulate, or at least approximate, the visual appearance of real-world materials and how light interacts with them. A powerful rendering technique that is often used to create photorealistic visual effects in film and television productions is *ray tracing*. It uses virtual rays to simulate how light moves and interacts with objects in a scene, and can be used to achieve a multitude of visual effects (such as reflections and realistic soft shadows) [16]. A ray-traced image with some of those effects is shown in Figure 1.1. Notice how the square light source is visible in not just direct reflections, but also in secondary ones (e.g., the bright spot in the lower-left part of the white sphere is due to a prior reflection in the floor).

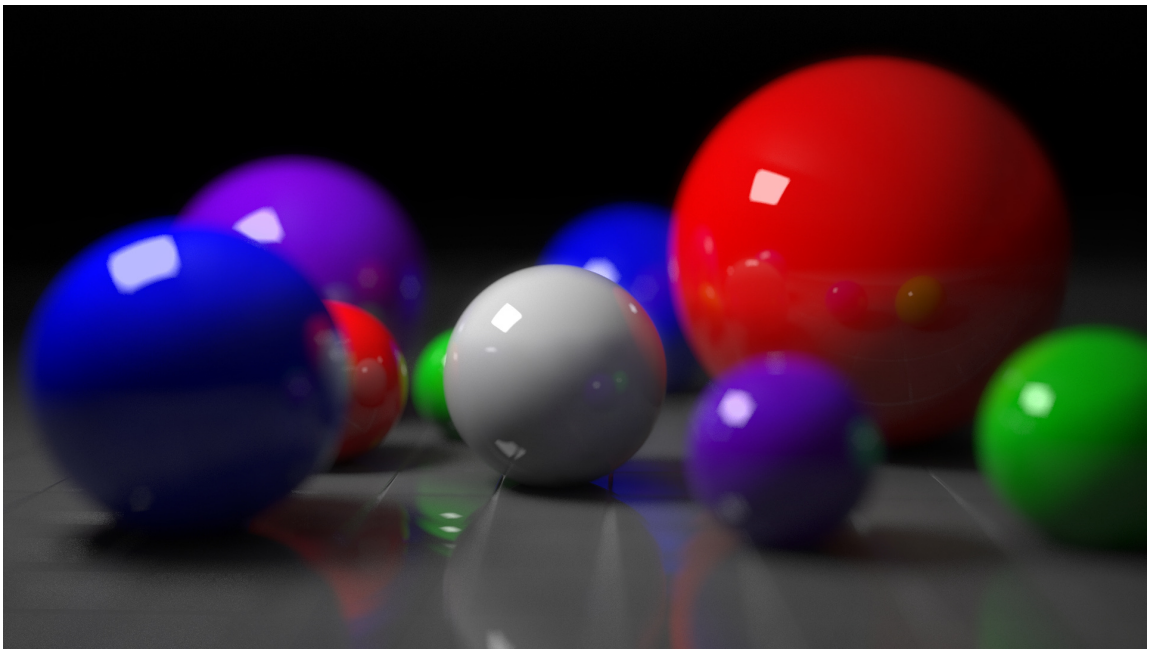


Figure 1.1: Image rendered with ray tracing and featuring effects such as global illumination, soft shadows, depth of field, and Fresnel reflections. *(Image from Wikipedia user Mimigu [24]).*

Ray tracing comes at a considerable computational cost, so it has historically not been used in complex real-time applications (such as video games) where cheaper, but more limiting, triangle rasterization techniques have been used instead [1, p. 415]. However, thanks to recent hardware and software advancements, ray tracing is now becoming increasingly common in higher budget computer games such as 4A Games' *Metro: Exodus* [3], Eidos-Montréal's *Shadow of the Tomb Raider* [15], and DICE's *Battlefield V* [28].

With these recent technological developments comes an opportunity to implement ray tracing features from offline renderers in real-time ones. One such feature is the simulation of light's *polarization state*: a property that describes the orientation of the electromagnetic field, and which can be impacted by interactions with various materials. The light that is reflected in *dielectric* (i.e., non-metallic) surfaces, for example, becomes partially or fully linearly polarized depending on the angle of incidence [7, p. 43]. Several calculi have been developed over the years to analyze polarization [5, p. 22.8], and a few of them have been implemented in *offline* ray-traced renderers to simulate numerous visual effects [26, 33, 35, 40]. However, they have (to the best of the author's knowledge) yet to be incorporated and tested in a *real-time* ray-traced renderer.

Figure 1.2: Photographs taken with (left) and without (right) a polarizing filter. In the version taken with a polarizing filter, the surface of the water appears to be significantly less reflective since the filter has prevented much of the reflected light from reaching the camera sensor. Note also how the color of the vegetation differs between the two photographs. *(Image from Dmitry Makeev [23]).*

The polarization of light generally has no directly noticeable impact on how our eyes perceive the light; however, it does have a significant impact on what the world looks like when viewed through a *polarizing filter* (which blocks out light that is not linearly polarized in the orientation of the filter). Photographers frequently use polarizing filters for various purposes: they can make glass and water surfaces appear more transparent, make foliage appear more colorful, and reduce the intensity of bright reflections (such as glare from sunlight). These effects are all difficult to replicate in software once a photograph is taken, yet they are easily obtainable by placing a polarizing filter in front of the camera lens and adjusting its rotation. An example of the visual impact this can have on a photograph is shown in Figure 1.2. In the left photograph (which was taken with a polarizing filter), the reflections in the water have been reduced significantly and made the stream bed more visible than it is in the unfiltered photograph on the right.

Many modern video games include a photo mode that allows the user to move around and take pictures within the game's virtual environments [4]. Often with several camera-like settings such as aperture, exposure, depth of field, focal length, and various filters. However, since the renderers that are used in games do not model the polarization state of light in their shading equations, polarizing filters have not yet been an option in these modes.

This thesis will examine if it is feasible, from a performance standpoint, to use an existing polarization technique in a real-time ray-traced renderer in order to simulate polarizing filters, or if that technique is too computationally expensive to use in real-time without further simplifications and optimizations.

1.1 Aim, Research Question, and Objectives

This research builds upon previous work about polarization ray tracing in *offline* renderers by, among others, Wolff and Kurlander [40], and which has been well summarized by Wilkie and Weidlich [38, 39]. The aim is to evaluate how feasible it is to simulate polarizing filters in a *real-time* ray-traced application (such as a video game) by using an existing polarization technique. The research question is:

RQ. "What is the performance impact of simulating polarizing filters and polarized light in a real-time ray-traced application?"

In order to answer it, the following objectives will be completed:

1. Create a **Baseline** real-time ray-traced renderer using conventional shading methods.
2. Create a **Polarization** renderer by duplicating the Baseline renderer and adding polarization parameters to it.
3. Create a **Hybrid** renderer that uses the same primary rays as the Polarization version, and the same reflection rays as the Baseline version.
4. Compare the performance of the Polarization and Hybrid renderers with the Baseline one in terms of frame times and video memory (VRAM) usage in scenes of varying geometrical complexity.

1.2 Delimitations

The study is limited to a single application that is tested on a single computer, and only using one kind of polarization ray tracing implementation. It does not investigate the performance impact a polarization implementation would have on different applications, nor does it investigate how different graphics processing units (GPUs) are affected by the implementation. The study only investigates the performance impact the implementations have in different scenes within the same application when running on the same hardware.

1.3 Contribution

When Wolff and Kurlander [40] first added polarization parameters to an *offline* ray-traced renderer in 1990, the rendering time of the polarized version approached that of the unmodified one in complex scenes, but was approximately doubled in simpler scenes. Rendering hardware and software have changed and improved significantly since then; with *real-time* ray tracing now being in its infancy, this thesis shows what performance impact polarization parameters have on a modern GPU with hardware-accelerated ray tracing.

This thesis concludes that implementing an existing polarization technique has a non-trivial impact on the rendering time, but that this impact is small enough for the technique to potentially be of use in some real-time renderers where the effects of a *polarizing filter* are desirable. As the shading calculations involved are rather complex, it also highlights the potential for more efficient approximate techniques (similar to those used in other shading methods) to be developed in the future.

1.4 Societal, Ethical, and Sustainability Aspects

This thesis is not expected to have an impact on any societal or ethical aspects since it is limited to researching graphical improvements to computational models of light. It does not involve any test subjects and is not expected to have any consequences outside of the field of computer graphics.

The author acknowledges potential sustainability-related issues surrounding the production and use of computer hardware; however, such systematic issues are deemed to be beyond the responsibility of the author and the scope of this project. The environmental impact of this research is limited to the power consumption associated with the desktop computer that was used for the development and testing, and the production of the graphics card that was purchased to run the tests.

1.5 Thesis Outline

Chapter 2 explains some basic information and theory about ray tracing, the polarization of light, Fresnel reflectance, polarizing filters, and the Stokes-Mueller calculus. This information is needed to understand the polarization-capable renderers.

Chapter 3 touches upon some related work from the area of polarization rendering.

Chapter 4 describes the three renderers that have been implemented in the test application. It details their shading models, their rendering algorithms, and how the polarization theory from chapter 2 has been implemented in the polarization-capable versions. Section 4.8 closes out this chapter by demonstrating the visual impact polarizing filters have in the two polarization-capable implementations.

Chapter 5 details the test scenes and settings that were used to evaluate the performance of the three renderers, how the tests were carried out, and the computer that was used to run the tests.

Chapter 6 presents and analyses the resulting frametime and VRAM usage measurements from the performance tests.

Chapter 7 discusses the implications of the results, some limitations of the tests, and some considerations for future implementations.

Chapter 8 concludes the thesis by answering the research question and suggesting some topics for future research in this area.

Chapter 2

Basic Concepts

Some information about the underlying theory behind the rendering implementations is needed to understand their context and how they function. Those basic concepts will be presented in this chapter, and later referred to from chapter 4's implementation details.

2.1 Ray Tracing

Ray tracing is the process of tracing virtual rays of light as they scatter around a scene and then using the information gathered by them to render an image of that scene. The most true-to-nature approach to it would be to spawn rays at each light source in the scene and have them reflect and refract in the objects they hit until they eventually make their way to the viewer. This approach would be highly inefficient since the vast majority of light rays in a scene will not hit the viewer and, therefore, do not contribute to the final image.

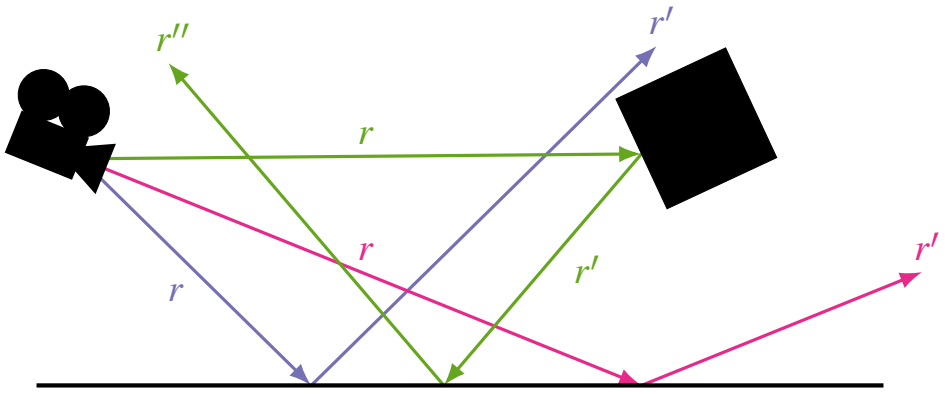


Figure 2.1: Ray tracing from the view-point. Primary rays r are cast from the view-point to the center of each pixel. Recursive reflection rays (r', r'', etc.) are then cast from the primary rays' and the subsequent reflections rays' intersection points. Shadow and refraction rays have been left out for simplicity, and they will not be used in this project.

The approach that is used instead is to trace rays from the viewer and into the scene (see Figure 2.1 for a visualization). This was first introduced in 1969 by Appel [2] and has since been developed into a few different variations. Appel's technique casts one ray per pixel (called a primary ray) into the scene and finds the first object that's blocking the ray (which is a process called *ray casting**). Shadow rays are then cast from the pixel's intersection point to the lights in the scene. If a shadow ray is obstructed, then its target light does not

*a term that is sometimes also used to refer to Appel's rendering algorithm as a whole

contribute to the shading of the pixel. Algorithm 2.1 describes the general algorithm that is used when ray tracing from the view-point.

Algorithm 2.1: Ray tracing rendering loop.

```
for each pixel (ray) ∈ output image do
    for each triangle ∈ scene do
        if triangle is ray's closest hit then
            SHADEPIXEL()
```

Whitted-style ray tracing* was developed by Whitted in 1979 and built upon Appel's technique by adding recursive rays to render reflections and refractions [36]. Each intersection point spawns a reflection and a refraction ray, as well as rays to all light sources. This process is repeated recursively, creating a tree of rays for each pixel, until none of the new rays intersect any object or some depth- or ray-limit is reached. The color of each pixel is determined by the primary ray, which gathers data from all of its recursive rays.

Almost all modern ray-tracers are based on some variation of recursive *Monte Carlo* ray tracing [16, p. 7], such as Kajiya's *path tracing* [20] which was partially based on Cook et al.'s *distribution ray tracing*† [10]. Recursive Monte Carlo ray tracing uses random sampling of possible reflection and light directions. This enables the rendering of soft shadows, depth of field, and fuzzy reflections. It also makes *global illumination* possible by taking the light contribution from *all* objects in the scene into account, instead of from just the light sources. The naive implementations of Monte Carlo ray tracing require many rays per pixel to get an image that does not have a considerable amount of noise; therefore, real-time applications need to use denoising techniques in order to achieve similar quality results with much fewer rays [16, p. 287].

2.1.1 Comparison With Rasterization

Real-time applications have historically used rasterization techniques instead of ray tracing due to the latter's computational complexity [1, p. 415]. As described in Algorithm 2.1, ray tracing iterates over pixels (rays) first and triangles second. Rasterization, on the other hand, instead iterates over triangles first and pixels second (as is described in Algorithm 2.2).

Algorithm 2.2: Rasterization rendering loop.

```
for each triangle ∈ scene do
    for each pixel ∈ output image do
        if triangle is closest then
            SHADEPIXEL()
```

Both rasterization and ray tracing are so-called "embarrassingly parallel" algorithms (i.e., they can easily be split into many independent tasks), so in practice their outermost loops are computed in parallel across multiple threads and cores on the GPU, instead of being iterated over sequentially as they would in a single-threaded process. The theoretical complexity of the two algorithms do not differ much when optimizations are taken into

*also called *classical ray tracing*

†which is occasionally referred to as *stochastic ray tracing*

account; however, the computations involved in the ray tracing process are much more time-consuming than the ones in the rasterization process, thereby making ray tracing significantly slower than rasterization.

