Photorealistic Virtual Reality

MASTER THESIS

Author: Ioannis Moschos
Supervisor: Georgios Papaioannou
June 2019
Department of Informatics
Athens University of Economics & Business
Greece

Abstract

Virtual reality (VR) describes an immersive, highly interactive simulated environment. Its principal objective is to suspend belief, providing a life-like, vivid experience. A way to accomplish this is by proper stimulation of the user's senses, e.g. sight, hearing, touch, smell and taste. In today's world, virtual reality is becoming increasingly mainstream, as the number of households owning a VR headset is constantly rising. In general, this equipment focuses on the visual and, occasionally, auditory aspects of VR, using stereoscopic displays and surround sound systems. Although a few of those headsets operate autonomously, the majority of them utilizes input acquired from computers, smart-phones and any other device capable of outputting the necessary data. The information stream that corresponds to the visual stimulus should be supplied at a high framerate and accurately emulate the user's motion. Additionally, it should provide a realistic representation of the surrounding environment, abiding by the laws of physics. On one hand, this implies inclusion of phenomena and events such as gravity, collisions, etc. On the other hand, this translates into a physically correct lighting model, i.e. photorealism. Due to the cost of stereoscopic rendering and photorealism, most VR applications use inferior graphics, reducing the level of immersion in the process. For the purposes of this thesis, we develop a photorealistic VR renderer designed to operate as a stepping stone for future experimentation and testing on novel techniques and ideas, that aim to tackle the current limitations of photorealistic virtual reality. To accommodate this, a graphics engine was programmed from scratch for the purpose of hosting our specialized renderer.

Περίληψη στα Ελληνικά

Ο όρος εικονική πραγματικότητα (VR) περιγράφει ένα εντυπωσιακό, διαδραστικό προσομοιωμένο περιβάλλον. Ο κύριος στόχος της είναι να πείσει τον χρήστη ότι βρίσκεται μέσα σε ένα πραγματικό περιβάλλον. Ένας τρόπος για να επιτευχθεί αυτό είναι με την κατάλληλη διέγερση των αισθήσεων του, όπως για παράδειγμα την όραση, την ακοή, την αφή, την όσφρηση και την γεύση του. Στη σημερινή εποχή, η εικονική πραγματικότητα καθίσταται ολοένα και περισσότερο δημοφιλής, καθώς ο αριθμός των ανθρώπων που κατέχουν γυαλιά VR αυξάνεται συνεχώς. Γενικά, ο εξοπλισμός αυτός επικεντρώνεται στις οπτικές και, ενίοτε, ακουστικές πτυχές του VR, χρησιμοποιώντας στερεοσκοπικές οθόνες και συστήματα στερεοφωνικού ήχου. Παρόλο που μερικά από αυτά τα εξαρτήματα λειτουργούν αυτόνομα, η πλειοψηφία τους λαμβάνει οπτικοακουστικές πληροφορίες από υπολογιστές, smart-phones και κάθε άλλη συσκευή ικανή να παράγει τα απαραίτητα δεδομένα. Η ροή των πληροφοριών που αντιστοιχεί στα οπτικά ερεθίσματα θα πρέπει να παρέχεται σε υψηλή συχνότητα και να μιμείται με ακρίβεια τις κινήσεις του χρήστη. Επιπλέον, θα πρέπει να παρέχει μια ρεαλιστική αναπαράσταση του περιβάλλοντος χώρου, τηρώντας τους νόμους της φυσικής. Από τη μία πλευρά, αυτό συνεπάγεται τη συμπερίληψη φαινομένων όπως η βαρύτητα, συγκρούσεις κλπ. Από την άλλη πλευρά, αυτό μεταφράζεται σε ένα φυσικά σωστό μοντέλο φωτισμού, δηλαδή στον φωτορεαλισμό. Λόγω του κόστους της στερεοσκοπικής, φωτορεαλιστικής φωτοαπόδοσης, οι περισσότερες εφαρμογές VR χρησιμοποιούν κατώτερα γραφικά, μειώνοντας την οπτική τους πιστότητα. Για τους σκοπούς της παρούσας διατριβής, αναπτύσσουμε ένα φωτοαποδότη φωτορεαλιστικού VR σχεδιασμένο για να λειτουργήσει ως ορόσημο για μελλοντικούς πειραματισμούς και δοκιμές σε νέες τεχνικές και ιδέες που στοχεύουν στην αντιμετώπιση των σημερινών περιορισμών της φωτορεαλιστικής εικονικής πραγματικότητας. Για τη φιλοξενία αυτού του έργου μας μια νέα μηχανή γραφικών προγραμματίστηκε από το μηδέν.

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my supervisor, Georgios Papaioannou, Assistant Professor in the Department of Informatics of Athens University of Economics and Business, for the continuous guidance and support throughout the completion of this thesis.

I am grateful to my friend Charalampos Katagis, university student of Computer Engineering and Informatics Department in the University of Patras. Our common interest and enthusiasm in the worlds of video games and software development have sparked many interesting and insightful conversations.

Last but not least, I would like to thank my family: my parents and my brother, for their unconditional support throughout my studies and my life in general.

In memory of Efthimia Bellou

Contents

Chapter 1

Introduction

1.1 3D Computer Graphics and Photorealism

Computer graphics are often mentioned in discussions that concern animated films and video games. In spite of that, this term refers to the process of generating images via computers, regardless of context and use. Production requirements depend, to a large extent, on the application scope. Typically, fields that benefit from computer graphics include art, illustration, simulation, entertainment as well as any other visually rich domains. In some cases, such as 3D video games, visual fidelity requires complete simulation of a 3D scene, i.e. the virtual environment from which the generated imagery is constructed. This area is known as "3D Computer Graphics".

1.1.1 Photorealistic Rendering

Rendering refers to the process of producing an image, photorealistic or not, from a 2D or 3D scene. The data that forms this scene may include models, lights, virtual cameras, textures, etc. A renderer is software that performs rendering, provided that it is supplied with the proper information. A prerequisite for a photorealistic renderer is that it should be able to simulate complex lighting phenomena that occur inside the scene. Those generally depend on the scene's setup, e.g. light entity and geometry positions, material and ambient information, etc.

Photorealistic rendering in 3D computer graphics, is the ability to generate synthetic images with a detailed visual representation similar to those obtained from a real-life camera. Nowadays, its concept is applied to elevate realism in entertainment media, artistic expression, etc. Even in cases where one pursues non-realistic results, e.g. cartoon-like worlds, some of photorealism's principles may be employed to aesthetically enhance the outcome. Figure 1.1 provides some examples on the application of photorealism.

FIGURE 1.1: Applications of photorealism. Top-Left: Animated movie, "Gantz: O". Top-Right: Oil canvas, "John's Diner with John's Chevelle". Bottom-Left: Video game, "Forza Horizon 4". Bottom-Right: Video game, "BOWLBO" (Sources: Top-Left: [OKS16], Top-Right: [Bae07], Bottom-Left: [Gam18], Bottom-Right: [O'N19])

Photorealism is usually referred to as the "holy grail" of 3D computer graphics. Researchers on this subject focus on achieving a good equilibrium between photorealism and performance. Unfortunately, demands of photorealistic rendering exceed those of plain simulation of geometric information inside virtual worlds. To achieve adequate results, photorealistic renderers perform intricate light calculations based on surface materials and atmospheric properties. To include more natural phenomena, translucent materials, liquid and volumetric information is needed. Hence, as shown in Figure 1.2, renderers that are given a more detailed description of those attributes, are capable of producing a wider range of photorealistic scenes, albeit at a significant performance cost.

1.1.2 The Cost of Photorealism

Generally, achieving a high level of photorealism is too costly for real-time interactive applications. Therefore there are different and contrasting goals and compromises when targeting real-time (online) applications or non-real-time (offline) ones. The first category generally tries to inexpensively imitate demanding offline techniques "tricking" viewers, often convincing them of the visual fidelity of its results. The second uses a range of mathematics and physics concepts, based on geometric optics, to simulate light's behaviour as it travels through the virtual world and is capable of generating

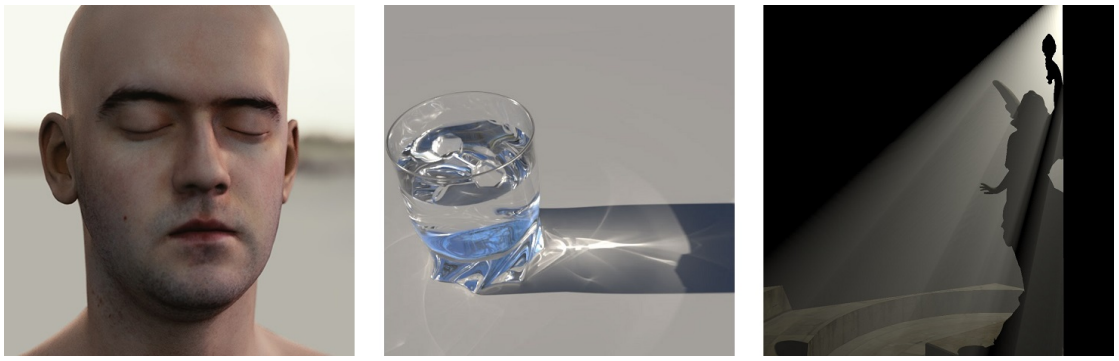

FIGURE 1.2: A wider range of phenomena. Left: Realistic skin. Middle: Liquid & caustics. Right: Volumetric lighting. (Sources: Left: [Hof16], Middle: [Duv18], Right: [Mos17])

results indistinguishable from real photographs. Both online and offline rendering techniques (see Figure 1.3) can be highly sophisticated and provide various performance and quality optimizations in their respective "territory".

FIGURE 1.3: Offline and online rendering results. Left: Offline technique. Right: Online technique. (Sources: Left: [Tra06], Right: [Gam16] (Adapted: Zoomed-in, ©2019, Epic Games Inc.)