2.1.2 Hardware Acceleration

Specialized hardware has been developed to speed up the expensive ray tracing–related calculations. In 2018, Nvidia debuted the first consumer graphics cards (the RTX 20 series) that have hardware acceleration for ray tracing. Together with the simultaneous release of Microsoft's DirectX RayTracing (DXR) feature to the Direct3D API, this has made ray tracing a viable (albeit still considerably slower) alternative to rasterization for certain effects in real-time applications. At the time of writing DXR has been used in several higher budget games: it is used to create global illumination in 4A Games' *Metro: Exodus* [3], shadows in Eidos-Montréal's *Shadow of the Tomb Raider* [15], and reflections in DICE's *Battlefield V* [28].

2.2 Light

Visible light is a subsection of the electromagnetic spectrum that represents electromagnetic radiation at wavelengths in the range of 400 to 700 nm. It can be generalized as a collection of plane electromagnetic waves, each consisting of synchronized oscillations in the electric and the magnetic field. Those plane waves are *transverse waves*, since their wave components' oscillations are perpendicular to each other and to the direction of propagation. A visualization of an electromagnetic plane wave is shown in Figure 2.2.

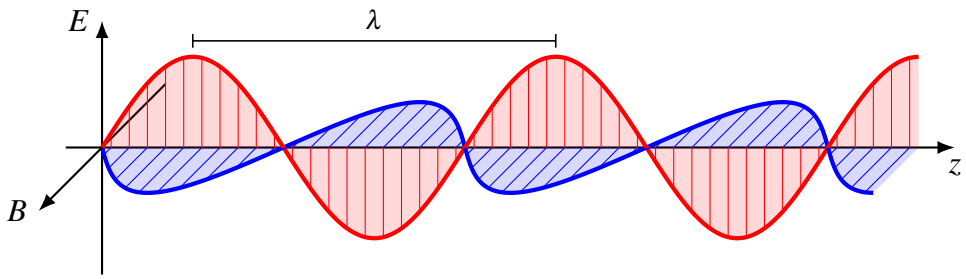
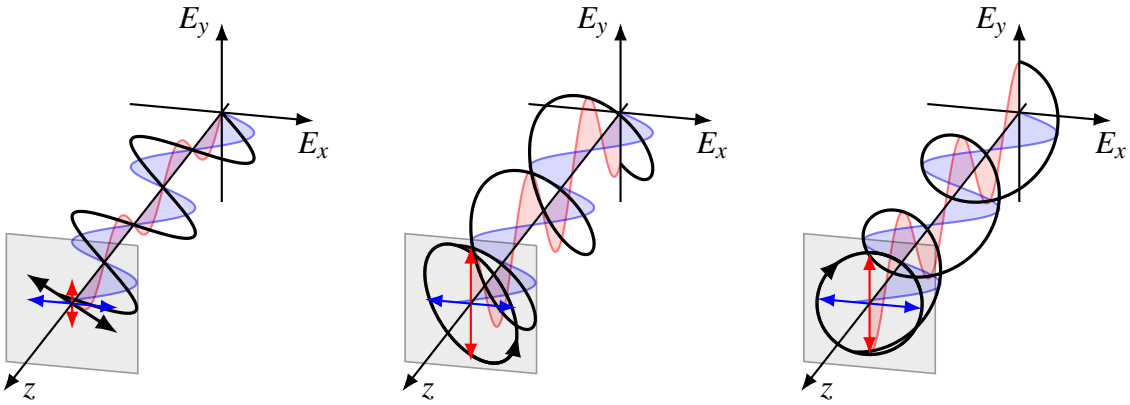


Figure 2.2: A plane electromagnetic wave visualized as a transverse wave consisting of an electric (red) and a magnetic (blue) wave. E is the electric field component, B is the magnetic field component, z is the direction of propagation, and λ is the wavelength.

For modeling purposes, a photon can be seen as a combination of two parallel plane electromagnetic waves that have the same wavelength—but not necessarily the same phase or rotation. This combined electromagnetic wave can, in turn, be represented by a transverse wave of the oscillations in the perpendicular electric field components E_x and E_y, both of which are perpendicular to the wave's direction of propagation z. Since the two electromagnetic waves have the same constant wavelength, the oscillations in the field components E_x and E_y will also have the same constant wavelength as each other, and the wave is therefore said to be *monochromatic*.

2.2.1 Polarization



(a) Linear (b) Left-hand elliptical (c) Right-hand circular

Figure 2.3: Three polarized waves with identical wavelengths. The wave is linearly polarized if the wave components' phase angles are equal (i.e., if their peaks coincide) and elliptically polarized if there is a phase difference and both components have non-zero amplitudes. Circular polarization is a special case of elliptical polarization, which occurs when the amplitudes are equal and the phase difference is exactly $\pm\lambda/4$.

When looking at a plane that is orthogonal to a transverse wave's direction, the polarization state of the wave specifies the pattern created by tracing the tip of the wave vector in that plane as the wave moves in its direction of propagation. Since photons are monochromatic, the waves in the electric field components E_x and E_y are inherently synchronized and the pattern (i.e., the polarization state) will not change as the photon moves through a perfect vacuum. Visualizations of three different polarization states of photons are shown in Figure 2.3.

The angle of the linear polarization depends on the difference between the two electric field components E_x and E_y's oscillation magnitudes (see Figure 2.3a). Likewise, elliptical polarization is perfectly circular if the magnitudes are equal and the phase difference is exactly $\pm 90°$ (i.e., $\pm\lambda/4$). Elliptical polarization is considered to be either right- or left-handed depending on the field's rotation when viewed against the wave's direction of propagation.

2.2.2 Light Rays and Their Polarization States

For modeling purposes, a *light ray* is a representation of the photons that flow along a beam of light's direction. These photons have the same direction and speed (since the speed of light is only dependent on the medium), but they do not necessarily have the same wavelength or polarization state.

If all photons in a light ray have the same polarization state, then the light ray is said to have that same polarization state as well; a light ray which consists of photons with randomized polarization states is said to be *unpolarized* (which is generally the case with spectrally and spatially averaged sunlight [14, p. 15]); and if some polarization states are more common than others, then the ray is said to be *partially polarized*.

In non-spectral renderers (such as the ones used in video games), all light values are approximated by combinations of three color components (red, green, and blue). To model the color components as light waves, they can be seen as independent *quasi-monochromatic* light waves. A quasi-monochromatic light wave contains photons of more than one unique wavelength, but only of wavelengths within a minimal spectrum that is much smaller than the wave's mean wavelength. Unlike monochromatic light, quasi-monochromatic light can be unpolarized; its polarization state can be modeled as the sum of one completely polarized and one completely unpolarized light wave [7, p. 551].

2.3 Fresnel Reflectance

When unpolarized light is reflected in a dielectric (i.e., non-metallic) material, it becomes fully or partially polarized unless the surface is perfectly diffuse or the angle of incidence is zero. If the surface is perfectly smooth, then this behavior can be precisely described by the Fresnel* terms [40, p. 47]. The Fresnel terms are a function of the material's complex index of refraction (IOR) η and the angle of incidence θ. The value of η is dependent on the wavelength and is defined as $\eta = n + \kappa i$, with $n > 0$ being the simple IOR and $\kappa \geq 0$ being the extinction coefficient.† By definition, κ is zero for dielectric materials [29, p. 13].

For the reflection of a monochromatic or quasi-monochromatic wave at the angle of incidence θ in an optically isotropic (i.e., same properties regardless of rotation) material with the IOR η, the Fresnel terms $F_\perp$, $F_\parallel$, $\delta_\perp$, and $\delta_\parallel$‡ are defined as

$$\begin{aligned}
F_\perp(\theta,\eta) &= \frac{a^2+b^2-2a\cos\theta+\cos^2\theta}{a^2+b^2+2a\cos\theta+\cos^2\theta}\,,\\
F_\parallel(\theta,\eta) &= \frac{a^2+b^2-2a\sin\theta\tan\theta+\sin^2\theta\tan^2\theta}{a^2+b^2+2a\sin\theta\tan\theta+\sin^2\theta\tan^2\theta}F_\perp(\theta,\eta)\,,\\
\delta_\perp(\theta,\eta) &= \arctan\left(\frac{2b\cos\theta}{\cos^2\theta-a^2-b^2}\right),\\
\delta_\parallel(\theta,\eta) &= \arctan\left(\frac{2\cos\theta\left((n^2-\kappa^2)b-(2n\kappa)a\right)}{(n^2+\kappa^2)^2\cos^2\theta-a^2-b^2}\right),
\end{aligned} \tag{2.1}$$

where a and b are given by the equations

$$\begin{aligned}
2a^2 &= \sqrt{(n^2-\kappa^2-\sin^2\theta)^2+4n^2\kappa^2}+n^2-\kappa^2-\sin^2\theta\,,\\
2b^2 &= \sqrt{(n^2-\kappa^2-\sin^2\theta)^2+4n^2\kappa^2}-n^2+\kappa^2+\sin^2\theta\,.
\end{aligned} \tag{2.2}$$

The first pair of equations $F_\perp$ and $F_\parallel$ determine the amount of the incoming light that is reflected in the electromagnetic fields that lie perpendicular to and parallel with the plane of incidence. Figure 2.4 shows how, when light is reflected in a dielectric surface at the angle known as Brewster's angle, $F_\parallel$ becomes zero and the reflected light is consequently

*pronounced Freh-*nel*

†The notation for complex and simple refraction indexes does not appear to be standardized; some texts might refer to n as the complex IOR and η as the simple one. The same symbol, n or η, is sometimes used for both terms with a marker (such as an overline) denoting the complex one. This paper will use the notation $\eta = n + \kappa i$ that is used by Wilkie and Weidlich [38, 39].

‡some literature use the subscripts s and p instead of $\perp$ and $\parallel$

fully linearly polarized in the direction perpendicular to the plane of incidence P [7, p. 43]. This is of significance because it means that a polarizing filter will be able to filter out this reflected light (more on this in section 2.4).

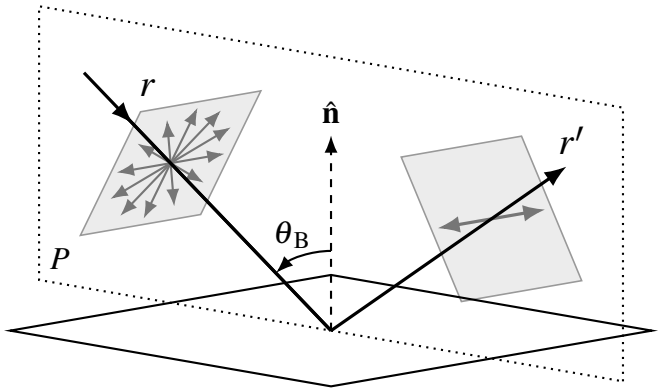


Figure 2.4: Interaction between a ray of unpolarized light and a dielectric surface at Brewster's angle. At the angle of incidence θ_B, the reflection ray r' becomes fully linearly polarized in the direction perpendicular to the plane of incidence P (which contains both the surface normal $\hat{\mathbf{n}}$ and the incident ray r). The refraction ray has been left out for simplicity.

The second pair of equations $\delta_\perp$ and $\delta_\parallel$ describe the phase retardation the perpendicular and parallel wave components undergo in the plane of incidence [40, p. 49]. This matters in a polarization renderer because (as mentioned in subsection 2.2.1) the angle, handedness, and shape of a wave's polarization are all affected by the phase offset. For reflections in metallic surfaces, $\delta_\perp$ and $\delta_\parallel$ are not always equal, and this can turn linearly polarized incident light into elliptically polarized reflected light.

2.3.1 Implementation in Non-Polarizing Real-Time Renderers

Fresnel reflectance is important in non-polarizing renderers as well, since it is what makes surfaces more reflective at grazing angles (i.e., when the angle of incidence θ is near 90°). If information about the reflection's polarization state is not of interest, then it is sufficient to calculate the average value $F = \frac{F_\perp + F_\parallel}{2}$ to get this effect. The δ Fresnel terms are not relevant in non-polarizing renderers, since they only affect the shape of the polarization and not the intensity of the light.

In real-time renderers, F is typically computed with Schlick's approximation [27]. It is defined using the reflection coefficient R_0 of incoming light that is parallel to the surface normal (i.e., $\theta = 0°$):

$$F_{\text{Schlick}} = R_0 + (1 - R_0)(1 - \cos\theta)^5 . \tag{2.3}$$

For reflections off of materials in air, R_0 can be calculated from the material's IOR values as

$$R_0 = \frac{(n-1)^2 + \kappa^2}{(n+1)^2 + \kappa^2} . \tag{2.4}$$

In rendering engines, R_0 is usually referred to as the specular color $\mathbf{c}_{\text{spec}}$ and is extracted from a material's texture files. The Falcor rendering framework that is used in this project calculates $\mathbf{c}_{\text{spec}}$ from a base color $\mathbf{c}_{\text{base}}$ and a metalness value m in the textures as

$$\mathbf{c}_{\text{spec}} = (1 - m)0.04 + m\,\mathbf{c}_{\text{base}} . \tag{2.5}$$

It is always in the range $[0.04, 1]$ since the reflection coefficient of dielectric materials is usually between 0.02 and 0.08. The metalness value m is usually binary (0 for dielectrics, 1 for metals), so $\mathbf{c}_{\text{spec}}$ will be equal to 0.04 for dielectrics and be set by the $\mathbf{c}_{\text{base}}$ texture for metals.

2.4 Polarizing Filters

A linear polarizing filter (also called polarizer) is an optical filter that produces linearly polarized light from unpolarized light. It will also block some of the light, which means that the incoming light's irradiance will be greater than the outgoing light's irradiance. This loss of brightness is described by Malus's law [14, p. 213], which states that the intensity I that passes through a perfect linear polarizing filter is

$$I = I_0 \cos^2 \phi \,, \tag{2.6}$$

where I_0 is the intensity of the incoming linearly polarized light, and ϕ (*phi*) is the angle between the filter's polarizing angle and the incoming light's angle of polarization. If the incoming light is unpolarized, and thus consists of light polarized in all angles, then the irradiance will be cut in half (since the average value of $\cos^2 \phi$ is $1/2$). Therefore, the exposure settings on a camera with a polarizing filter need to be changed to get an image that is as bright as it would have been without the filter; in a renderer, this is easily achieved by doubling the light's intensity after it has passed through the filter.

If the incoming light is completely linearly polarized at an angle of exactly $\pm 90°$ in relation to the filter, then all of the light will be blocked (since $\cos^2(\pm 90°) = 0$). This effect can be seen by placing a polarizing filter in front of another one and rotating it 90°, or by rotating a single filter in front of an LCD screen (which produces linearly polarized light).

The polarizing filters used by photographers typically consist of a linear polarizer followed by a quarter-wave plate that transforms the linearly polarized light into circularly polarized light. This combination is called a *circular polarizing filter* (usually abbreviated as CPL on the filter) and is used to prevent images from potentially getting over- or underexposed as they might with a linear polarizing filter. This is due to metering systems relying on partially reflecting mirrors that, as described by the Fresnel terms, do not reflect the same amount of linearly polarized light regardless of orientation. If the intensity is already known (as it is in a renderer), then there is no need for the quarter-wave plate, and a simpler linear polarizing filter can be used.

2.5 Polarization Calculus

To simulate the polarization state of light in a renderer, one must have a mathematical model to describe it. Several mathematical formalisms to describe and analyze the polarization state of monochromatic (and quasi-monochromatic) light have been developed over the years. Some of them, such as the Jones calculus [19], are limited to modeling only fully polarized light and are therefore not of interest to computer graphics applications. The most prominent ones that can model fully polarized, partially polarized, as well as unpolarized light are the coherency matrix and Stokes vector formalisms.

In the opinion of Wilkie and Weidlich [39, p. 19], the Stokes vector–based Mueller calculus* is the most convenient model of polarization to use in computer graphics. They argue that this is the case because the polarization-describing Stokes parameters do not involve any complex numbers and are more intuitively obvious than the coherency matrices. The Mueller calculus can not represent absolute phase, which the Jones calculus can and which is needed for certain calculations involving coherent light (i.e., light in which all photons have the same direction, wavelength, phase, and polarization state); however, this quantity does not appear to play any notable role in conventional graphics applications.

The Mueller calculus was developed by Hans Müller in the early 1940s. It involves the multiplication of Stokes vectors with Mueller matrices, and for those multiplications to work correctly: the rotation of the light rays needs to be tracked as well (which is detailed in section 2.5.3). For a more comprehensive introduction, see Shurcliff [30, Ch. 8].

2.5.1 Stokes Parameters

The Stokes parameters can be used to describe the polarization state of any monochromatic or quasi-monochromatic light ray. They were first described in 1852 by George Gabriel Stokes [31], and are defined as the four real values s_0, s_1, s_2, and s_3:

$$\begin{aligned} s_0 &= I\,, \\ s_1 &= Ip\cos(2\psi)\cos(2\chi)\,, \\ s_2 &= Ip\sin(2\psi)\cos(2\chi)\,, \\ s_3 &= Ip\sin(2\chi)\,, \end{aligned} \tag{2.7}$$

where $p \in [0,1]$ is the degree of polarization (DOP), ψ (*psi*) and χ (*chi*) are the orientation and ellipticity angles in the polarization ellipse (as shown in Figure 2.5), and I is the intensity of the light.

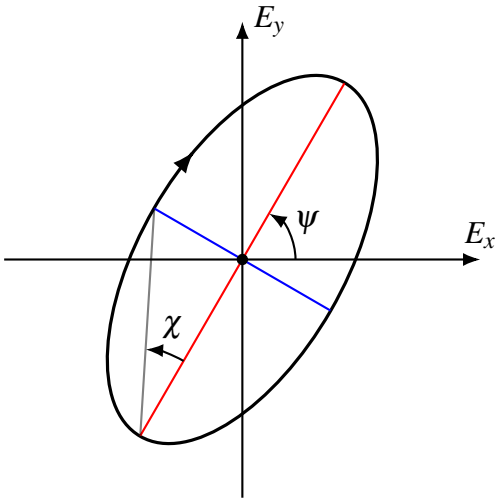


Figure 2.5: Polarization ellipse showing the relation between the orientation angle ψ, the ellipticity angle χ, and the shape of the polarization. The direction of propagation z is pointing towards the viewer.

*sometimes also referred to as the *Müller calculus* after the original German spelling of the name

The Stokes parameters are typically grouped together in a Stokes vector $\vec{\mathbf{S}}$, which is defined as

$$\vec{\mathbf{S}} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} I \\ Q \\ U \\ V \end{bmatrix} = \begin{bmatrix} \leftrightarrow + \updownarrow \\ \leftrightarrow - \updownarrow \\ \nearrow - \searrow \\ \circlearrowright - \circlearrowleft \end{bmatrix}, \tag{2.8}$$

where I is the aforementioned intensity of the light, Q and U describe the angle of linear polarization, and V is the handedness of the elliptical polarization (when viewed against the light's direction of propagation).* Right-handed polarization is indicated by $V > 0$, and left-handed by $V < 0$.

The Stokes vector that describes completely unpolarized quasi-monochromatic light is $\vec{\mathbf{S}} = (I_c, 0, 0, 0)$, where I_c is the same intensity value that would be used for that color channel in a non-polarizing renderer.

Properties of the Stokes Parameters

The degree of polarization p can be calculated from the Stokes parameters as

$$p = \frac{\sqrt{s_1^2 + s_2^2 + s_3^2}}{s_0}, \tag{2.9}$$

with $p = 1$ describing fully polarized light and $p = 0$ describing completely unpolarized light. Since $p \in [0, 1]$, the first Stokes parameter s_0 is bound by the inequality

$$s_0 \geq \sqrt{s_1^2 + s_2^2 + s_3^2}. \tag{2.10}$$

Likewise, s_1, s_2, and s_3 are all constrained to the range $[-s_0, s_0]$.

2.5.2 Mueller Matrices

Mueller matrices are used to model how light rays' Stokes vectors are altered by interactions with various samples (e.g., when the light is reflected off of a surface or passes through a polarizing filter). They are defined as real-valued 4×4 matrices:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}. \tag{2.11}$$

Mueller matrices can be used to model various polarization elements [5, Sec. 22.17]. For the purposes of this thesis, the matrices of interest are the ones that describe Fresnel reflectance and linear polarizing filters.

*In Stokes' original notation, the modern form of the Stokes vector $\vec{\mathbf{S}} = (s_0, s_1, s_2, s_3)$ would be written as $\vec{\mathbf{S}} = (A, C, D, B)$.

Mueller Matrix for Fresnel Reflectance

The Mueller matrix for the Fresnel reflectance is defined as

$$\mathbf{M}_{\text{Fresnel}} = \begin{bmatrix} A & B & 0 & 0 \\ B & A & 0 & 0 \\ 0 & 0 & C & S \\ 0 & 0 & -S & C \end{bmatrix}, \tag{2.12}$$

where

$$\begin{aligned} A &= \frac{F_\perp + F_\parallel}{2}, \\ B &= \frac{F_\perp - F_\parallel}{2}, \\ C &= \cos(\delta_\perp - \delta_\parallel)\sqrt{F_\perp F_\parallel}, \\ S &= \sin(\delta_\perp - \delta_\parallel)\sqrt{F_\perp F_\parallel}, \end{aligned} \tag{2.13}$$

and $F_\perp$, $F_\parallel$, $\delta_\perp$, and $\delta_\parallel$ are the Fresnel terms from Equation 2.1 [39, p. 23; 14, p. 178].

Mueller Matrix for Polarization Filters

The Mueller matrices for perfect horizontal and perfect vertical linear polarizing filters (called $\mathbf{M}_{\text{lph}}$ and $\mathbf{M}_{\text{lpv}}$ respectively) are defined as

$$\mathbf{M}_{\text{lph}} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{M}_{\text{lpv}} = \frac{1}{2}\begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{2.14}$$

Other angles for linear polarizing filters can be achieved by rotating the matrices with two rotation matrices. The Mueller matrix for a horizontal linear polarizing filter that has been rotated clockwise by the angle ϕ is given by the equation

$$\mathbf{M}_{\text{lp}\phi} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(2\phi) & -\sin(2\phi) & 0 \\ 0 & \sin(2\phi) & \cos(2\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{M}_{\text{lph}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(2\phi) & \sin(2\phi) & 0 \\ 0 & -\sin(2\phi) & \cos(2\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.15}$$

which can be rewritten as the single matrix

$$\mathbf{M}_{\text{lp}\phi} = \frac{1}{2}\begin{bmatrix} 1 & \cos(2\phi) & \sin(2\phi) & 0 \\ \cos(2\phi) & \cos^2(2\phi) & \cos(2\phi)\sin(2\phi) & 0 \\ \sin(2\phi) & \cos(2\phi)\sin(2\phi) & \sin^2(2\phi) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{2.16}$$

2.5.3 Operations on Stokes Vectors and Mueller Matrices

To alter the polarization state of a Stokes vector with a Mueller matrix, the two simply need to be multiplied. After a light ray with the polarization state $\vec{\mathbf{S}}$ has interacted with a sample

described by the Mueller matrix $\mathbf{M}$, the new polarization state $\vec{\mathbf{S}}'$ is calculated as

$$\vec{\mathbf{S}}' = \mathbf{M}\vec{\mathbf{S}} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}. \tag{2.17}$$

Two rays of light which are moving along the same ray in space can be combined by adding their Stokes vectors together. The new combined Stokes vector $\vec{\mathbf{S}}$ is calculated as the sum of the two light rays a and b's Stokes parameters:

$$\vec{\mathbf{S}} = \vec{\mathbf{S}}_a + \vec{\mathbf{S}}_b = \begin{bmatrix} s_{a_0} \\ s_{a_1} \\ s_{a_2} \\ s_{a_3} \end{bmatrix} + \begin{bmatrix} s_{b_0} \\ s_{b_1} \\ s_{b_2} \\ s_{b_3} \end{bmatrix} = \begin{bmatrix} s_{a_0} + s_{b_0} \\ s_{a_1} + s_{b_1} \\ s_{a_2} + s_{b_2} \\ s_{a_3} + s_{b_3} \end{bmatrix}. \tag{2.18}$$

Since Stokes vectors can be rotated at some arbitrary angle around their direction of propagation, the above mathematical operations are only permissible if the orientations of the two terms' local coordinate systems are aligned. However, if one of the Stokes vectors is either fully circularly polarized or completely unpolarized, then the alignment is irrelevant since those states are unaffected by rotations.

Reference Frame Tracking

The rotation of a Stokes vector is represented by reference frames, which can be visualized as the orientation of a plane that is perpendicular to the light wave's direction; Mueller matrices have both an entry and an exit frame. For the multiplication of a Stokes vector with a Mueller matrix to be permissible, the Stokes vector's reference frame has to match the Mueller matrix's entry frame.

In order to make reference frames match, a Stokes vector can simply be rotated. The components of rotated Stokes vector $\vec{\mathbf{S}}'$ are calculated as

$$\begin{aligned} s'_0 &= s_0\,, \\ s'_1 &= \cos(2\phi)s_1 + \sin(2\phi)s_2\,, \\ s'_2 &= -\sin(2\phi)s_1 + \cos(2\phi)s_2\,, \\ s'_3 &= s_3\,, \end{aligned} \tag{2.19}$$

where ϕ is the angle between the old and the new reference frame. Note how the rotation does not change the intensity of the light ($s'_0 = s_0$) or the handedness of the polarization ($s'_3 = s_3$), it only changes the orientation of the linear polarization (s'_1 and s'_2) so that the angle relates to the orientation of the new reference frame.

Chapter 3

Related Work

Several mathematical models have been created over the years to simulate light's polarization state and its interactions with various media. A few new models have been developed in the past five years to increase their physical accuracy in order to develop more accurate microscopes and polarimetric cameras [21, 25, 41]. Polarimetric cameras are used to measure target polarization, and it is therefore important that the cameras themselves do not alter the polarization of the measured light.

Wolff and Kurlander [40] were the first to implement polarization parameters in a ray-traced renderer and used it to simulate polarizing filters and more realistic reflections. They did so using the same formulation of the Fresnel functions used in this thesis, but with the coherency matrix formalism to represent polarization instead of the Stokes-Mueller one. In order to reduce the amount of shading computations they used pre-computed lookup tables for the values of $F_\perp$, $F_\parallel$, and δ for each combination of material (i.e., complex IOR) and wavelength (i.e., red, green, and blue) in each scene. If texture mapping is used, as it is in modern renderers, then each object could potentially consist of thousands of unique materials, resulting in thousands of lookup tables for just a single object if this approach were to be used.

Wilkie and Weidlich [37] have proposed a few standardized visualizations of the polarization state to assist in the development of polarizing renderers and to help assure their correctness. The visualizations are based on the Stokes parameters but can also be adapted for coherency matrices since the two formalisms are interchangeable (as was shown by Sánchez Almeida [32]). These visualizations were used during the development of this thesis and helped identify several bugs along the way; the author recommends that they are implemented in any rendering project involving polarization parameters.

Polarization rendering can be used to simulate many visual effects apart from polarizing filters. Weidlich and Wilkie [35] used the Mueller calculus to render uniaxial crystals more realistically. Such crystals produce doubly refracted images that are offset from each other and of different polarization states (one horizontal, the other one vertical).

Völker and Hamann [33] used the coherency matrix formalism to render cut diamonds in real-time. They used beam tracing for their rendering algorithm, which, performance-wise, is more suitable for high-quality rendering of large smooth flat surfaces such as the facets on a diamond than ray tracing is. Beam tracing is similar to ray tracing, but instead of using infinitesimally thin rays, thicker "pyramid-shaped" beams are used instead.

Although no such techniques are used in this thesis, recent developments in denoising and filtering algorithms have been a key factor in making the use of Monte Carlo ray tracing possible in real-time. This is an active area of research, and an in-depth overview of some fairly recent denoising algorithms can be found in the survey by Zwicker et al. [42].

Chapter 4

Implementation

Three renderers have been implemented in an application that was created with Nvidia's research-focused Falcor rendering framework [6].