Through use of remote cloud computing and the resurgence of fields in computer science such as machine learning, offline techniques are getting faster, approaching real-time rendering capabilities. For example, techniques that required hundreds of thousands of iterations to "converge"[1] to a photorealistic outcome now produce results with fewer and fewer iterations using the power of neural networks and AI denoising[2] ([LMH+18], [CUM18]). Other optimization examples include remote GPU calculations, through cloud computing ([Cor18a]). It is safe to say that in near future, real-time interactive and highly photorealistic 3D applications will be accessible to ordinary consumers.

[1] Approach a physically correct result
[2] Image noise removal. In case of rendering, noise originates mainly from under-sampling

1.2 Virtual Reality

Typically, in 3D applications, a sequence of rendered pictures is displayed on a stable frame rate, yielding a visual representation of the virtual world. Specifically, observers see a flat, two-dimensional rendition on their computer monitor, TV, cinema display or smart-phone. Although this is not necessarily a "lackluster" experience, it is far from its real-life counterpart. Human experience is not limited to monoscopic visual stimuli. Incorporating more senses, e.g. auditory, smell, touch, etc. creates the perception of being physically present in a non-physical world.

Virtual reality is an immersive, artificial environment created with software and presented to the user in such a way that the user suspends belief and accepts it as real. It mainly incorporates visual and auditory senses, though, in its principle, it aims to completely immerse[3] users into the virtual environment, e.g. by integrating more senses such as smell, taste or touch. Another important trait of VR is its highly interactive nature. In fact, lack of interactions and a more passive experience may break said immersion. Therefore, users are typically given an artificial presence, e.g. body and limbs, and the freedom to perform simple tasks such as movement and interaction with objects through bodily actions and gestures.

1.2.1 Reality vs Virtuality

Reality–Virtuality Continuum

Paul Milgram first introduced the concept of the reality–virtuality continuum [MTUK95]. This notion describes all possible variations and compositions of real and virtual objects. Between a real and a completely virtual environment, there are those that Milgram defined as mixed reality (MR) environments. Inside these, real and virtual objects are presented together within a single display. As depicted in Figure 1.4, MR environments may be described as augmented reality (AR) and/or as augmented virtuality (AV) ones.

An AR environment is mostly real world, but with virtual objects superimposed upon or composited with it. AR acts supplementary to reality instead of completely replacing it. AV, on the other hand refers to live blending of real world information into a virtual environment. Examples include projecting input devices, e.g. keyboards, or other surrounding room information inside the virtual world. In this thesis, we are mostly interested in immersing observers into a completely virtual environment, nevertheless, this does not mean that we will not be using information derived from real environments.

[3] Engross users in a non-physical world

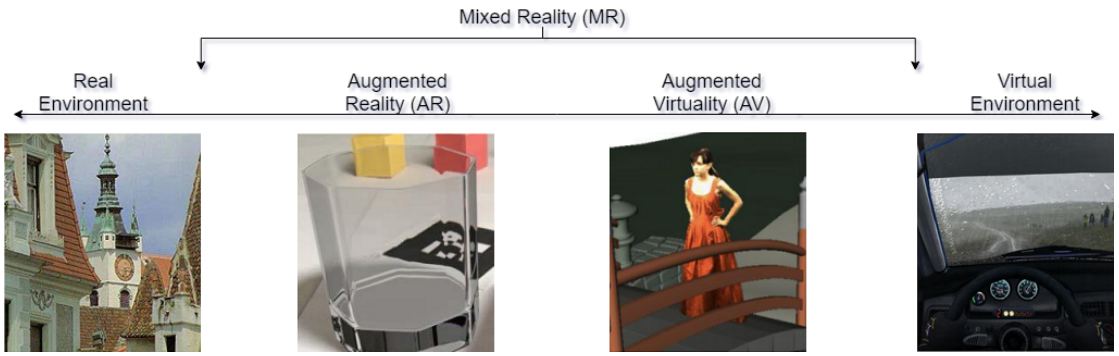

FIGURE 1.4: The Reality–Virtuality Continuum. (Sources: Adapted from [MTUK95], [Pap18c] & [Kau16])

1.2.2 History of VR

Although Jaron Lanier first coined the term "Virtual Reality" in 1987, the concept of an alternate reality that stimulates the senses of the observer, was of great interest to many philosophers and intellectuals. Artwork such as panoramic paintings, drawn as early as the 1800s, attempt to place admirers inside a historical scene, such as the one shown in Figure 1.5.

FIGURE 1.5: Raevsky Battery during the Battle of Borodino (1812). (Sources: Photograph: [Sha12], Painting [Rou12])

Actual VR technological accomplishments and milestones, begin before mid-1800s and span throughout the 19th, 20th and early 21st centuries. The course and history of VR grants an insight into the current state and progress of this field. Selectively chosen, key events and developments are presented in chronological order, starting from primitive

stereoscopic visualization tools, to the current era and climate of virtual reality (Sources: [Dor17], [Bar18] and [vrs17b]).

1838 - Stereoscopic Photos and Viewers

In 1838, Charles Wheatstone [Whe38] demonstrated by using a stereoscope, that seeing two slightly different images from each eye gives viewers a sense of depth and immersion. Shown in Figure 1.6, a stereoscope is a device for viewing a stereoscopic pair of separate images. Devices that are designed based on the principles of stereoscopes include the "View-Master" and most of modern VR head mounted displays used by smart-phones, such as, the "Google Cardboard".

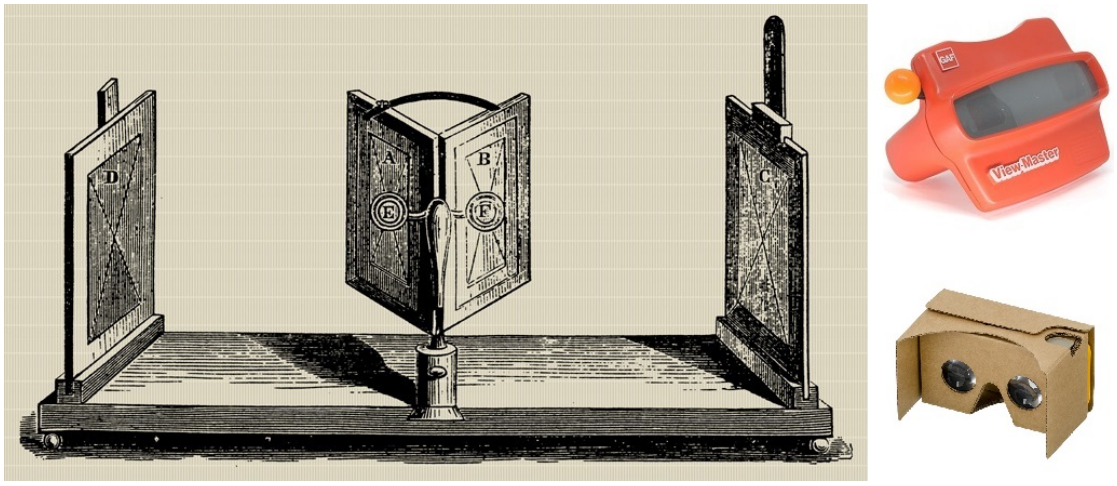

FIGURE 1.6: Stereoscopes. Left: Reflection stereoscope design. Top-Right: ViewMaster. Bottom-Right: Google Cardboard. (Sources: Top-Right: [Kob16], Bottom-Right: [Amo15])

1950 - Sensorama

Morton Heilig, in mid 1950s, developed the "Sensorama". This device was an all-in-one VR experience designed to stimulate almost all of the user's senses. It was an arcade cabinet-theatre (see Figure 1.7), that featured stereoscopic 3D display, fans, scents generators and a vibrating chair, objectively, an ambitious product which aimed to fully immerse the spectator to the film.

1960 - Telesphere Mask

In addition, Morton Heilig invented the "Telesphere Mask" (see Figure 1.7). This was the first head-mounted display (HMD). It featured stereo 3D vision and sound and a wide field of view (FOV)[4]. Contrary to today's VR HMDs, it lacked motion tracking and its content was not interactive.

[4]Extent of the visible part of the scene displayed on the screen

1961 - Headsight

In 1961, two Philco Corporation engineers, Comeau and Bryan, developed the "Headsight". This device, shown in Figure 1.7, was more comparable to modern VR HMDs as it incorporated a magnetic motion tracking system. Alas, it was restricted to military applications and specifically to distant viewing of dangerous situations. A remote camera was synchronized, via a motion tracking system, with user's orientation and movements, allowing them to look around the distant environment.