The Falcor rendering framework was chosen because of its built-in support for DirectX RayTracing and its ease of use. It abstracts some of the CPU-side DirectX 12 code, which allowed for more of the development time to be spent on the GPU-side shader code. The use of this thin abstraction layer is not expected to make a notable impact on the results, since the difference between the three renderers is limited to differences in their shader code (and no CPU-intensive techniques are used in the application). Falcor version 3.2.2 was used as the base for all renderers, and only two minor changes (which are detailed in section 4.6) were made to its API. The GPU-side shaders were written in the High-Level Shading Language for DirectX (HLSL), and the code for those shaders (along with the source code for the rest of the application) is publicly available on GitHub.*

All three renderers use Whitted-style ray tracing [36] with one primary ray per pixel, one reflection ray per intersection point, and no refraction or shadow rays. The implementations would have been more immediately applicable in modern real-time applications if Monte Carlo ray tracing had been used; however, that would have been more time-consuming to develop as it requires temporal accumulation and denoising of Stokes vector data. Therefore, the simpler Whitted-style approach was chosen to ensure that the project would be completed on time.

4.1 Shading Model

Shading computations are implemented with the Cook-Torrance *microfacet* model [9], which was chosen since it is a commonly used shading model that is compatible with Whitted-style ray tracing and polarizing Fresnel functions [39, p. 36]. It uses tiny, perfectly flat, mirror-like surfaces called microfacets to model both smooth and rough materials. The shading of these materials is achieved with the use of one or more bidirectional reflectance distribution functions (BRDFs). A BRDF is a function $f(\hat{\mathbf{l}}, \hat{\mathbf{v}})$ that defines how much light is reflected towards a viewer from a point X on a surface, depending on the normalized vectors $\hat{\mathbf{l}}$ and $\hat{\mathbf{v}}$ that point from X to the light source and to the viewer respectively. For more information about BRDFs, see Akenine-Möller et al. [1, Ch. 7].

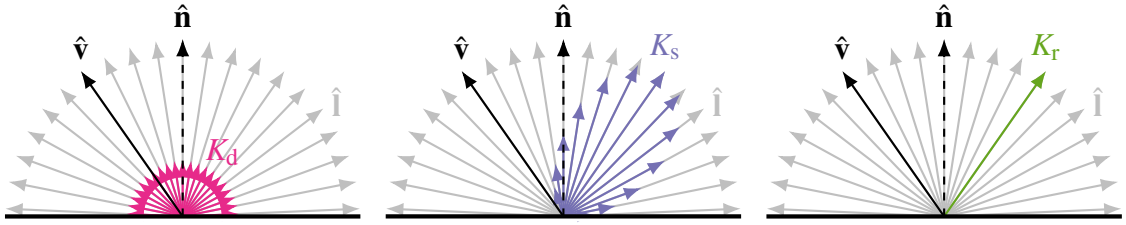
The shading model used by the three renderers can be expressed as the BRDF

$$f(\hat{\mathbf{l}}, \hat{\mathbf{v}}) = (K_\mathrm{r} E_r) \overline{\cos}\, \theta_r + \sum_{k \in \text{lights}} \left(K_\mathrm{d} E_{d_k} + K_\mathrm{s} E_{s_k} \right) \overline{\cos}\, \theta_k \,, \tag{4.1}$$

*https://github.com/viktor4006094/DegreeProject

where K_r, K_d, and K_s are the mirror, diffuse, and specular reflection coefficients; E_{d_k} and E_{s_k} is the diffuse and specular radiance at the point X from the light with index k; E_r is the radiance of the point that is being reflected in X; θ_r is the angle between the surface normal and the reflection vector; θ_k is the angle between the surface normal and a vector to the light with index k; and $\overline{\cos}$ is the cosine function clamped to non-negative values.

The approximate shape of the diffuse, specular, and mirror reflection coefficients are shown in Figure 4.1. The coefficients are defined as the amount of incoming light from the direction $\hat{\mathbf{l}}$ that is reflected in the view direction $\hat{\mathbf{v}}$.



(a) Diffuse reflection (b) Specular reflection (c) Mirror reflection

Figure 4.1: Visualization of the diffuse, specular, and mirror reflection terms. The relative length of the vector K represents how much of the light is reflected from that direction to the view direction $\hat{\mathbf{v}}$. If it is the same length as $\hat{\mathbf{l}}$, then all of the incoming light from that direction is reflected in $\hat{\mathbf{v}}$.

4.1.1 Diffuse Reflection Term

For the diffuse component—which in this implementation is assumed to be completely unpolarized—the simple Lambertian BRDF function

$$K_d = f\left(\hat{\mathbf{l}}, \hat{\mathbf{v}}\right) = \frac{\mathbf{c}_{\text{diff}}}{\pi} \tag{4.2}$$

is used [1, p. 240], with the term $\mathbf{c}_{\text{diff}}$ being the diffuse color of the material at the intersection point X.

4.1.2 Specular Reflection Term

The specular component uses the Cook-Torrance microfacet specular BRDF [9]. In its basic form it is defined as

$$K_s = f\left(\hat{\mathbf{l}}, \hat{\mathbf{v}}\right) = \frac{D(\hat{\mathbf{h}}, \alpha) G(\hat{\mathbf{l}}, \hat{\mathbf{v}}, \alpha) F(\hat{\mathbf{h}}, \hat{\mathbf{v}})}{4(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}})(\hat{\mathbf{v}} \cdot \hat{\mathbf{n}})}, \tag{4.3}$$

where $D \in [0, \infty)$ is a normal distribution function, $G \in [0, 1]$ is a geometric shadowing function, $F \in [0, 1]$ is a Fresnel function, α is the square value of the material's roughness, $\hat{\mathbf{n}}$ is the surface normal, $\hat{\mathbf{h}}$ is the half vector between $\hat{\mathbf{v}}$ and $\hat{\mathbf{l}}$, and the operator $\cdot$ is the dot product.

The renderers all use the GGX normal distribution function from Walter et al. [34]:

$$D_{GGX}(\hat{\mathbf{h}}, \alpha) = \frac{\alpha^2}{\pi((\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^2(\alpha^2 - 1) + 1)^2}. \tag{4.4}$$

The correct and exact geometric term G to use is the Smith one [17]. It is defined as

$$G_{Smith}(\hat{\mathbf{l}}, \hat{\mathbf{v}}, \alpha) = G_1(\hat{\mathbf{l}}, \alpha) G_1(\hat{\mathbf{v}}, \alpha) \tag{4.5}$$

where G_1 can be one of several alternatives. Due to its compatibility with the D_{GGX} normal distribution term, the GGX variant of G_1 is used. It is defined as

$$G_1(\hat{\mathbf{v}}, \alpha) = G_{GGX}(\hat{\mathbf{v}}, \alpha) = \frac{2(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}})}{(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}}) + \sqrt{\alpha^2 + (1 - \alpha^2)(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}})^2}}, \tag{4.6}$$

and the $G_1(\hat{\mathbf{l}}, \alpha)$ term is identical to $G_1(\hat{\mathbf{v}}, \alpha)$, but with $\hat{\mathbf{v}}$ replaced by $\hat{\mathbf{l}}$.

The denominator from Equation 4.3 can be used to simplify the geometric shadowing function into a visibility function V:

$$V(\hat{\mathbf{l}}, \hat{\mathbf{v}}, \alpha) = \frac{G(\hat{\mathbf{l}}, \hat{\mathbf{v}}, \alpha)}{4(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}})} = V_1(\hat{\mathbf{l}}, \alpha) V_1(\hat{\mathbf{v}}, \alpha), \tag{4.7}$$

where

$$V_1(\hat{\mathbf{v}}, \alpha) = \frac{1}{(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}}) + \sqrt{\alpha^2 + (1 - \alpha^2)(\hat{\mathbf{n}} \cdot \hat{\mathbf{v}})^2}}. \tag{4.8}$$

Just as with the G_1 terms, $V_1(\hat{\mathbf{l}}, \alpha)$ is identical to $V_1(\hat{\mathbf{v}}, \alpha)$, but with $\hat{\mathbf{v}}$ replaced by $\hat{\mathbf{l}}$. With this visibility function, the specular BRDF from Equation 4.3 can be simplified as

$$K_s = f\left(\hat{\mathbf{l}}, \hat{\mathbf{v}}\right) = D(\hat{\mathbf{h}}, \alpha) V(\hat{\mathbf{l}}, \hat{\mathbf{v}}, \alpha) F(\hat{\mathbf{h}}, \hat{\mathbf{v}}), \tag{4.9}$$

and it is this specular BRDF that is used by all implemented renderers.

What sets the three renderers' shading apart is the Fresnel function F. The Baseline renderer uses Schlick's approximation from Equation 2.3; the Polarization renderer uses the Mueller matrix for Fresnel reflectance from Equation 2.12; and the Hybrid renderer uses the latter for primary rays and the former for all reflection rays.

4.1.3 Mirror Reflection Term

Mirror-like reflections are implemented using a single non-stochastic recursive ray that is sent from the viewer to the center of each pixel. Unless the maximum recursion depth has been reached, a reflection ray is cast every time a ray hits a surface. The reflected radiance term E_r in Equation 4.1 represents the light that is returned by these recursive rays.

The mirror reflection coefficient uses the same functions as the specular BRDF:

$$K_r = f\left(\hat{\mathbf{l}}, \hat{\mathbf{v}}\right) = D(\hat{\mathbf{n}}, \alpha) V(\hat{\mathbf{r}}, \hat{\mathbf{v}}, \alpha) F(\hat{\mathbf{n}}, \hat{\mathbf{v}}), \tag{4.10}$$

with the visibility term V's incoming light vector $\hat{\mathbf{l}}$ replaced by the reflection vector $\hat{\mathbf{r}}$ (i.e., the mirror reflection of $\hat{\mathbf{v}}$ in the surface normal $\hat{\mathbf{n}}$), and the half vector $\hat{\mathbf{h}}$ replaced by the surface normal $\hat{\mathbf{n}}$ (since the two are equivalent for mirror reflections).

The result of the multiplication $D(\hat{\mathbf{n}}, \alpha) V(\hat{\mathbf{r}}, \hat{\mathbf{v}}, \alpha) \overline{\cos}\,\theta_r$ is clamped to the range $[0, 1]$ in order to prevent reflections from being brighter than what is being reflected (since D tends towards infinity for mirror reflections in smooth surfaces).

4.2 Baseline Renderer

The Baseline implementation uses the shading model described in the previous section with Schlick's approximation from Equation 2.3 as its Fresnel function F, and with all light data represented by `float3` RGB-color vectors.

4.2.1 Data Structures and Rendering Algorithm

The information carried by rays in DXR is defined in programmable data structures called *payloads*. A ray's payload is used to return information from the ray's operations and is initialized before the ray is cast.

Listing 4.1 shows the payload that is used in the Baseline renderer: `color` is the light data from the ray's intersection point, and `recursionDepth` is used to keep track of how deep a ray is so that the number of recursive reflections can be limited.

Listing 4.1: Ray payload used in the Baseline implementation.

```
1 struct Payload
2 {
3     float3 color;
4     uint   recursionDepth; // How many reflections deep the payload is
5 };
```

Algorithm 4.1 describes the Baseline implementation's rendering algorithm, using the data types from Listing 4.1 and the shading terms from Equation 4.1.

Algorithm 4.1: The Baseline implementation's rendering algorithm.

procedure TRACERAY(*ray*)
  if *ray* intersects with geometry **then**
    for each light source $k \in$ scene **do**
      ray.`color` += $(K_d E_{d_k} + K_s E_{s_k})\overline{\cos}\,\theta_k$
    if *ray*.`recursionDepth` < max recursion depth **then**
      TRACERAY(r') ▷ Cast a reflection ray
      ray.`color` += $(K_r\, r'.\texttt{color})\overline{\cos}\,\theta_r$
procedure MAIN
  for each primary ray $r \in$ output image **do**
    TRACERAY(r) ▷ Cast a primary ray
    output color = r.`color`

4.3 Polarization Renderer

The Polarization implementation has the same structure as the Baseline version, but it differs in two key ways: the Fresnel function F uses the Mueller matrix for Fresnel reflectance from Equation 2.12 instead of Schlick's approximation, and light data is represented by three Stokes vectors and a reference frame instead of a `float3` vector.

The diffuse term is assumed to be completely unpolarized in this implementation, so the same Lambertian shading function from subsection 4.1.1 is used in both the Baseline and the Polarization renderer. As mentioned in section 2.5.3, unpolarized light does not

need to take its rotation into account; therefore, each color channel k in the diffuse term's resulting `float3` vector $\mathbf{c} = (I_R, I_G, I_B)$ can be added to the specular term's Stokes vector $\vec{\mathbf{S}}_k$ for that color channel with the simple equation

$$\vec{\mathbf{S}}_k + I_k = \begin{bmatrix} s_{k_0} + I_k \\ s_{k_1} \\ s_{k_2} \\ s_{k_3} \end{bmatrix}. \tag{4.11}$$

It is worth noting that subsurface scattering (i.e., diffuse lighting) in thin materials can show a significant degree of polarization in some scenarios [8]; however, the influence that might have on the visual effects of a polarizing filter is assumed to be mostly insignificant when compared to the influence of specular reflections.

4.3.1 Data Structures and Rendering Algorithm

The Polarization implementation needs to keep track of more information about the light than the Baseline one does. As such, its ray payload is significantly larger (64 bytes instead of 16 bytes). The three data structures used in the Polarization renderer are shown in Listing 4.2.

Listing 4.2: Ray payload and data structures used in the Polarization implementation.

```
struct MuellerData
{
    // red, green, and blue Mueller matrices
    float4x4 mmR;
    float4x4 mmG;
    float4x4 mmB;
};

struct StokesLight
{
    // red, green, and blue Stokes vectors
    float4 svR;
    float4 svG;
    float4 svB;

    // local coordinate system's x-axis unit vector
    float3 referenceX;
};

struct Payload
{
    StokesLight lightData;
    uint        recursionDepth; // How many reflections deep the payload is
};
```

Just like in the Baseline renderer, `recursionDepth` is used to limit the number of recursive reflections, and light is approximated as the sum of three quasi-monochromatic color components (red, green, and blue); however, instead of using a `float3` to represent them, three Stokes vectors are used (one for each color component). Since the polarization state is per-color, it is possible (albeit unlikely) for a ray's red wavelengths to be completely linearly polarized while its blue wavelengths are completely unpolarized.

Algorithm 4.2 describes the Baseline implementation's rendering algorithm, using the data types from Listing 4.2 and the shading terms from Equation 4.1.

Algorithm 4.2: The Polarization implementation's rendering algorithm.

```
1:  procedure TraceRay(ray)
2:      if ray intersects with geometry then
3:          for each light source k ∈ scene do
4:              AlignReferenceFrame(K_s E_{s_k}, ray.lightData)
5:              ray.lightData += (K_d E_{d_k} + K_s E_{s_k}) cos θ_k
6:          if ray.recursionDepth < max recursion depth then
7:              TraceRay(r')                                  ▷ Cast a reflection ray
8:              AlignReferenceFrame(r'.lightData, ray.lightData)
9:              ray.lightData += (K_r r'.lightData) cos θ_r
10: procedure Main
11:     for each primary ray r ∈ output image do
12:         TraceRay(r)                                       ▷ Cast a primary ray
13:         if polarizing filter enabled then
14:             AlignReferenceFrame(r.lightData, screen)
15:             ApplyPolarizingFilter(r.lightData)
16:         output color = r.lightData
```

4.3.2 Reference Frame Tracking

As was discussed in section 2.5.3, information about the rotation of the Stokes vectors has to be maintained in order for operations with them to function correctly. This is done using the `referenceX` vector in the `StokesLight` data structure shown in Listing 4.2. It represents the ray's reference frame's local coordinate system's x-axis unit vector $\hat{\mathbf{x}}$ and is equivalent to the E_x vector from Figure 2.3.

For the Mueller matrix representing reflection in a point on a surface, $\hat{\mathbf{x}}$ is calculated as the cross product $\hat{\mathbf{h}} \times \hat{\mathbf{v}}$ (i.e., it is a normalized vector that is perpendicular to both the microfacet normal and the view direction). The same $\hat{\mathbf{x}}$ can represent both the Mueller matrix's entry and exit reference frames since their x-axis unit vectors are identical.

Before a color channel's Stokes vector is multiplied with the Mueller matrix that represents a surface's reflection component K_r for that color channel, the Stokes vector's $\hat{\mathbf{x}}$ vector is rotated to match that Mueller matrix's entry frame (represented by $\hat{\mathbf{x}}'$). This is accomplished using Equation 2.19, where $\phi \in [-\pi, \pi]$ is the signed rotation angle between the two vectors. The angle ϕ is calculated with the help of the direction of the light $\hat{\mathbf{z}}$* as

$$\phi = \operatorname{atan2}\left(\hat{\mathbf{z}} \cdot \left(\hat{\mathbf{x}} \times \hat{\mathbf{x}}'\right), \hat{\mathbf{x}} \cdot \hat{\mathbf{x}}'\right), \tag{4.12}$$

where $\operatorname{atan2}(a, b)$ is the two-argument arctangent function that computes an angle in radians between the positive x-axis and the vector pointing from $(0,0)$ to the point (a,b). Figure 4.2 shows a visualization of two reflections' reference frames and the angle ϕ that is used to rotate the reflected Stokes vectors between the two reflections.

This reference frame rotation is also performed when a Stokes vector is added to another one (with $\hat{\mathbf{x}}'$ then representing the reference x-axis vector of the target Stokes vector). No rotation is needed before multiplying the specular radiance E_s with K_s since all light sources in the application are assumed to be unpolarized. The resulting Stokes vectors, however, do need to be aligned correctly before they can be added to the ray's `lightData`.

*i.e., the opposite direction of the ray used in the ray tracing

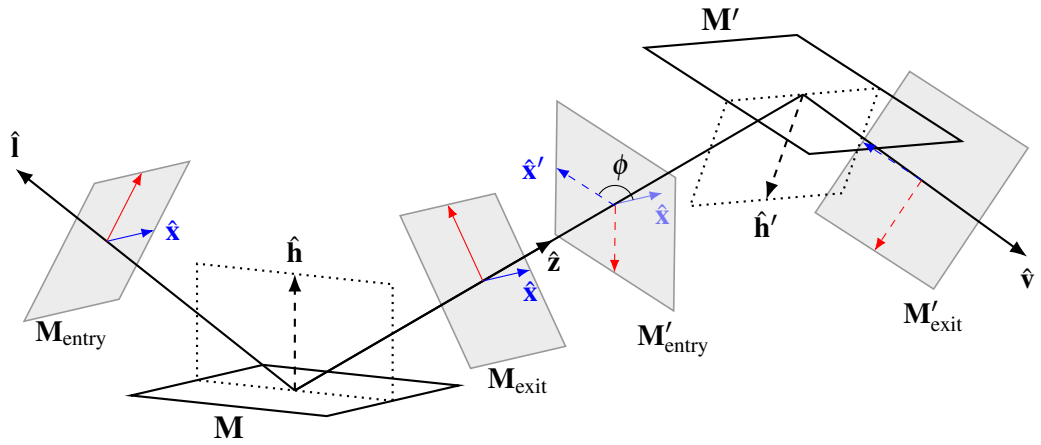


Figure 4.2: Visualization of a reference frame rotation for a sequence of reflections. The Stokes vectors in the ray that is reflected off of the surface represented by the Mueller matrices $\mathbf{M}$ are aligned with the reference frame $\mathbf{M}_{\text{exit}}$, and they need to be rotated by the angle ϕ to match the entry frame $\mathbf{M}'_{\text{entry}}$ before they can be multiplied by the second surface's Mueller matrices $\mathbf{M}'$.

4.4 Hybrid Renderer

The Hybrid implementation is a combination of the Baseline and Polarization implementations, and makes use of both of their payloads and Fresnel functions. Shading of the primary ray is accomplished with the data structures and functions from the Polarization implementation, and shading of all recursive reflection rays is done with the simpler payload and calculations from the Baseline implementation. Equation 4.11 is used to incorporate the reflection rays' unpolarized result into the primary ray's shading computations.

4.5 Implementation Summary

A summary of the differences between the Baseline, Polarization, and Hybrid renderers (which have been described in the preceding sections) is shown in Table 4.1.

Table 4.1: Summary of which Fresnel function F and which light representation is used by the primary and reflection rays in the three renderers.

	Primary rays		Reflection rays	
Renderer	F	Light representation	F	Light representation
Baseline	F_{Schlick}	`float4`	F_{Schlick}	`float4`
Polarization	$\mathbf{M}_{\text{Fresnel}}$	`StokesLight`	$\mathbf{M}_{\text{Fresnel}}$	`StokesLight`
Hybrid	$\mathbf{M}_{\text{Fresnel}}$	`StokesLight`	F_{Schlick}	`float4`

4.6 Changes Made to the Falcor API

Two changes were made to the Falcor API to accommodate for the Polarization and Hybrid renderers: the size limit `FALCOR_RT_MAX_PAYLOAD_SIZE_IN_BYTES` was increased from

`16*sizeof(float)` to `64*sizeof(float)`, to allow for larger ray payloads (which were described in subsection 4.2.1 and 4.3.1); and a metalness value of type `float` was added to the `ShadingData` struct, to be used to set the IOR values (which will be described in the next section).

4.7 Material Texture Limitations

All the test scenes used in this project have textures defining their specular color $\mathbf{c}_{\text{spec}}$ that is used by the Baseline implementation's Fresnel function, but none of them have textures defining their complex IOR η that is used in the Polarization and Hybrid implementations' polarizing Fresnel function (as discussed in section 2.3).

Because of this, the values of η are approximated in the renderer from the available material data. For dielectrics, the values are set to $n = 1.5$ and $\kappa = 0.0$, which are near the IOR values of most glass and many other common dielectric materials. For metals, they are set from a list of pre-defined materials in the application's interface, with gold being the default option. To prevent the shader compiler from optimizing away the IOR values, they are sent to the GPU via a constant buffer. In the shader code, n and κ are then set to a blend of these pre-defined values depending on the metalness value in the textures (which is typically either 1.0 or 0.0).

As a consequence of this, *all* metallic surfaces rendered with the Polarization and Hybrid implementations are given the same IOR values, making them look like the same material (regardless of which type of metal they were supposed to be according to their textures). Since the Hybrid renderer uses the Polarization renderer's Fresnel function for the first shading point and the Baseline one's for all subsequent reflection points, this could also cause inconsistencies in how surfaces (metallic ones in particular) appear when viewed directly and when viewed in reflections. In order to prevent this, the R_0 values used by the Hybrid renderer's F_{Schlick} function are calculated from the application-controlled IOR values with Equation 2.4, instead of being set from the scenes' material textures as they are in the Baseline renderer.

In a proper renderer, the IOR values would be saved in material textures just as $\mathbf{c}_{\text{spec}}$ is in the current implementation. Metallic materials would need one `float3` texture for their n values and one `float3` texture for their κ values. For dielectric materials, it would be enough with just a `float3` texture for n (since κ is zero for dielectrics). A single `float` value might even suffice if the value of n is deemed to be similar enough in the red, green, and blue wavelengths of light.

4.8 Polarization Demonstration

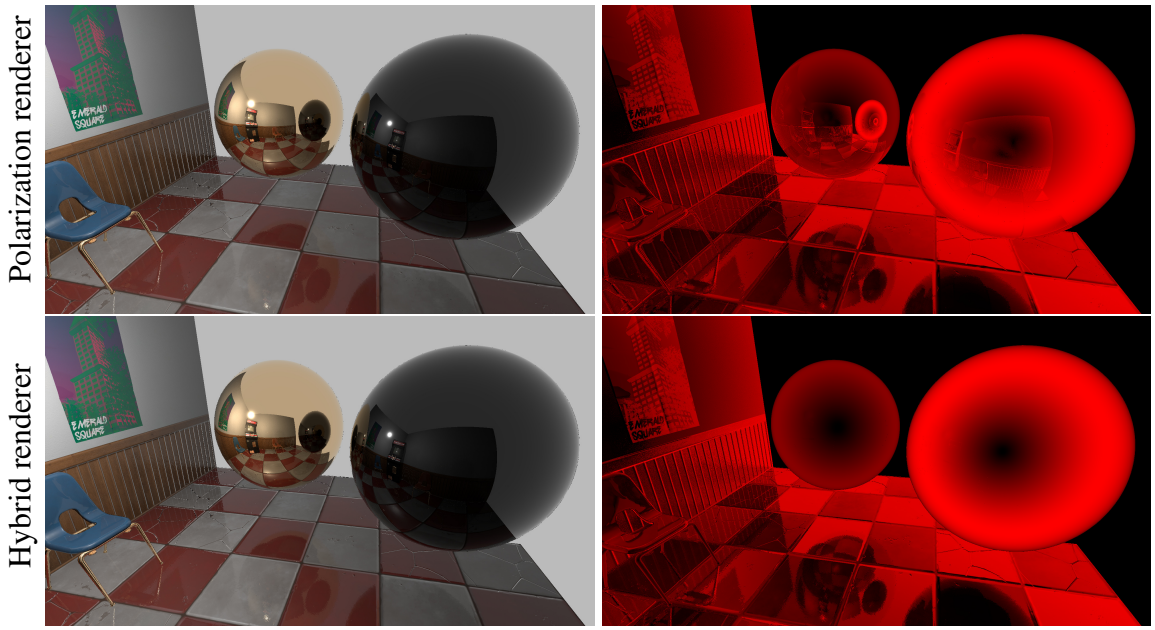
Figure 4.3 demonstrates how a polarizing filter implemented with Equation 2.16 changes the appearance of a scene in the two polarization-capable renderers. Apart from functioning as a polarizing filter, the implemented filter also doubles the light's intensity to compensate for the filter's reduction of brightness (which was discussed in section 2.4).

The degree of polarization in Figure 4.3b is calculated as the average value of the three color channels' p values from Equation 2.9. The difference between the Polarization and Hybrid renderers is evident by looking at the two spheres: in the Polarization version, the

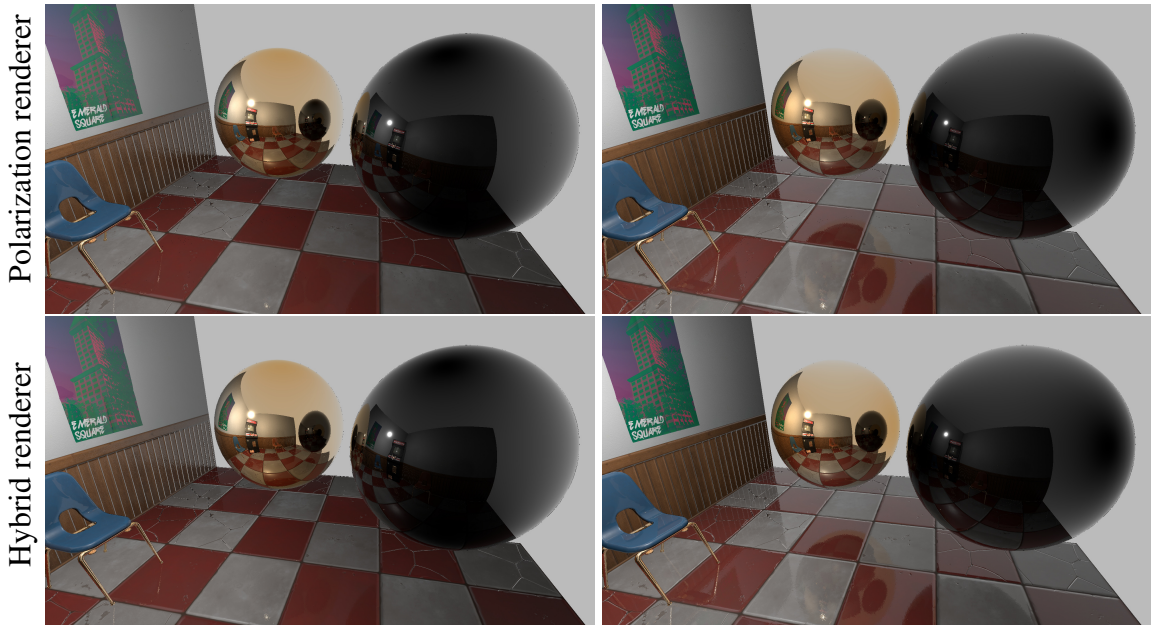
polarization data from beyond the first reflection point is included, while in the Hybrid renderer only the reflection point that is closest to the viewer has any impact on the polarization. Note also how the light rays coming from the black dielectric sphere are significantly more polarized than light rays coming from the gold sphere in both implementations.

In Figure 4.3c, the vertically angled polarizing filter has significantly reduced the intensity of the reflected light in the floor and the top and bottom of the black sphere. In the Polarization version, this effect also applies to the reflection of the black sphere in the golden one. While in the Hybrid version, the reflection of the black sphere is unaffected by the filter. The brightness of the reflections in the golden sphere's top and bottom sides have been reduced as well, but the effect is not as noticeable as it is in the black sphere. As per Malus' law from Equation 2.6, all of the vertically linearly polarized light is let through, and half of the unpolarized light is filtered out. The relative brightness of the reflected light in the wall and the sides of the spheres has thereby increased.