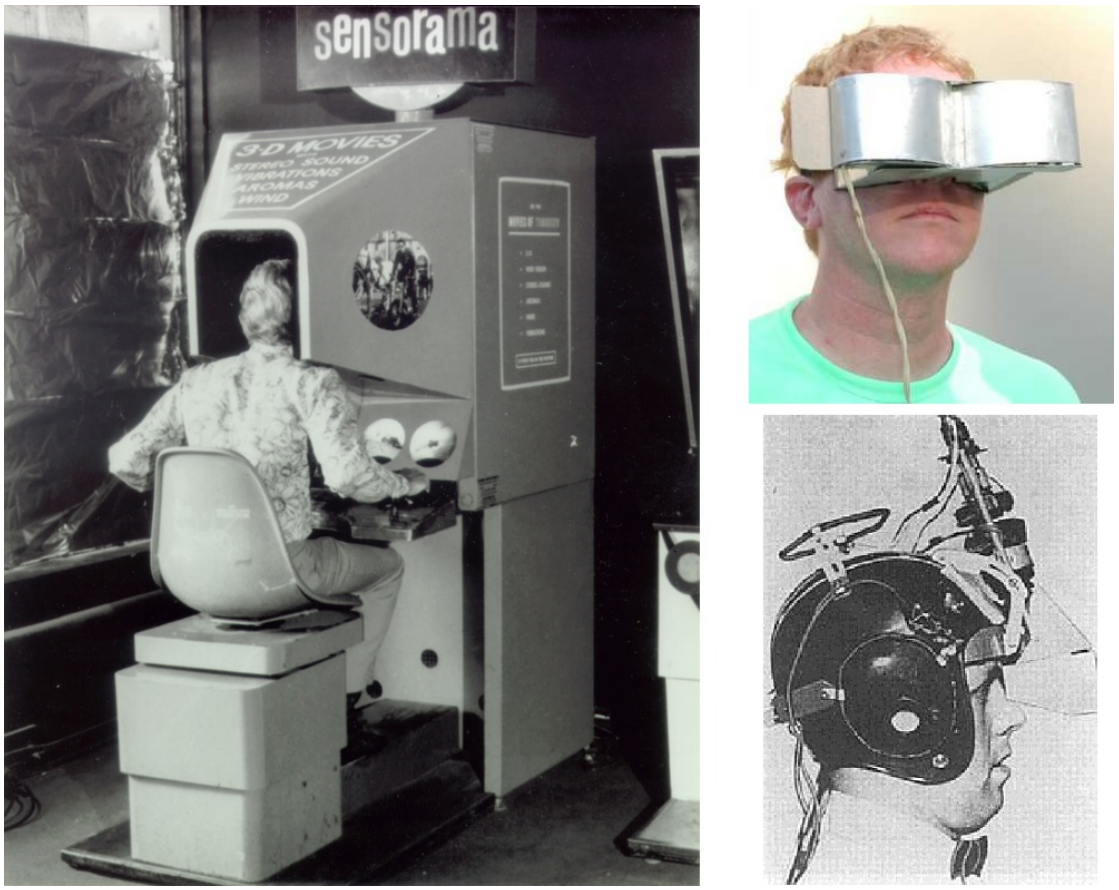

FIGURE 1.7: Virtual reality inventions. Left: Sensorama. Top-Right: Telesphere Mask. Bottom-Right: Headsight.

1965 - The Ultimate Display

Ivan Sutherland described the "Ultimate Display" [Sut65] as follows: "The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked." In this day and age, his vision is considered to be the fundamental blueprint for VR.

1968 - Sword of Damocles

"Sword of Damocles", shown in Figure 1.8, was a lab project created by Ivan Sutherland and his student Bob Sproull. This HMD was the first with the capability to display computer generated images. Due to its size, this device had to be hanged from the ceiling and users had to be strapped into it. Being connected to a computer, "Sword of Damocles" could display generated graphics, but at that time those were nevertheless limited to simple 3D model wire-frames.

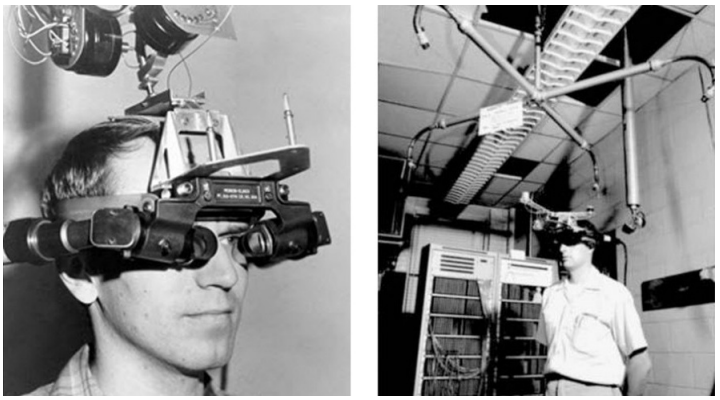

FIGURE 1.8: Sword of Damocles.

1987 - Virtual Reality

Jaron Lanier, founder of visual programming lab (VPL) coined the term "Virtual Reality". His company assembled virtual reality gear, such as the "Dataglove", the "EyePhone HMD" and a full body outfit named "DataSuit" (see Figure 1.9). Those were the first virtual reality gloves, goggles and suit sold by a company, however, their price range was quite high for ordinary consumers.

FIGURE 1.9: VPL's products. Left: Datasuit. Right: EyePhone & Dataglove. (Sources: Left: [Pap99a], Right: [Pap99b])

1991 - Public Access of Virtual Reality

Although consumers could not yet own a VR system in their household, VR gear made its appearance into the local arcades. The Virtuality Group had launched a range of arcade games and machines and thus, the average person could now enjoy real-time immersive stereoscopic 3D video games, some of which featured multi-player capabilities.

1993 and 1995 - Early Hype in Consumer-grade Products

Sega, in 1993, and Nintendo, in 1995, both announced VR products shown in Figure 1.10. Unfortunately, those products failed to reach to fruition due to technical difficulties and lack of software support. It was still too early to build hype for privately owned VR-ready equipment. This technology was still immature and expensive.

FIGURE 1.10: Early hyped virtual reality products. Left: Nintendo's Virtual Boy. Right: Sega VR. (Sources: Left: [Amo11], Right: [Squ16])

2012 - Kickstarting the HMD-based VR era

Palmer Luckey launched a Kickstarter campaign for Oculus Rift (Figure 1.11) which raised over $2.4 million. The HMD featured was the "Oculus Rift Development Kit 1". Technical specifications of this device included 6 degrees of freedom (DoF)[5] low latency head tracking, a field of view of 110° diagonal and 80° horizontal and a resolution of 1280x800 (640x800 per eye).

FIGURE 1.11: Oculus Rift campaign. (Sources: Left: [Sta13], Right: [Ocu12])

[5]Moves up and down (1), left and right (2), forwards and backwards (3). Swivels left and right (4). Tilts forwards and backwards (5). Pivots side to side (6)

2014 to Present day

Two years later, Facebook bought the Oculus VR company, meaning that virtual reality was now backed by a colossal enterprise. Other major players decided to step into the newly formed VR hype, with HTC unveiling the "HTC Vive", Sony announcing "Project Morpheus", Google releasing the "Cardboard", and Samsung presenting the "Samsung Gear" VR headset (Figure 1.12). Many people, researchers, engineers and hobbyists started experimenting with the capabilities of the HMD-based VR.

FIGURE 1.12: Modern head-mounted displays. Left: PlayStation VR. Middle: Samsung Gear. Right: HTC Vive. (Sources: Left: [Amo17], Middle: [Ele15], Right: [HC16])

In the present climate, a host of companies develop their own VR hardware and software support. Researchers concentrate their efforts on achieving an immersive and comfortable VR experience. Multiple campaigns, similar to the one Palmer Luckey launched in 2012, have emerged, each providing their own novel ideas and concepts that could benefit VR. The future of virtual reality seems bright and exciting.

1.2.3 Application Areas

Contrary to popular belief, virtual reality applications are not limited to interactive video game worlds and the entertainment industry. In fact, industrial, medical, educational, therapy, design and a range of other fields could greatly benefit from a transition from monoscopic to stereoscopic VR visualization (Sources: [vrs17a]).

Healthcare

VR in healthcare could be used for both training purposes and as a supplementary tool for doctors and surgeons. Inexperienced medical interns could be taught via VR medical training which is definitely safer than operating on real patients. As currently performed during remote tele-surgery, surgeons could operate at a different location than that of patients, with help of robotics and VR visualization. This facilitates in keeping patients' surrounding environment clean and sterilized and prevents the spread of infectious diseases to medical staff. VR visualization could also be utilized as a

diagnostic tool to assist doctors with the diagnosis procedure, thus eliminating the need of any unnecessary invasive procedures or surgery.

Education

The entertainment aspects of VR are fairly evident. Teaching and learning could really benefit from the interactive and "fun" nature of VR. For example, students could explore 3D simulated solar systems, navigate anatomies of humans and animals, perform collaborative activities in shared 3D worlds and be trained within virtual simulators with equipment that their schools or universities do not posses.

Engineering

Engineers often use 3D modelling tools during the design process of their projects for visualization and evaluation purposes. That being said, VR imaging could help them identify imperfections and possible dangers before the beginning of the implementation stage. Changes could be implemented locally with simple hand movements and, with proper software support, many engineers could work collaboratively on a shared 3D space.

Scientific Visualization

Scientists generally struggle to communicate abstract concepts and ideas to an audience that is not necessarily familiar with their research. VR could provide a means of interaction with their work, e.g. seeing a simulated complex structure at different angles. This could generate more interest and traction to academic research and the scientific sector in general.

Military

Nowadays, military applications of VR are found in the army, navy and the air force. They include battlefield, vehicle and flight simulators, medical training, virtual bootcamps and combat visualization. Another important relevant application of VR in the military concerns sufferers of battlefield caused trauma and PTSD[6]. Through VR, they are taught how to cope with their symptoms in a "safe" environment, reducing them in the long run.

1.2.4 The Cost of Virtual Reality

From Mono to Stereo Rendering

Stereopsis is a trait of binocular vision found in humans and some animals that enables them to perceive depth and 3D-structure information. Interpupillary distance between

[6]Post-traumatic stress disorder

eyes results in two slightly different images being projected to each retina. These spatial differences are processed in the visual cortex of the brain to yield the aforementioned depth and structural perception [HR96]. In reality, those mechanisms are much more complex and deep, though, for our work, we mainly focus on replicating visual input (eye vision) rather than dealing with the way this information is processed (brain function).

In VR, stereopsis is a critical factor of immersion. Without it, the world appears somewhat flat, as if it was observed monoscopically. Therefore, transitioning to stereoscopic rendering is necessary for achieving the perception of being surrounded by a simulated environment. This brings about an enormous computational cost. In fact, without any optimizations, stereo rendering requires double the amount of calculations, as, simpler, naive approaches typically render the scene twice, each time from each eye's perspective.