With the polarizing filter angled horizontally, as shown in Figure 4.3d: the brightness of reflections is reduced in vertical surfaces instead of horizontal ones; the reflections in the wall and the sides of the spheres have become dimmer; and the relative brightness of the reflections in the floor, and in the top and bottom of the spheres has increased. The same limitations of polarization effects in reflections in the Hybrid renderer apply here as well.



(a) Unfiltered (b) DOP

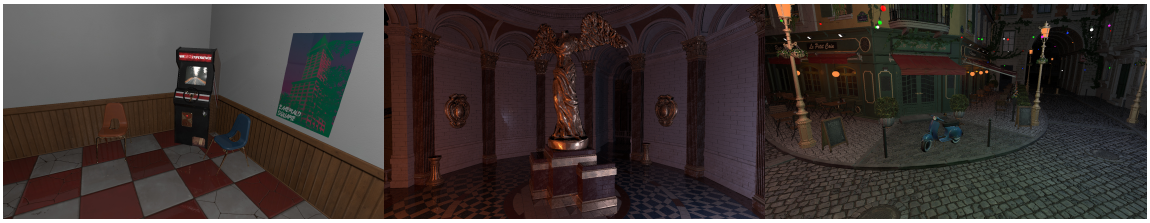


(c) Vertical polarizing filter ↕ (d) Horizontal polarizing filter ↔

Figure 4.3: Polarization and the visual impact of a polarizing filter in the Polarization and Hybrid renderers. (b) shows the degree of polarization in red, and visualizes how the Polarization renderer captures polarization information beyond the first reflection point, while the Hybrid renderer does not (as is apparent from the lack of polarization detail in the reflections in the spheres). It also shows how light reflected in the right dielectric (black glass) sphere is polarized to a much greater degree than the light reflected from the left metallic (gold) sphere. (c) and (d) show the visual effects of a polarizing filter. Notice how the filters do not affect the reflection of the black sphere in the gold one when the Hybrid renderer is used.

Chapter 5

Experiment Details



(a) Arcade (b) Temple (c) Bistro

Figure 5.1: The three test scenes used in the performance tests.

The three renderers were tested in three scenes of varying complexity to evaluate their relative performance. The scenes used were: *Nvidia Arcade* ("Arcade") [18], *Unreal Engine Sun Temple* ("Temple") [13], and *Amazon Lumberyard Bistro Exterior* ("Bistro") [22]. Their light setups and camera locations were modified for the tests.

Figure 5.1 shows what the three test scenes look like when rendered with the Polarization renderer; the scenes' triangle and light counts are listed in Table 5.1.

Table 5.1: Test scene triangle and light counts.

Scene name	Triangles	Lights
Arcade	8,758	1
Temple	606,376	2
Bistro	2,832,120	8

Each scene was tested with 1–5 recursive reflection rays (i.e., up to 2–6 rays per pixel) to see how, if at all, the performance difference between the implementations scaled with the per-pixel ray count. No higher recursion depths were tested due to time constraints and the fact that modern GPUs can only be assumed to trace "at most a few rays per pixel" in real-time [16, p. 46], so testing higher recursion depths was not considered to be relevant for the purposes of this work. Furthermore, the deeper down the recursion tree a ray is, the less impact it will have on the final output color of a pixel, so increasing the ray count further would only have had a negligible impact on the visual appearance of the test scenes.

All tests were performed with the application running at a resolution of 1920×1080 px; the maximum number of rays per frame ranged from 4,147,200 when using a ray recursion

depth of one to 12,441,600* when using a ray recursion depth of five. In each test, the max trace recursion depth in DXR was set to the maximum number of rays per pixel permitted by the recursion depth, so that unnecessary stack memory would not be allocated [12].

5.1 Measurements

The average frame rate (FPS) and frametimes were measured with Nvidia's FrameView application [11]. Frametime is defined as the time between consecutive calls to `Present()`, i.e., how much time it takes from the time a frame is done rendering until the next frame is done rendering (the frametime is therefore equal to $\frac{1}{\text{FPS}}$).

VRAM usage was recorded at the start of each test from within the application by calling the `IDXGIAdapter3::QueryVideoMemoryInfo` function and saving the value of the returned `CurrentUsage` field. It was recorded just once per test since the value did not change during runtime.

5.2 Validity

All applications that did not have to be active when running the tests were closed for the duration of the tests in order to minimize any potential performance impact they might have on the application's performance.

Frametimes are not necessarily stable from frame to frame, so each test was run for approximately 9.5 seconds† in order to get accurate measurements. Each test configuration was run two separate times to compensate for possible performance fluctuations caused by variances in CPU and GPU temperatures (only two tests per configuration were run due to time constraints and how consistent the resulting measurements were). The order of the performance tests is described in Algorithm 5.1; in total, 90 individual tests were run.

Algorithm 5.1: Performance test order.

```
for each recursion depth ∈ [1,2,3,4,5] do
    for each test iteration ∈ [1,2] do
        for each scene ∈ [Arcade, Temple, Bistro] do
            for each renderer ∈ [Baseline, Polarization, Hybrid] do
                RECORDDATAFOR(10 seconds)
```

5.3 System Under Test

At the time of writing, only the Turing series of GPUs have hardware acceleration for DirectX RayTracing. A RTX 20 series graphics card was therefore used for all the performance tests. The specifications of the computer that was used are listed in Table 5.2.

*in practice, this upper limit is unlikely to be reached since the recursive generation of reflection rays ends when a ray misses all scene geometry

†capture duration set to 10 seconds in FrameView

Table 5.2: System under test.

CPU	AMD Ryzen 5 1600. Six cores @ 3.2 GHz with SMT enabled.
GPU	GeForce RTX 2070 Super 8 GB. *
RAM	16 GB DDR4-3200 @ 2933 MHz.
OS	Windows 10 Home 64 bit (1909). OS Build: 18363.778. †

*Display driver: GeForce Game Ready Driver Version 445.87.

†Active power plan: AMD Ryzen™ Balanced.

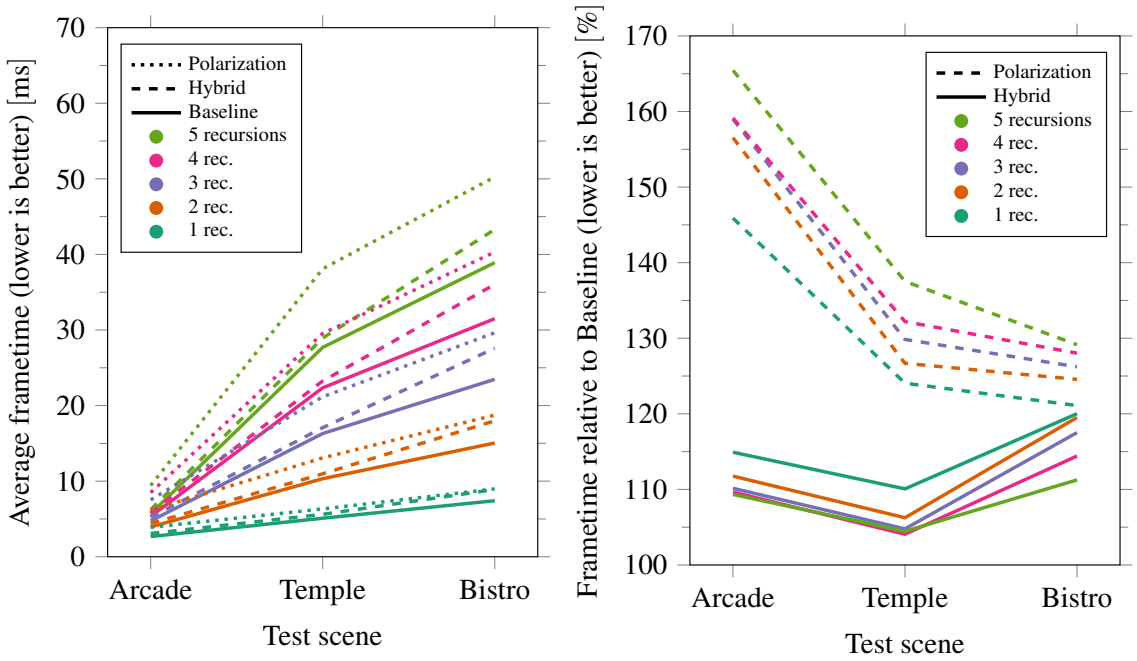
Chapter 6

Results and Analysis

The results of the performance tests described in the previous chapter are presented and analyzed here to illustrate the performance difference between the three renderers.

6.1 Average Frametimes and Frame Rates

The average frametimes are shown in Figure 6.1a. As expected, the more complex scenes took longer to render, and the Polarization and Hybrid renderers performed worse than the Baseline one in all of the tests.



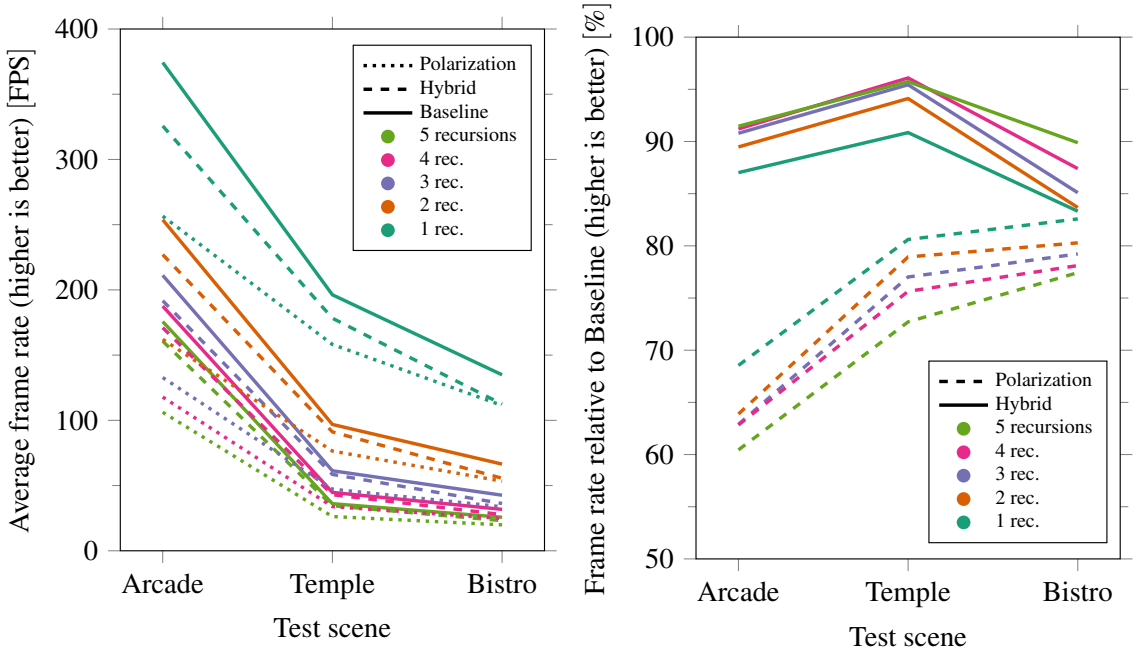
(a) Average frametimes

(b) Average frametime differences

Figure 6.1: Average frametimes and frametime differences from the Baseline for the three test scenes. Average differences are calculated relative to the Baseline version (Baseline = 100%).

Figure 6.1b shows the difference between the average frametime in the polarization-capable renderers and the Baseline one. As can be seen in the graph, the Polarization implementation needs ca. 60% more time to render a frame in the Arcade scene than the Baseline does. In the more complex Temple and Bistro scenes, this difference is closer to 30%. In all scenes, this difference increased with the recursion depth.

Figure 6.1b also shows that the Hybrid renderer only had a performance difference of about 10–20% when compared to the Baseline version in all scenes. Unlike the Polarization renderer's results, this difference decreased as the recursion depth increased.



(a) Average frame rates

(b) Average frame rate differences

Figure 6.2: Average frame rates and frame rate differences from the Baseline for the three test scenes. Average differences are calculated relative to the Baseline version (Baseline = 100%).

Figure 6.2 shows the equivalent graphs for the average frame rates. Figure 6.2a gives a clearer picture than Figure 6.1a of how the different test configurations performed in the Arcade scene, where the absolute differences between frametime measurements were small compared to the other scenes' results (but the relative differences were large; thus, the impact on the frame rate was more noticeable than on the frametime). For example, it can here be seen that when using one recursive ray in the Arcade tests, the cost of switching from the Baseline renderer to the Polarization renderer is as expensive as increasing the recursion depth to two. The performance impact of switching from the Baseline to the Hybrid renderer is also apparent in this graph, but the slowdown is not as significant as when switching to the Polarization renderer.

6.2 VRAM Usage

The results of the VRAM measurements are summarized in Table 6.1 and Table 6.2, and their absolute values are listed in Table 6.3.

Table 6.1 shows that the memory usage was higher in the Polarization implementation than in the baseline one, but only for the two deepest recursion depths tested.

The Hybrid renderer's VRAM usage did not increase with the recursion depth as the Polarization renderer's VRAM usage did*; however, as can be seen in Table 6.2, there was

*at least not with any of the tested recursion depths

a constant difference of exactly 4 and 8 KiB in the Temple and Bistro scenes respectively. These differences are orders of magnitude smaller than any of the non-zero increases caused by the Polarization renderer, but they are there nonetheless. For context, the graphics card that was used to perform the tests has 8 GiB of dedicated VRAM.

Table 6.1: Increase in VRAM usage of the Polarization renderer over the Baseline renderer. No per–test scene values are shown because the difference in bytes between the Polarization and Baseline results was identical in all of them.

Recursion depth	Increase [KiB]	Increase [B/px]
1–3	0	0.00
4	10,240	5.06
5	15,360	7.59

Table 6.2: Increase in VRAM usage of the Hybrid renderer over the Baseline renderer. No per–recursion depth values are shown because the difference in bytes between the Hybrid and the Basline results was identical in all of them.

Test scene	Increase [KiB]
Arcade	0
Temple	4
Bistro	8

Table 6.3: Raw VRAM usage measurements from all the test configurations. All measurement values are expressed in KiB.