Photorealism and VR

Photorealistic rendering is also key to maintaining the immersive nature of VR environments. Unfortunately, as noted in Section 1.1.2, photorealism comes at a great computational cost. VR requires online rendering of two high definition, high frame rate, distinct channels (eyes). Consequently, most VR renderers employ online rendering techniques that are less demanding, but produce inferior results in comparison to their offline counterparts.

Enriching the Experience

As mentioned before, VR environments are presented to the user in a way that the user suspends belief and accepts them as real environments. The stereoscopic renderer is only responsible of producing the visual stimuli. Detailed scene-graphs, physics and 3D sound systems should also be employed, adding up to the actual cost of VR. Furthermore, to incorporate more senses, additional hardware integration may be required. Eye-tracking technology, touch devices with haptic feedback, body positional and movement tracking, as well as scent generators are only a few examples, depicted in Figure 1.13. Therefore, it is safe to say that the cost of virtual reality is proportional to the level of immersion it is able to achieve.

1.3 Photorealistic Virtual Reality

In context of a VR stereoscopic renderer, photorealism is essential for immersion. As mentioned multiple times in this chapter, VR's aim is to persuade the user that the surrounding virtual environment is real. To accomplish this, it stimulates their senses in a familiar manner, similar to that real-life experience have taught them. On one hand,

FIGURE 1.13: Modern immersion-focused virtual reality products. Left: FeelReal. Top-Right: VRFREE. Bottom-Right: Cybershoes. (Sources: Left: [Fee19], Top-Right: [Sen18], Bottom-Right: [Gmb19])

low quality models, textures and animations, lack of game physics[7] and interactions with the environment and absence of stereophonic sound negatively affect the credibility of the VR world. On the other hand, missing shadows and reflections, unreal lighting due to lack of energy conservation laws, unnatural surface, liquid and atmospheric illumination can be detrimental to the visual fidelity and, as a consequence, to the user's immersion. This thesis focuses mostly on the latter, i.e. the graphics side, and aims to achieve a satisfactory level of photorealism in a stereoscopic context.

1.3.1 Is It Possible?

Nowadays, photorealistic virtual reality is to some extent possible due to the fact that real-time techniques have reached to an adequate level of photorealism. Nevertheless, they are limited to how closely they are able to simulate the physical phenomena, as they usually constitute inexpensive imitations of their offline counterparts. Close to real image synthesis in VR is no easy feat. Especially when applying, computationally heavy, offline rendering techniques that currently do not cope well even with real-time monoscopic renderers. That being said, there have been several software and hardware based optimizations lately ([LMH+18], [Cor18b]), that would be interesting to experiment on.

[7] Gravity, collisions, inertia

1.3.2 How Far Are We?

With enough computational power, it is certainly possible to create a highly photorealistic real-time VR environment. Unfortunately, consumer-grade technology and resources are not yet capable of supporting those demands. As of the time of writing, real-life levels of photorealism, in an interactive context, highly depend on achieving real time performance on currently offline rendering techniques that simulate light's transport and interactions.

Many researchers and engineers have come up with algorithms that optimize and speed up those techniques, though at the cost of introducing bias[8] and affecting the quality of the outcome. Also, most of those optimizations simulate an incomplete model of light's behaviour as they are forced to omit a large portion of the required calculations, otherwise they are unable to perform in real time. Donald Greenberg, one of fathers of CGI stated that technologies like AR, VR, etc. are still "babies" and in order to have a VR experience that is very close to reality, we will need around 20 years from now. However, he did emphasized that this is something that will inevitably happen and that VR is here to stay [Ska18].

[8] Difference between the estimated and true value. In rendering, bias refers to error in radiance approximation (discussed in Section 2.2.1)

Chapter 2

Background

2.1 Physically-based Rendering

Physically-based rendering (PBR) refers to a 3D computer graphics approach that seeks to accurately simulate the behaviour of light inside the virtual scene. This concept integrates photogrammetry, i.e. measurements of real-world materials using photographs, to replicate physical properties of actual surfaces. Many PBR work-flows achieve highly photorealistic results using a variety of mathematical models that replicate surface material properties, e.g. roughness, albedo and reflectance extinction (related to the metallic quality of conductors). Thus, it is able to approximate the material's contribution to any ray interacting with the modeled surfaces. Some of the aformentioned mathematical models are based on the idea that surfaces can be modeled as a collection of micro-cavities. These are known as "Microfacet" models and will be described later in depth.

Nowadays, many popular graphics/game engines use some variation of a PBR work-flow. Although PBR may be more elaborate than other approaches, it is not exclusive to offline rendering. As a principle, it focuses on energy conservation and light-surface interactions based on material properties. Different methods of computation (further discussed in Section 2.1.3) can benefit quality-wise following utilization of PBR, either when those are adjusted and optimized for online rendering or constitute rigorous simulations of light's transport.

2.1.1 PBR Materials

Prior to PBR, most renderers delivered results that were a combination of several illumination models. Diffuse lighting was used to calculate the amount of light reflected

from all directions of a surface based on the incidence angle of light. Specular illumination simulated the reflection of light (bright spots) on shiny objects, i.e. the specular highlights. Those two models as well as other lighting that originated from self-emitting surfaces or ambient information were all "hacked" together to generate outcomes that, most of the time, were not even physically plausible. Typically, materials of such implementations were a combination of diffuse, specular and emissions maps (see Figure 2.1), including other secondary properties. Most renderers implemented their own ad-hoc shading model. Consequently, exporting and importing assets between engines was tedious and impractical.

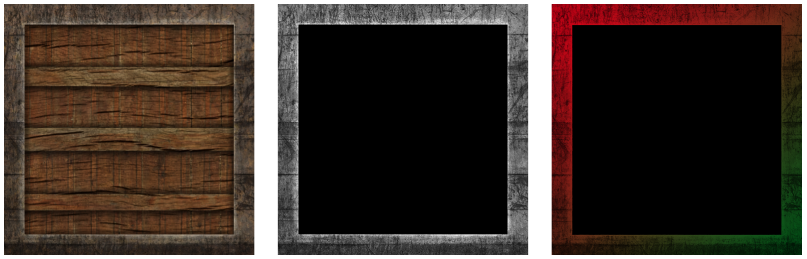

FIGURE 2.1: Textures of a material used by an ad-hoc renderer. (Sources: Left: [Vri16a], Middle: [Vri16b], Right: [Vri16e],)

On the other hand, PBR materials, i.e. materials that contain physical information and are used by a PBR work-flow, share most of the required data between variations of these models. Generally, physically-based materials are a combination of albedo, normal, metallic, roughness and some secondary maps such as emission, opacity and ambient occlusion (see Figure 2.2). These parameters are used by a variety of mathematical models to determine light's scattering after intersection with geometry occurs, e.g. reflection, transmission and sub-surface scattering. In conclusion, using PBR materials is not only essential for producing physically correct and photorealistic results, but it is also useful as it simplifies the transfer and re-usability of artistic work between rendering platforms.

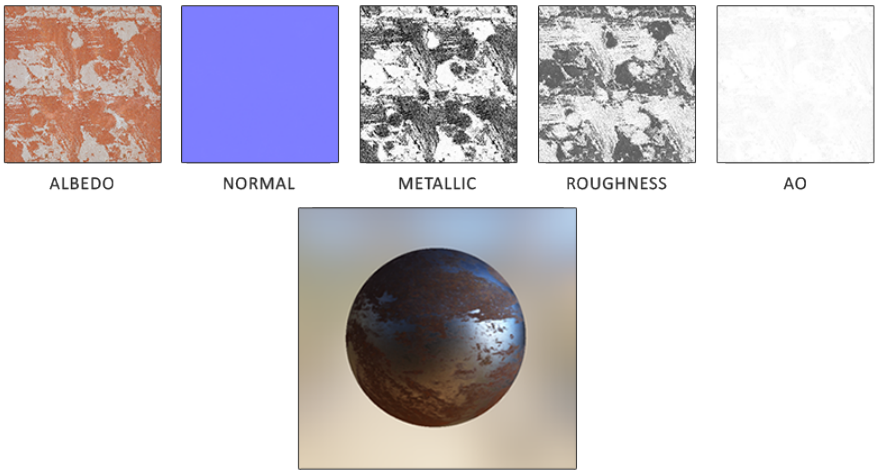

FIGURE 2.2: Physically-based textures and result. Top: Textures of a PBR material. Bottom: PBR of the combined textures. (Sources: [Vri16f])

2.1.2 Reflection Model

Mathematically speaking, PBR uses the bi-directional scattering distribution function (BSDF) to encompass both effects of reflection and transmission occurring at a surface. Additionally, to model sub-surface light transport, the bi-directional sub-surface scattering reflectance distribution function (BSSRDF) is utilized. However, for the purposes of this thesis, the current PBR model employed, uses a subset of the BSDF that is the bi-directional reflectance distribution function (BRDF). In other words, it only accounts for reflections based on physical material properties of surfaces.

Bidirectional Reflectance Distribution Function

The mathematical model that encapsulates the throughput of a material surface when reflecting light coming from a specific incident direction ω_i to an output direction ω_o is the aformentioned BRDF [NW09]. In general, the BRDF is defined as the ratio of the differential outgoing radiance to the differential irradiance. In practical terms, this function measures the relative contribution of light leaving the surface in a particular direction against the part of the irradiance for which a single input direction is (see Figure 2.3). For each specific point $\mathbf{x}$ of a surface, this equation depends on the incident and reflected light directions (Equation 2.1). For more information on BRDF fundamentals, the interested reader is referred to [PJH17, chap. 8-9].

$$BRDF(x, \omega_o, \omega_i) = \frac{dL(x, \omega_o)}{dE(x, \omega_i)} \tag{2.1}$$

where:

$dL(x, \omega_o) =$ *differential outgoing radiance*
$dE(x, \omega_i) =$ *differential incident irradiance*

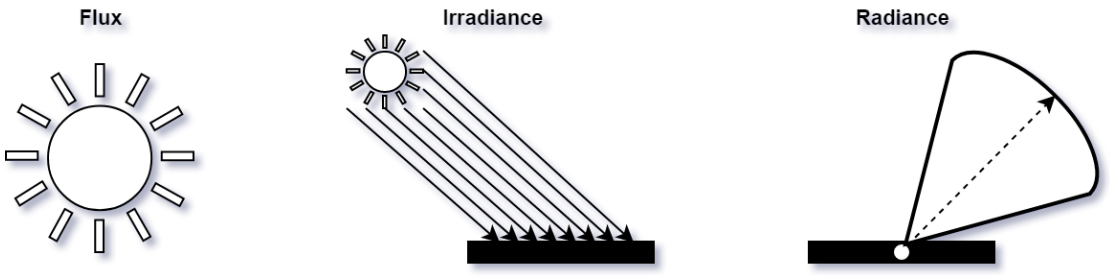

FIGURE 2.3: Flux: the total radiant energy emitted, reflected, transmitted or received, per unit time. Irradiance: the radiant energy received by a surface per unit area. Radiance: the radiant energy emitted, reflected, transmitted or received by a given surface, per unit projected solid angle.

An important property of a physically correct BRDF is the symmetry between the aformentioned incident direction ω_i and the output direction ω_o which is described by

the Helmholtz reciprocity. In practise, this means that if we were to swap those two directions we would still obtain the same result (see Equation 2.2).

$$BRDF(x, \omega_o, \omega_i) = BRDF(x, \omega_i, \omega_o) \tag{2.2}$$

Reflections

In reality, when a light ray collides with any surface, it is either transmitted or reflected. The final outcome depends on a range of different factors, such as the material properties of the surface, light's characteristics (e.g. wavelength) and the angle of incidence. There is an extremely small amount of artificially made materials that completely reflect or absorb light and are predominantly used in camera lens and military stealth technology production. Interaction of light with most of real-world's surfaces typically results in a combination of those two effects.

In computer graphics, it is typical to classify the reflection of light as either specular or glossy, according to the smoothness and resulting distribution of out-scattered light. Before delving further into the nature of these phenomena, we should first provide common terminology that appears in most cases. Incident light consists of incoming rays of light that hit a surface, whereas, reflected light refers to outgoing rays that are deflected. Surface normal in this case, is a vector which is perpendicular to a surface at the point of light's collision. The angle of incidence and reflection are considered between the respective ray directions and the normal vector. The following Figure 2.4 illustrates the aforementioned concepts.

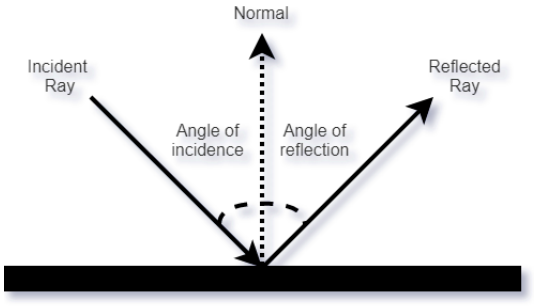

FIGURE 2.4: Basic concepts of a reflection.

Specular Reflections

Specular reflections are mirror-like reflections of light occurring during intersection with very smooth surfaces (see Figure 2.5). Their angle of incidence is equal to their angle of reflection resulting in a clear image of the surface's surrounding environment, similarly to how a mirror operates.

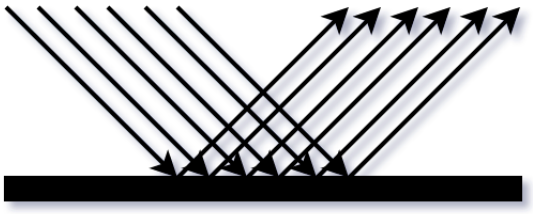

FIGURE 2.5: Specular Reflection.

Diffuse Reflections

Diffuse reflections happen when light rays are scattered in multiple directions in contrast to specular reflections (see Figure 2.5). In fact, diffuse reflection, also known as body or volume reflection is no reflection in the physical sense, as the light enters the surface and gets scattered back to the surface, very close to the point of entry. Depending on the scale of observation and the material structure, the re-emerging light can be considered exiting at the location of entry, appearing as being "reflected". Otherwise, light exits from a different location and the phenomenon is attributed the term subsurface scattering. Diffuse reflection is responsible for the intrinsic colour of the surfaces, i.e. their albedo.

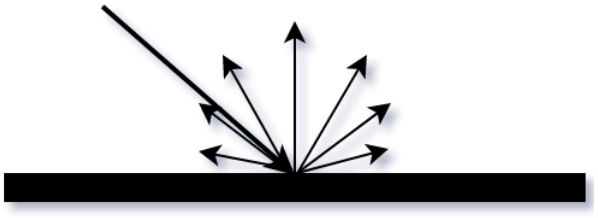

FIGURE 2.6: Diffuse Reflection.

Glossy Reflections

Glossy reflections are the most common in nature, as surfaces tend to be imperfect. Those reflections result in a distribution of reflected rays that depends on the material properties of the surfaces, and particularly their roughness. To simplify the process of understanding the nature of those reflections, we will use the aformentioned reciprocity property. As we mentioned during the discussion of the BRDF, it is acceptable to swap the incidence and output directions of light. As Figure 2.7 illustrates, light from multiple incident directions contributes to a single output direction thus blurring the reflected light.

The majority of real-world reflections on surfaces are a combination of the types analyzed above. Material properties are mainly responsible for how surfaces reflect light, as they determine their roughness and metalness. Therefore, an adequate approximation of the

FIGURE 2.7: Glossy Reflection.

effect that those properties have on the visual outcome, is required in order to correctly simulate the greatest part of these phenomena.

There is a wide range of BRDF functions that produce different results. Next, we provide a brief description of the BRDF models utilized for the purposes of this work:

GGX Specular Microfacet BRDF with Smith Geometric Masking

Regarding surface reflectance, we follow the principles of the Microfacet model [TS67, CT82, WMLT07a] which simulates the surface as several microscopic, planar and ideal reflectors. Each of those micro-cavities reflects incoming rays of light in only one particular outgoing direction. This model is mathematically represented via Equation 2.3. In this equation, the light direction **l** and the view direction **v** correspond to directions ω_i and ω_o respectively.

$$BRDF(l,v) = \frac{D(h,n)F(u,h)G(u,n)}{4(n\cdot l)(n\cdot v)} \tag{2.3}$$

where:

l = *light direction vector*
v = *view direction vector*
n = *surface normal vector*
$h = \frac{v+l}{||v+l||}$, *halfway vector, equals to microfacet normal*

In Equation 2.3, highlighted with red color are the following three important terms of the Cook-Torrance microfacet model which is ubiquitously used as the basis for the formulation of more specialized ones:

- **D**istribution Function

  This term represents the micro-facet density oriented along a specific direction. Specifically, it expresses the amount of micro-mirrors properly aligned so that light coming from a direction **l** is ideally reflected towards a direction **v**. These micro-surfaces are therefore aligned with the halfway vector **h** between **l** and **v**

(see Equation 2.3). For our implementation, we use the GGX (Trowbridge-Reitz) [WMLT07b] distribution function to describe dispersal of micro-facets on the surface (Equation 2.4, Figure 2.8).

$$D_{GGX}(m) = \frac{a^2}{\pi((n \cdot h)^2(a^2 - 1) + 1)^2} \tag{2.4}$$

where:

$a = roughness$

$n = surface\ normal\ vector$

$h = \frac{v + l}{||v + l||}$, *halfway vector, equals to microfacet normal*

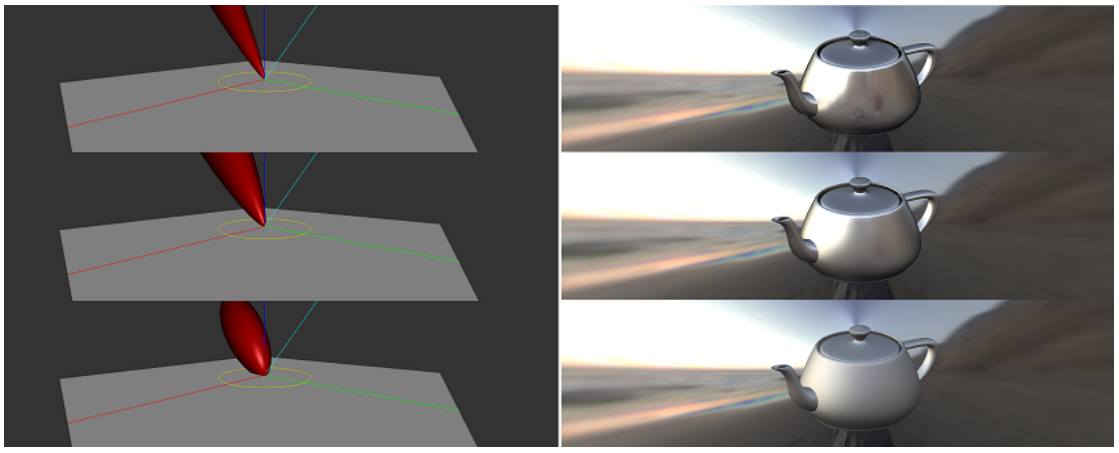

FIGURE 2.8: GGX distribution function's directionality and size of the lobe signifies the direction and width of the specular highlight. (Images rendered using Disney's BRDF Explorer [DS13a])

- **F**resnel

  The Fresnel equations govern the amount of light reflected and transmitted at an interface between two materials based on the index of refraction of the media, the angle of incidence and the polarization. For this term we use the Schlick Fresnel [Sch94] approximation formula (Equation 2.5). In computer graphics, simplified models such as the one we utilize are mainly applied for two reasons. Firstly in order to dispense with the dependence on refractive index and secondly, due to the fact that they are simpler to compute. The Schlick approximation specifies the effective reflectance of the surface with respect to the angle of incidence and the reflectance at normal incidence, i.e. when light enters the interface in a direction perpendicular to it. The non-reflected portion of the incident radiance is assumed to cross the interface and be transmitted (scattered, refracted or "diffusely" reflected).