		Recursion depth				
Test scene	Renderer	1	2	3	4	5
Arcade	Baseline	3,949,900	3,949,900	3,949,900	3,949,900	3,949,900
	Polarization	3,949,900	3,949,900	3,949,900	3,960,140	3,965,260
	Hybrid	3,949,900	3,949,900	3,949,900	3,949,900	3,949,900
Temple	Baseline	4,060,156	4,060,156	4,060,156	4,060,156	4,060,156
	Polarization	4,060,156	4,060,156	4,060,156	4,070,396	4,075,516
	Hybrid	4,060,160	4,060,160	4,060,160	4,060,160	4,060,160
Bistro	Baseline	4,732,980	4,732,980	4,732,980	4,732,980	4,732,980
	Polarization	4,732,980	4,732,980	4,732,980	4,743,220	4,748,340
	Hybrid	4,732,988	4,732,988	4,732,988	4,732,988	4,732,988

6.3 Raw Frametime Measurements

The last graphs visualize the underlying raw frametime measurements and the per–test configuration averages that were used in Figure 6.1 and Figure 6.2. The Arcade scene frametimes are shown in Figure 6.3, the Temple scene frametimes in Figure 6.4, and the Bistro scene frametimes in Figure 6.5.

Note that the averages shown in these graphs are the average values of *all* frametime measurements for those test configurations. They are visualized as lines, but each line represents *one* single constant value, not one value per time point.

With the exception of a few short spikes, the frametimes remained consistent throughout each test in all of the tested configurations. This is indicated by how close the raw data measurements are to the per–test configuration averages.

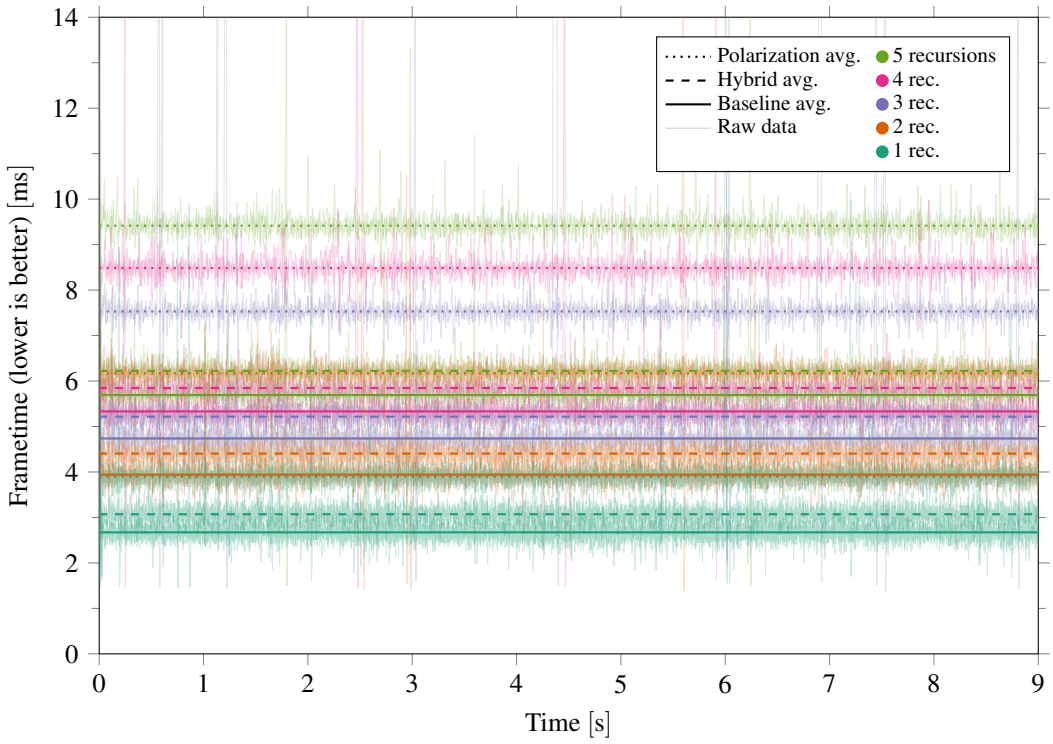


Figure 6.3: Raw and per–test configuration average frametime data for the Arcade test scene.

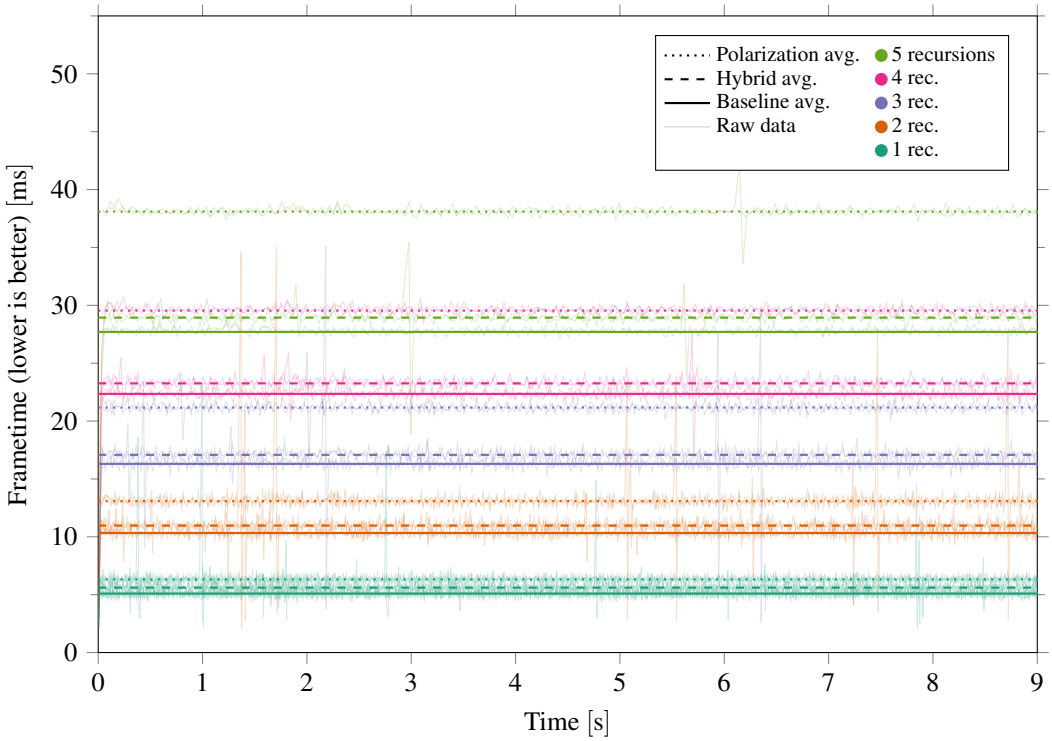


Figure 6.4: Raw and per–test configuration average frametime data for the Temple test scene.

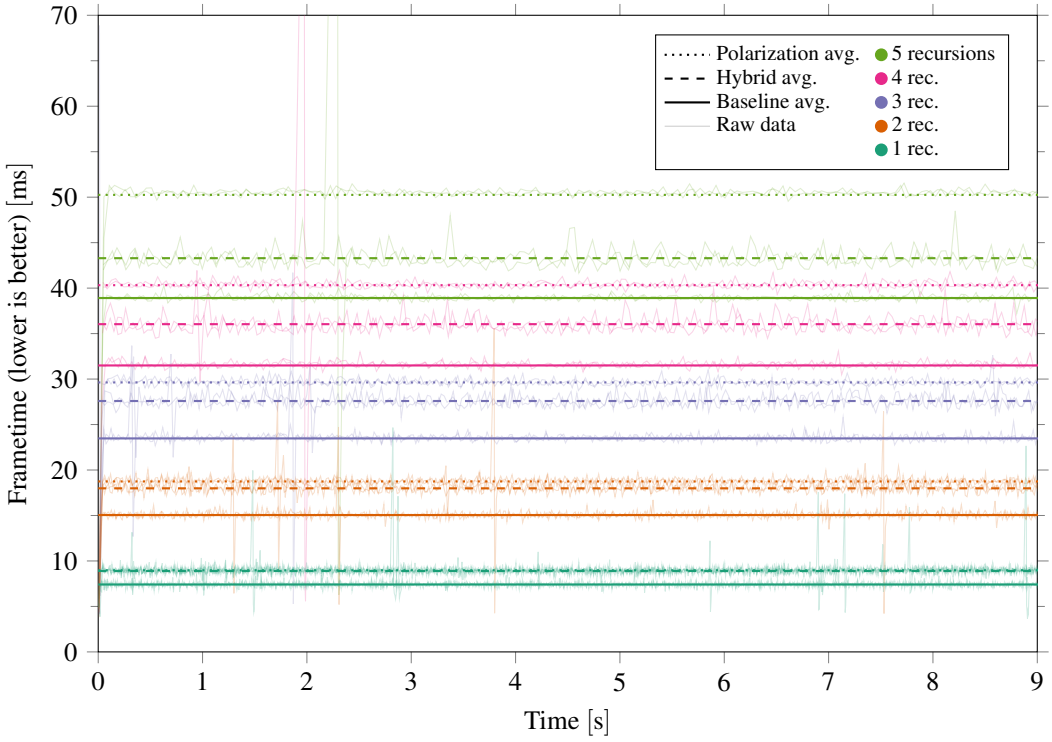


Figure 6.5: Raw and per–test configuration average frametime data for the Bistro test scene.

Chapter 7

Discussion

The results presented in the previous chapter were overall expected and enough to provide an answer to the research question; however, there are some aspects worth taking into consideration when interpreting the measurements.

7.1 Performance Impact

The frametime results presented in chapter 6 were mostly expected from the apparent difference in computational complexity between the renderers. The VRAM results, on the other hand, were unexpected. It was assumed that the memory footprint of the Baseline renderer would be noticeably smaller than of the other two renderers, but the recorded measurements did not reflect this. Limitations of the measurement methods or the limited scale of the application might be the cause of this lack of a noticeable difference.

7.1.1 Frametimes

Although the magnitude of the performance difference was difficult to estimate in advance, it was expected that the Polarization renderer would perform worse than the other two. It uses not only a larger ray payload than the Baseline renderer, but also a significantly more elaborate Fresnel function (compare the calculations required to produce the Mueller matrix in Equation 2.12 with the ones in Equation 2.3).

There was a significant difference between the performance impact in the simple Arcade scene (ca. 60%) and in the more complex Temple and Bistro scenes (ca. 35% and ca. 30% respectively). Due to the latter two scenes' geometrical complexity, the performance impact in those scenes is likely more representative of what the performance impact would be in a complex ray-traced application such as a video game.

Polarization having a larger impact on the performance in the simplest scene than in the more complex ones matches the—now three decades old—findings by Wolff and Kurlander [40, p. 51]. In their coherency matrix–based implementation, the rendering time was approximately doubled in simple scenes, but approached the performance of their polarization-less version as scenes grew in complexity. The reason for this is that the execution time of the shading computations is mostly dependent on the size of the output window, while the execution time of the ray tracing intersection computations is mostly dependent on the geometrical complexity of the scene. So as the scenes grow more complex, the time spent on ray tracing calculations is increased while the time spent on shading calculations remains fairly constant.

The performance improvement of the Hybrid renderer over the Polarization one was also expected, and in many scenes the visual difference between the two implementations will likely be difficult to notice. However, the Hybrid version does lack polarization information in rays beyond the primary ones and is therefore noticeably less realistic than the Polarization version when viewing objects in a mirror or similarly smooth reflective surface. It would certainly be possible to use the polarization-capable payload and calculations for the first reflection ray as well and improve the realism that way, but that would also increase the performance impact. Perhaps a *max polarization recursion depth* limit could be implemented on a per-object basis. So that the first reflections in mirrors and glass can be shaded with polarization parameters, while reflections in rougher objects are always shaded without them.

As mentioned in section 6.1, the relative performance difference increased with the recursion depth in when using the Polarization renderer, but decreased when using the Hybrid renderer. This is likely due to the Hybrid implementation using the same payload and shading computations as the Baseline renderer for all reflection rays, so as the recursion depth increases, so does the ratio of rays that are shaded with those simpler payloads and calculations. With a recursion depth of one (i.e., one primary ray and one reflection ray per pixel), up to 50% of the shading calculations are using the Baseline payload and functions, and if the recursion depth is increased to four then that maximum ratio increases to 75%.

As was shown in section 6.3, the frametime measurements were fairly stable around the average per–test configuration values. This consistency is primarily the result of the virtual camera being stationary in all of the test scenes—since the time needed to render a frame is highly dependent on how complex the scene geometry in front of the camera is. Had moving cameras been used in the scenes then these measurements would likely have been much less consistent, and analyzing them primarily by their average values would not have been as meaningful as it is when stationary cameras are used.

7.1.2 VRAM Usage

The difference between the Polarization renderer's and the Baseline's VRAM usage was identical across all test scenes and so it is likely only dependent on the recursion depth and the application's rendering resolution. This increased VRAM usage was expected due to the Polarization renderer's larger ray payload and more complex shader code, but the measured difference is smaller than anticipated at less than eight bytes per pixel with the highest tested recursion depth (and zero bytes per pixel for the three lowest ones). The fact that there was only an increase when using the two highest recursion depths tested suggests that there might be a difference in the lower recursion depths as well. This theoretical difference was, nevertheless, not noticeable in the measurements (likely due to the register pressure being too low for it to impact the VRAM).

Apart from the polarization calculations, the shading computations were relatively simple and did not include many of the graphical effects used in modern video games. Had more of those effects been included (thereby making the shading computations more complex and increasing the register pressure), then the actual difference between the renderers' memory requirements might have been more apparent. In the current implementation, a different measurement technique than simply checking the applications total VRAM usage would be needed in order to get a clearer picture of exactly how significant the impact on the memory usage is.

What caused the constant differences in VRAM usage between the Hybrid and Baseline renderers is not known; however, those differences were small enough to essentially be inconsequential (8 KiB is less than 0.0001% of the 8 GiB of dedicated VRAM on the graphics card that was used).

7.1.3 Limitations of the Tests

The performance tests were all limited to the same resolution (1920×1080 px), so it is not known how the performance and memory usage of the renderers would scale with an increase or decrease in resolution.

The scenes were all rendered from stationary viewpoints, which resulted in measurements that were easy to compare. Had a moving camera been used instead, then the difference between the renderers would likely have been higher in some scenarios and lower in others.

As mentioned in subsection 7.1.2, a more complex rendering pipeline would likely have given results that are more immediately applicable to a large computer graphics application such as a video game. Although the geometry in the test scenes was completely static, the polarization effects would apply for moving and animated geometry as well. Ultimately, the best way to evaluate the feasibility of using polarization parameters in a real-time ray-traced application would be to implement it in that application and compare the performance.