$$F_{Schlick}(v, h) = F_0 + (1 - F_0)(1 - v \cdot h)^5 \tag{2.5}$$

where:

F_0 = *reflectance at normal incidence which depends on the surface's material properties*
v = *view direction vector*
$h = \frac{v+l}{||v+l||}$, *halfway vector, equals to microfacet normal*

- **G**eometric Shadowing

  Geometric shadowing describes any shadowing originating from the micro-facets. If light hits or leaves the surface at a low angle, almost parallel to it, the slope of the micro-facets can potentially intercept it. There have been a couple of models for this attenuation phenomenon, but here we rely on the Smith geometric shadowing function, which takes into account the mean slope of the micro-facets (i.e. expressed through the controlling variable of roughness). [Smi67, WMLT07b] (Equations 2.6 and 2.7).

$$G_{Smith}(l,v) = G_1(l)G_1(v) \tag{2.6}$$

where:

l = *light direction vector*
v = *view direction vector*

$$G_{1-GGX}(v) = \frac{2(n\cdot v)}{(n\cdot v)+\sqrt{a^2+(1-a^2)(n\cdot v)^2}} \tag{2.7}$$

where:

a = *roughness*
v = *input direction vector (light and view, see Equation 2.6 above)*
n = *surface normal vector*

Lambert Diffuse BRDF

Although the Lambert BRDF [Lam60] is the simplest body reflectance model, it is still widely used as its approximative results are sufficient for photorealistic rendering of body reflection, especially when properly balanced against surface reflection via the Fresnel term. Surface reflections described by this BRDF appear homogeneously bright and illuminated (Equation 2.8, Figure 2.9).

$$BRDF_{diffuse}(x) = \frac{p(x)}{\pi} \tag{2.8}$$

where:

$p(x)$ = *albedo, i.e. energy reflected, at point* x *on surface*

FIGURE 2.9: Lambert BRDF has no directional dependence and the size of its "lobe" (inner green circle), which is dependant on the surface's point albedo, determines the average amount of energy from all diffusion paths of the surface. (Images rendered using Disney's BRDF Explorer [DS13a])

2.1.3 Rendering Techniques

As mentioned in the introduction of this section, there are several techniques that accomplish physically correct and photorealistic rendering, each accompanied by their own advantages and disadvantages. Some of them focus on performance by sacrificing quality or completeness of effects that mimic the natural phenomena. Others aim to outright simulate light's transport in order to produce stunningly realistic results, albeit at the cost of real-time interactivity with the virtual environment.

Although shading is crucial for the visual outcome, it is only part of the equation. To create any results, photorealistic or of artistic intent, it is equally important to confront the visibility problem. Specifically, before applying any shading, it is necessary to determine the visible geometry and other defined entities and objects inside the scene from any given viewpoint. To combat this problem, two approaches are used predominantly, rasterization and ray tracing. The first is nowadays associated with real time rendering and the second is known for its computational demands and more importantly, the superior quality of its results.

Rasterization

Rasterization is the process of generating pixel-based samples, on the geometry primitives (typically triangles). These samples are then separately shaded and potentially sorted according to distance from the view point to form the final digital image. The shading itself can be an arbitrarily complex process. Geometry must first be expressed in the proper coordinate system for any required operation to take place. As shown in Figure 2.10, a series of transformations are applied to express the coordinates from the objects' local reference frame, to a common (global or world) coordinate system and then via projection, coordinate normalization and conversion to pixel units, an image-space coordinate system (screen coordinates).

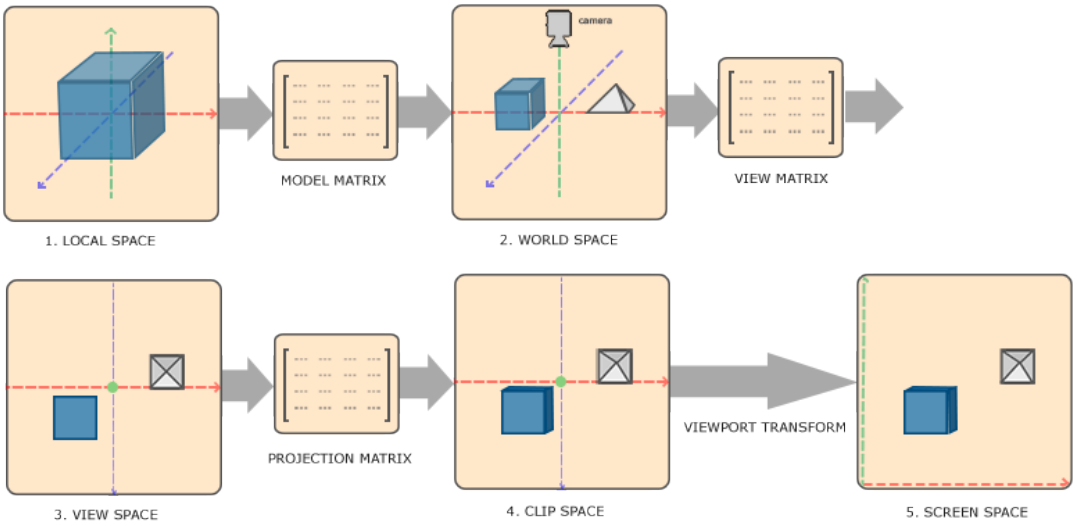

FIGURE 2.10: In order to transform geometry from its local space to the screen coordinates several matrix multiplications must be applied on its vertices. For more information see the source of this image. (Sources: [Vri16c])

The rasterizer samples primitives (triangles) testing which pixels in the bounds of the screen-space triangle are actually covered by it. Vertices typically carry information such as position, texture coordinate, normal vector, etc. For each fragment, the rasterizer performs value interpolation using the primitive's barycentric coordinates and data stored inside its vertices. Typically, after projecting geometry to the NDC, a pixel-based depth sorting is performed to achieve hidden surface elimination (general overview illustrated in Figure 2.11).

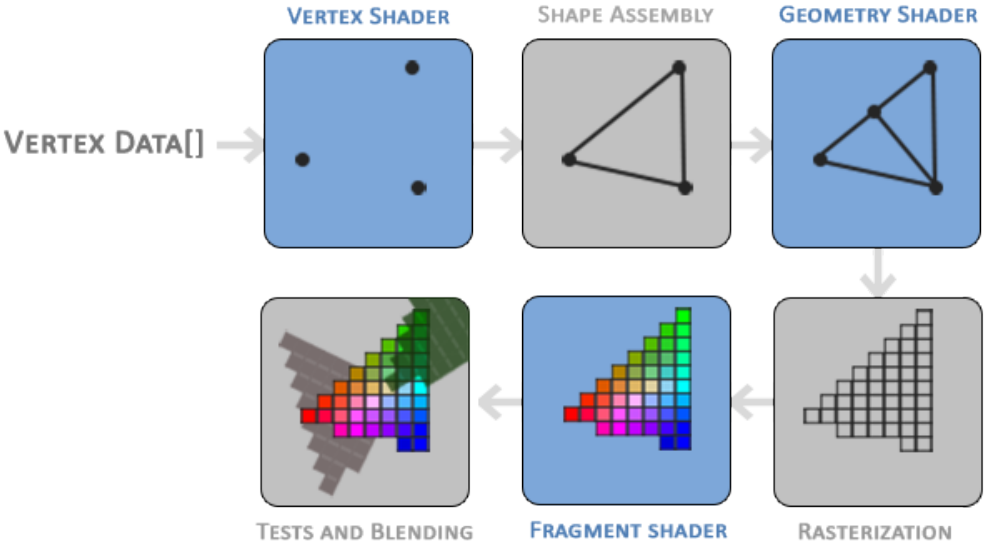

FIGURE 2.11: Blue sections depict the programmable stages during which we are able to generate extra geometry and perform fragment shading. Gray sections are non-programmable stages that perform the rasterization, blending, etc. For more details see the source of this image (Sources: [Vri16g])

As a final note, rasterization utilizes many optimizations, including back-face and fustum[1] culling. The rasterizer iterates all shapes assembled by the input data regardless to how those may be positioned inside the scene. Therefore, at least the early stages of the rasterization pipeline (geometric transformations and other vertex and primitive processing, clipping) are performed on all input geometry, without exceptions. Even invisible (to the camera) geometry goes through this process, which results in wasteful operations. To reduce this pointless additional workload we perform the aformentioned culling techniques (see Figure 2.12).

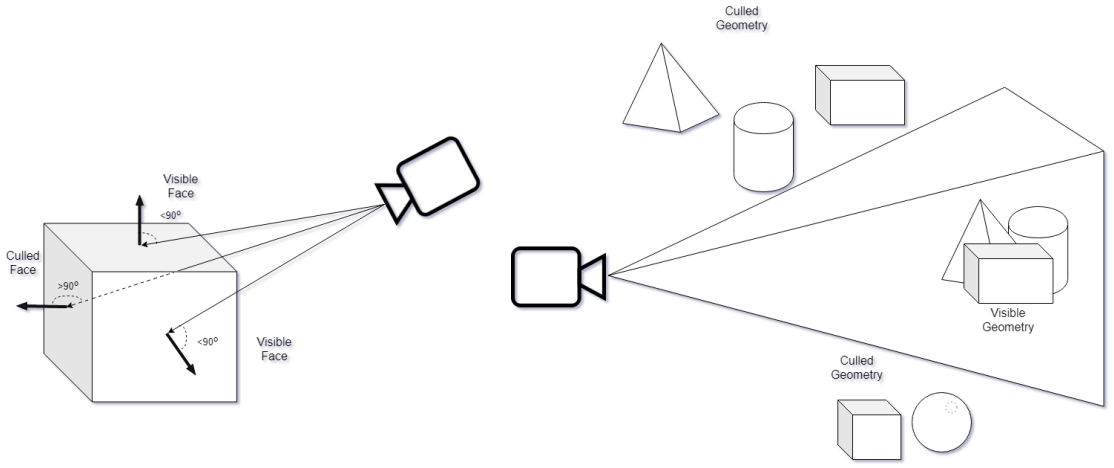

FIGURE 2.12: Rasterization-based optimizations. Left: Back-face culling. If the angle between the surface normal and the viewing vector is higher than 90° then this surface is invisible to the camera and therefore omitted from rasterization. Right: Frustum culling. Geometry that is outside the camera's viewing frustum is omitted from rasterization.

Compared to ray tracing, rasterization performs extremely well in terms of its online rendering capabilities. Additionally, throughout the years, rasterization-based techniques have gotten to an decent level of photorealism, an indisputable fact given examples of application such as the ones shown in Figure 2.13. Those are usually employed to visualize video game worlds and other virtual environments that need to provide a solid interactive experience.

FIGURE 2.13: Rasterization-based results. Left: Video game, Assassin's Creed: Odyssey. Right: Video game, Shadow of the Tomb Raider. (Sources: [Bra19])

[1] Space of the scene that is visible to the camera

These results are achieved primarily through the use of multiple rendering passes, each one generating geometry, material and lighting samples (potentially from multiple views) that are accessed from GPU memory in subsequent passes (see Figure 2.14). Since in the rasterization paradigm each token of information is processed independently and in isolation, such cascaded spawn-gather operations are necessary to allow point shading computations to have access to non-local information. An example of such a task is the ability to detect whether a shaded point is is shadow or not. This query requires the knowledge of interception of light by other parts of the geometry, which become available through a sampling process in preceding steps (see shadow mapping [Wil78]).

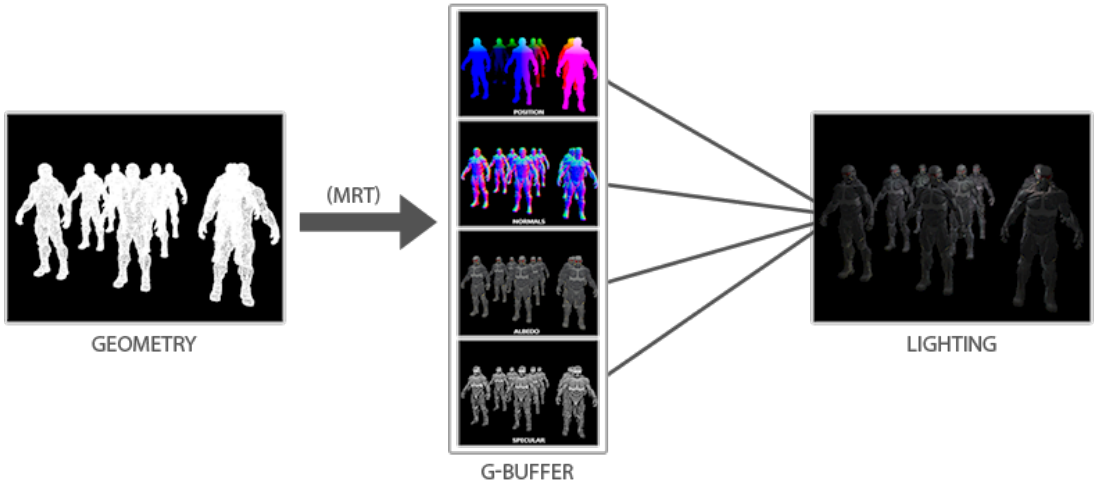

FIGURE 2.14: Example of multiple passes during Deferred shading. For more information see the source of this image. (Sources: [Vri16d])

Ray Tracing

Ray tracing refers to any technique that samples light paths (rays) inside a 3D scene. One may justifiably presume that such method would be implemented by generating rays that originate from light sources inside the scene and calculating shading that may occur throughout their journey to reach the designated camera. While this is not unreasonable, it is definitely a naive approach. This is because, although an infinitely large sum of rays may depart from a light source, only a minuscule number of them arrives at the camera's viewport, or in photography terms, the film plane[2]. Therefore, as illustrated in Figure 2.15, most ray tracing techniques trace light rays backwards, starting from the camera.

[2]Film/Image/Sensor (depending on the equipment) plane is a flat surface onto which light that passed through the camera lens is focused

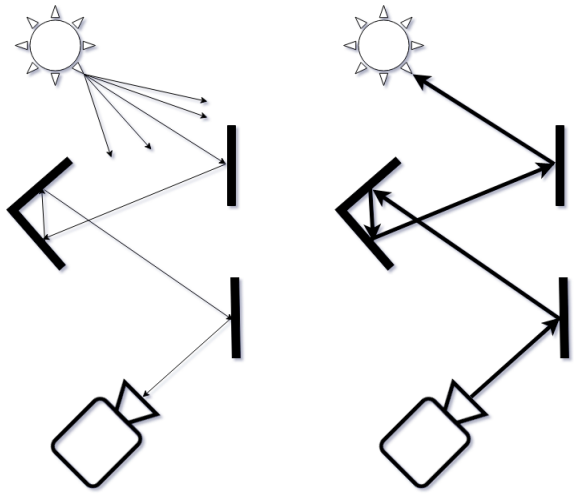

FIGURE 2.15: Ray generation approaches. Left: Forwards tracing. Right: Backwards tracing.

In typical ray tracing implementations, rays are cast from the camera's position and cross every pixel of the viewport, as illustrated in Figure 2.16. When rays intersect an object, they are either absorbed or reflected, spawning additional rays. Some perform direct lighting tests, to examine whether a surface is lit or not. In case a ray misses all geometry present in the scene, the background color is returned. Those algorithms are usually recursive and terminate either at a fixed depth or based on other conditions, e.g. ray contribution.

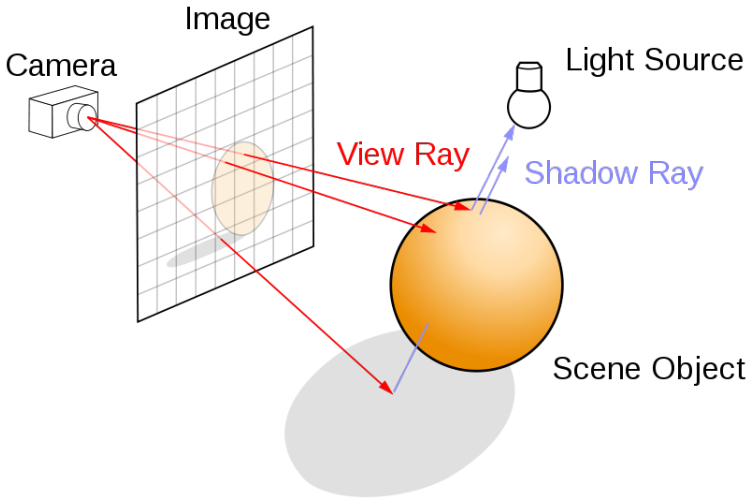

FIGURE 2.16: Ray tracing. (Sources: [Hen08])

A ray is a data structure that is defined by several attributes. First of all, it is constructed given an origin, i.e. a starting point generally described in world coordinates, and a direction. Using its initial position and direction vector, a ray can be cast from any place inside the virtual scene, including the camera location and any intermediate intersected surface. Apart from vectors needed to define its path, a ray may carry information such as attenuation and recursion depth which are essential for degradation effects and

termination determination. Furthermore, it may cache distance to nearest hit points and sort any intersected geometry in its path, enabling implementation of transparent geometry techniques. As a final note, it is usually required to store material, and other accumulated local attributes (e.g. surface normal), as well as any other payload that may be considered necessary for the shading calculations of the ray tracer.

Naive approaches of ray-primitive intersections would exhaustively examine a ray against all primitives inside a scene. Intersections are typically the most computationally intensive part of a ray tracer. Consequently, we need to prioritize reduction of their workload. To achieve this, many acceleration concepts can be applied. One example is using early termination and adaptive sampling to reduce total number of intersections needed to produce our final result. Another important one that we will discuss further in this section, is using spatial acceleration data structures known as bounding volume hierarchies (BVH).

A bounding volume is a space into which primitives can be grouped and placed, often chosen with certain helpful properties, such as being convex. A variety of bounding volume types are utilized, each optimized for different purposes. A BVH (see Figure 2.17) is a tree structure that encases all geometric objects of a scene. Those are placed inside bounding volumes stored as leaf nodes of this tree. Their parents and any ancestors are also bounding volumes up until the root node which is the largest volume that contains everything (scene). This way, geometry is organized into several "bins". Please note that this hierarchical organization can be defined at a more local level, enclosing separate objects, instead of the entire scene (object-level BVHs).

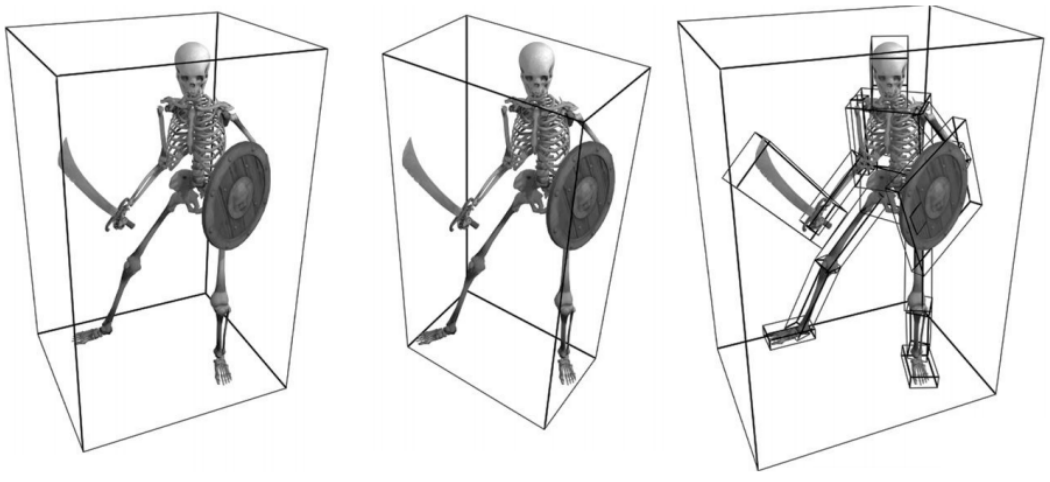

FIGURE 2.17: Bounding volumes and a BVH. Left: Axes-aligned bounding box. Middle: Oriented bounding box. Right: Bound volume hierarchy. (Sources: [Pap18a])

To test for intersections, the renderer checks which of those smallest "bins" are in a ray's path and only performs ray-primitive intersection inside them, as illustrated in Figure 2.18. This is an important tool for traversal and intersection query optimization that

is not only used to improve performance in rendering, but also applied on several other applications, e.g. collision detection. Since this is only a simple overview of bounding volume hierarchies we encourage interested readers to study a more thorough explanation of those concepts in [Eri04], chapters 4-6.

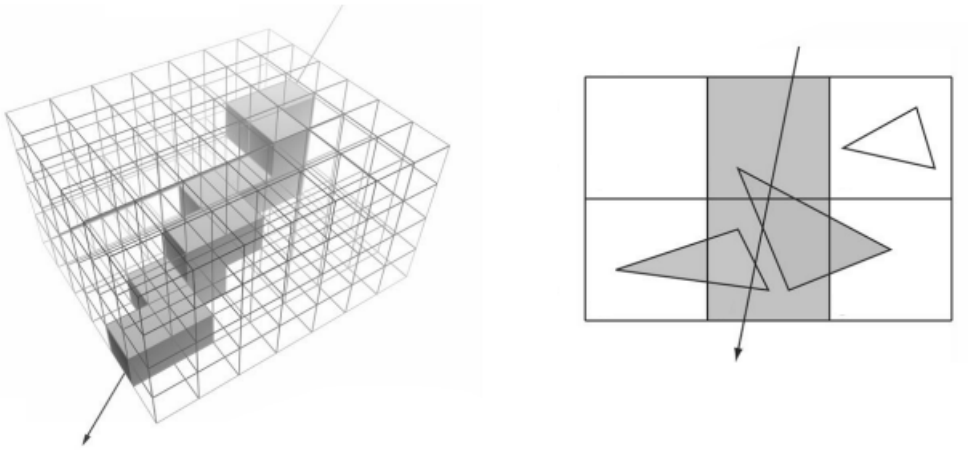

FIGURE 2.18: We initially perform ray-bounding volume intersection tests until we reach to a deep level within the hierarchy where ray-primitive intersection tests are performed. Therefore, we only check a subset of the scene's geometry. (Sources: [Pap18a](Adapted))

Ray tracing techniques are commonly much slower than rasterization-based ones. Generally they require many recursions, or if implemented in an iterative way, many repetitions until they converge to a photorealistic result. Alas, performance requirements of traditional implementations of ray tracers that utilize PBR are prohibitive for online rendering. However, they produce results that are, if not indistinguishable, at least very good approximations of real photographs (see Figure 2.19).

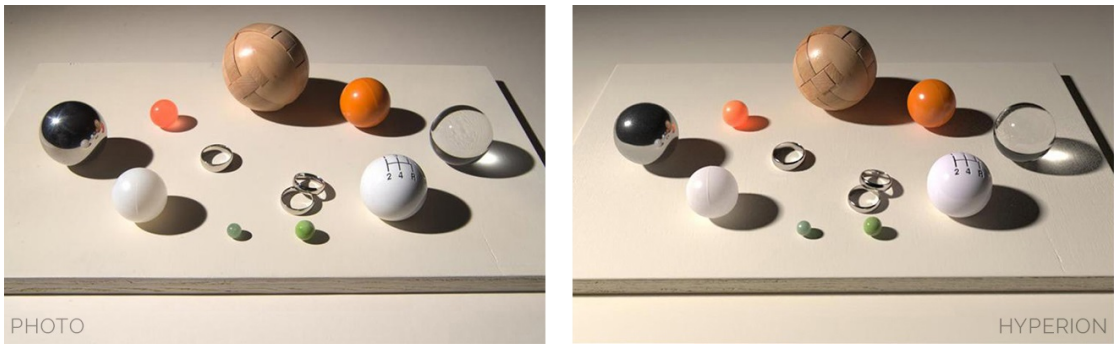

FIGURE 2.19: Ray tracing rendering capabilities. Left: Real photograph. Right: Hyperion's rendering. (Sources: Image: [DS13b] Related Article: [ENSB13]

Hybrid Techniques

Recently, with the release of Turing architecture and RTX technology [Cor18b] NVIDIA popularized hybrid rendering for improving photorealism in rasterization-based rendering. Faster ray tracing and denoising capabilities enabled those techniques to be used

in consumer applications, specifically, video games. This model uses rasterization for most rendering and ray tracing for specific graphical effects. Beforehand, those were implemented by "faking" light transport, using mostly screen-space[3] information and cube maps[4] with pre-filtered information. With the new model, techniques such as reflections, ambient occlusion, shadows and a substantial part of global illumination in general, become more photorealistic without an extreme impact on real-time interactivity (see Figure 2.20).

FIGURE 2.20: Comparisons of rasterization without/with ray traced effects. Left: Video game, Battlefield 5. Right: Video game, Metro Exodus. (Sources: [Bra19])

2.2 Stochastic Path Tracing

Popular ray tracing techniques such as Whitted-style [Whi80], fail to simulate distribution effects, i.e. integrals that take part in the accumulation of energy through a pixel for a specific duration. On the other hand, stochastic path tracing, by using several mathematical and statistical techniques that we will discuss in this section, is able to properly estimate those integrals and the resulting, non-surrealistically crisp images. Stochastic path tracing explores the path space by constructing paths in a non-deterministic manner, based on pdfs that govern the materials (BSDFs), the camera model (defocus blurring), time (motion blurring), path length, volumetric scattering and image plane coverage.

This is one of the most accurate and unbiased graphics techniques that is currently utilized for many applications. Animated movies, 3D modelers, design and illustration software typically take advantage of the stunningly photorealistic results of this method. Unfortunately, it is almost exclusively applied in a non real-time context, owing to the fact that it requires many rays, or in its case paths, to be traced otherwise it produces noisy and inadequate results (see Figure 2.21).

[3] Uses data limited to the visible segment of the scene
[4] Uses data from the six faces(textures) of a cube , see [Mos17], subject of "Environment Mapping"

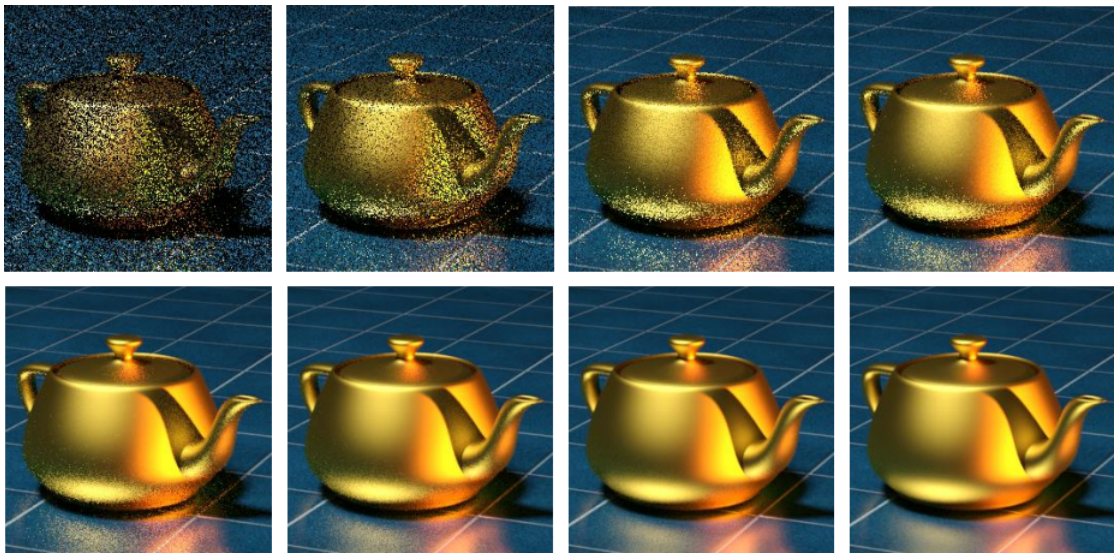

FIGURE 2.21: Different stages of convergence. (Sources: [Pap18b] (Adapted))

2.2.1 The Rendering Equation

The rendering equation proposed by Kajiya in 1986 [Kaj86] describes the energy balance due to incident and emitted light at the interface of two materials. Abiding by the law of conservation of energy, during events that occur from said light-matter interactions, this formula is able to approximate the intricate nature of these events. Specifically, it is an integral equation that is extended and varied to many forms, to include different and specialized phenomena. A generalization of most of the scattering events can be described by Equation 2.9.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L(x, \omega_i) f_s(x, \omega_o, \omega_i) d\sigma\perp(\omega_i) \tag{2.9}$$

where:

x	$=$ *position in space*
ω_o	$=$ *outgoing direction of light*
$L_o(x, \omega_o)$	$=$ *total outgoing radiance along the direction* ω_o *from a point* x
$L_e(x, \omega_o)$	$=$ *emitted radiance from a point* x
Ω	$=$ *the unit sphere (or hemisphere if only reflections)*
ω_i	$=$ *incident direction of light*
$f_s(x, \omega_o, \omega_i)$	$=$ *BSDF (or BRDF(*