7.2 Implementation Considerations

The relative slowdown of the Hybrid renderer (about 10% in the tested scenes) is significant enough that such an implementation is unlikely to be considered feasible in applications where a high frame rate and a low latency is important; however, it is small enough that it could possibly be used in less performance-critical modes (e.g., photo modes). In those modes the slower, but more physically accurate, Polarization implementation could possibly be considered as well.

Using both polarization rendering and conventional rendering in the same application, as is done in the Hybrid renderer, comes with a few complications. All shaders that are to be used would need to be adapted for both light calculations without and with polarization parameters (including reference frame rotations). The material textures would also need IOR values for the polarization-capable Fresnel function that match the specular color used in the polarization-less Fresnel function. As mentioned in subsection 2.3.1 (and implemented in the Hybrid renderer), the specular color can be calculated from the IOR values, but such computations should optimally be avoided at runtime.

The Hybrid implementation could likely be further simplified and optimized. If a point that is being shaded is only reached by unpolarized light, then only the first column of values in the Mueller matrix for Fresnel reflectance would matter, and there would be no need to calculate the phase retardation terms $\delta_\perp$ and $\delta_\parallel$. The need for one Stokes vector per color channel could also be reduced for dielectric surfaces (since the difference between their red, green, and blue IOR values is generally fairly small). For a metal such as gold, however, the differences in its IOR values is what give the material its distinct color.

As mentioned in section 2.1, Monte Carlo ray tracing with sophisticated denoising is commonly used in modern real-time ray tracers. Implementing polarizing filters in such an

application would require that temporal polarization data (i.e., the Stokes vector information for each pixel) is saved in a buffer so that it can be reused the next frame. Since each pixel has three color channels and four Stokes vector components per color, this would require a significant amount of memory unless the data is first pruned or compressed in some way. Saving just one color channel's s_1 and s_2 values would likely be sufficient since the three channel's polarization states should be fairly similar, and the intensity value s_0 is typically already saved in a buffer in such an implementation. The last Stokes parameter s_3 would not need to be saved at all since it has no impact on a polarizing filter.

Chapter 8

Conclusions and Future Work

Two polarization-capable real-time renderers have been presented and had their performance evaluated in comparison with a conventional polarization-less approach. Due to limitations in how the memory usage was measured and the limited scale of the test application, reliable conclusions regarding the memory usage impact could not be drawn; however, the frametime results provide enough data to answer the research question. They show that the naive Polarization approach causes a roughly 30–60% increase in the time it takes to render a frame, and the slightly less realistic Hybrid approach increases frametimes by only about 5–15%.

This thesis has shown the performance impact of adding polarization parameters to a real-time renderer is low enough to potentially be of use in applications where the effects of a polarizing filter are desirable—especially if the implementation is refined with further optimizations and simplifications. Such implementations do, nonetheless, come with a noticeable performance impact and require some non-trivial changes to existing rendering pipelines.

8.1 Future Work

There are several potential improvements that might be able to reduce the impact on frametimes and memory footprint without significantly altering the obtained visual effects. They were beyond the scope of this thesis, but here are some ideas that could be used as starting points for future research.

8.1.1 Stokes-Mueller Calculus in Denoised Monte Carlo Ray Tracing

As discussed in section 7.2, incorporating the Stokes-Mueller calculus in a real-time ray-traced Monte Carlo renderer would not be as simple as just changing the ray payloads and shading calculations. It would also require temporal accumulation and denoising of polarization data, and preferably without significantly increasing the memory footprint. Investigating how to optimally implement these changes would be another step towards making polarization rendering a feature in real-time ray-traced applications.

8.1.2 Performance Study in a Real-World Application

To see just how much of a performance impact simulated polarizing filters have on an application where its visual effects are desirable, it would be best to implement them in that application and see how much of an impact it has on the performance.

8.1.3 Simplifying the Polarizing Fresnel Function

The calculations required in the polarizing Fresnel function are significantly more complex than the ones needed in Schlick's approximation. Simplifying this function while keeping the attributes needed for the polarizing filters to have the same visual effects would likely increase the performance. It might, for example, be feasible to implement lookup tables with pre-calculated $F_\perp$, $F_\parallel$, and $\delta_\perp - \delta_\parallel$ values of common materials and then interpolate between them instead of recomputing them each frame (similar to the implementation by Wolff and Kurlander [40], but without as many lookup tables in scenes with many different materials).

8.1.4 Simplifying the Polarization Data

The ray payload that was used by the polarization implementations in this thesis used 48 bytes more data than the payload used by the Baseline renderer. As it is recommended to keep ray payloads small for performance reasons, finding ways to reduce this size increase would likely be a worthwhile endeavor. If the purpose of tracking polarization is to simulate polarizing filters, then it might be possible to simplify the polarization calculus and the Stokes vector representation. The last Stokes parameter can be removed from at least the primary ray's payload without any impact on the visual effects (since it has no impact on polarizing filters). As mentioned in section 7.2, it might be sufficient to use the same Stokes vector for all color channels when dealing with dielectric materials.

8.1.5 Selective Polarization

A drawback of the Hybrid approach used in this thesis is that the lack of polarization data from reflections is apparent when polarizing filters and mirrors-like surfaces are involved (as could be seen in Figure 4.3c). One way to improve realism without significantly increasing the amount of polarization-capable rays would be to selectively use those rays based on the material properties at the intersection point, instead of just on the recursion depth. This might best be accomplished using the roughness as the qualifier on an object-by-object basis and not a pixel-by-pixel one, since the latter would cause artifacts around the threshold values on a surface when a polarizing filter is used.

8.1.6 Polarizing Filters Without Polarization Parameters

This thesis relied on an existing generalized polarization calculus from the optics field in order to simulate a polarizing filter. Video games and similar applications often strive to *look* realistic without necessarily *being* realistic, and most of the effects of a polarizing filter might be possible to emulate without having to implement any polarization parameters (such as Stokes vectors) at all. A good place to start with such an approach might be a viewpoint rotation–dependent BRDF that reduces the brightness of reflections in one axis but increases them in the other.

References

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. CRC Press, Boca Raton, FL, 3rd edition, 2012.

[2] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '68, pages 37–45, Atlantic City, NJ, 1968. Association for Computing Machinery. doi: 10.1145/1468075.1468082.

[3] Ben Archard, Sergei Karmalsky, Oles Shyshkovtsov, and Dmitry Zhdan. Exploring the ray traced future in 'Metro Exodus', 2019. URL: https://gdcvault.com/play/1026159/Exploring-the-Ray-Traced-Future. [Accessed: June 12, 2020]. Presented at GDC 2019.

[4] Clayton Ashley. Photo mode: A tribute. *Polygon*, July 2018. URL: https://www.polygon.com/videos/2018/7/9/17528640/photo-mode-god-of-war-doom-no-mans-sky-compilation. [Accessed: June 12, 2020].

[5] Michael Bass, editor. *Handbook of Optics*, volume 2. McGraw-Hill, New York, NY, 2nd edition, 1995.

[6] Nir Benty, Kai-Hwa Yao, Tim Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. The Falcor rendering framework (version 3.2.2), Apr. 2018. URL: https://github.com/NVIDIAGameWorks/Falcor. [Accessed: June 12, 2020].

[7] Max Born and Emil Wolf. *Principles of Optics*. Pergamon Press, Oxford, England, 4th edition, 1970.

[8] Charly Collin, Sumanta Pattanaik, Patrick LiKamWa, and Kadi Bouatouch. Computation of polarized subsurface BRDF for rendering. In *Proceedings of Graphics Interface*, GI '14, pages 201–208, Montreal, Canada, May 2014. Canadian Information Processing Society.

[9] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM SIGGRAPH Computer Graphics*, 15(3):307–316, Aug. 1981. doi: 10.1145/965161.806819.

[10] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 137–145, New York, NY, Jan. 1984. Association for Computing Machinery. doi: 10.1145/800031.808590.

[11] Nvidia Corporation. FrameView, June 2019. URL: https://www.nvidia.com/en-us/geforce/technologies/frameview/. [Accessed: June 12, 2020].

[12] Alex Dunn. Tips and tricks: Ray tracing best practices. *Nvidia Developer Blog*, Mar. 2019. URL: https://devblogs.nvidia.com/rtx-best-practices/. [Accessed: June 12, 2020].

[13] Epic Games. Unreal Engine Sun Temple, Open Research Content Archive, Oct. 2017. URL: https://developer.nvidia.com/ue4-sun-temple. [Accessed: June 12, 2020]. Used under CC BY-NC-SA 4.0.

[14] Dennis H. Goldstein. *Polarized Light*. CRC Press, Boca Raton, FL, 3rd edition, 2011.

[15] Holger Gruen, Michiel Roza, and Jon Story. "Shadows" of the Tomb Raider: A ray tracing deep dive, Mar. 2019. URL: https://www.gdcvault.com/play/1026163/-Shadows-of-the-Tomb. [Accessed: June 12, 2020]. Presented at GDC 2019.

[16] Eric Haines and Tomas Akenine-Möller, editors. *Ray Tracing Gems: High-Quality and Real-Time Rendering With DXR and Other APIs*. Apress, Berkeley, CA, 2019. doi: 10.1007/978-1-4842-4427-2.

[17] Eric Heitz. Understanding the masking-shadowing function in microfacet-based BRDFs. *Journal of Computer Graphics Techniques*, 3(2):48–107, 2014.

[18] Nicholas Hull. Nvidia Arcade sample scene, 2017.

[19] R. Clark Jones. A new calculus for the treatment of optical systems. *Journal of the Optical Society of America*, 31(7):488–493, July 1941. doi: 10.1364/JOSA.31.000488.

[20] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, Aug. 1986. Association for Computing Machinery. doi: 10.1145/15922.15902.

[21] Yahong Li, Yuegang Fu, Zhiying Liu, Jianhong Zhou, Peter J. Bryanston-Cross, Yan Li, and Wenjun He. Three-dimensional polarization algebra for all polarization sensitive optical systems. *Optics Express*, 26(11):14109–14122, May 2018. doi: 10.1364/OE.26.014109.

[22] Amazon Lumberyard. Amazon Lumberyard Bistro, Open Research Content Archive, July 2017. URL: https://developer.nvidia.com/orca/amazon-lumberyard-bistro. [Accessed: June 12, 2020]. Used under CC BY 4.0.

[23] Dmitry Makeev. Photographs with polarization filter, Dec. 2019. URL: https://commons.wikimedia.org/wiki/File:Test._Photographs_with_polarization_filter._img_01.jpg. [Accessed: June 12, 2020]. Used under CC BY-SA 4.0.

[24] Mimigu. A render of a few spheres, Feb. 2009. URL: https://commons.wikimedia.org/wiki/File:BallsRender.png. [Accessed: June 12, 2020]. Used under CC BY 3.0.

[25] Michal Mojzík, Tomáš Skřivan, Alexander Wilkie, and Jaroslav Křivánek. Bidirectional polarised light transport. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*, EGSR '16, pages 97–108, Dublin, Ireland, 2016. The Eurographics Association. doi: 10.2312/sre.20161215.

[26] Sairam Sankaranarayanan. *Modelling polarized light for computer graphics*. Ph.D. dissertation, Iowa State University, Ames, IA, 1997. doi: 10.31274/rtd-180813-13307.

[27] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994. doi: 10.1111/1467-8659.1330233.

[28] Jan Schmid and Johannes Deligiannis. It just works: Ray-traced reflections in 'Battlefield V', Mar. 2019. URL: https://gdcvault.com/play/1026282/It-Just-Works-Ray-Traced. [Accessed: June 12, 2020]. Presented at GDC 2019.

[29] Peter Shirley. *Physically based lighting for computer graphics*. Ph.D. thesis, University of Illinois, Urbana, IL, 1991.

[30] William A. Shurcliff. *Polarized Light: Production and Use*. Harvard University Press, Cambridge, MA, 1962.

[31] George Gabriel Stokes. On the composition and resolution of streams of polarized light from different sources. *Transactions of the Cambridge Philosophical Society*, 9: 399–416, 1852.

[32] Jorge Sánchez Almeida. Radiative transfer for polarized light: Equivalence between Stokes parameters and coherency matrix formalisms. *Solar Physics*, 137(1):1–14, Jan. 1992. doi: 10.1007/BF00146572.

[33] Moritz Völker and Bernd Hamann. Real-time rendering of cut diamonds. Technical report, University of California, Davis, CA, 2013.

[34] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, EGSR'07, pages 195–206, Grenoble, France, June 2007. Eurographics Association.

[35] Andrea Weidlich and Alexander Wilkie. Realistic rendering of birefringency in uniaxial crystals. *ACM Transactions on Graphics*, 27(1), Mar. 2008. doi: 10.1145/1330511.1330517.

[36] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, page 14, Chicago, IL, Aug. 1979. Association for Computing Machinery. doi: 10.1145/800249.807419.

[37] Alexander Wilkie and Andrea Weidlich. A standardised polarisation visualisation for images. In *Proceedings of the 26th Spring Conference on Computer Graphics*, SCCG '10, pages 43–50, Budmerice, Slovakia, May 2010. Association for Computing Machinery. doi: 10.1145/1925059.1925070.

[38] Alexander Wilkie and Andrea Weidlich. How to write a polarisation ray tracer. In *SIGGRAPH Asia 2011 Courses*, SA '11, pages 1–36, Hong Kong, Dec. 2011. Association for Computing Machinery. doi: 10.1145/2077434.2077442.

[39] Alexander Wilkie and Andrea Weidlich. Polarised light in computer graphics. In *SIGGRAPH Asia 2012 Courses*, SA '12, Singapore, Nov. 2012. Association for Computing Machinery. doi: 10.1145/2407783.2407791.

[40] Lawrence B. Wolff and David J. Kurlander. Ray tracing with polarization parameters. *IEEE Computer Graphics and Applications*, 10(6):44–55, Nov. 1990. doi: 10.1109/38.62695.

[41] Haiyang Zhang, Yi Li, Changxiang Yan, and Junqiang Zhang. Three-dimensional polarization ray tracing calculus for partially polarized light. *Optics Express*, 25(22): 26973–26986, Oct. 2017. doi: 10.1364/OE.25.026973.

[42] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and Sung-Eui Yoon. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum*, 34(2):667–681, May 2015. doi: 10.1111/cgf.12592.

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden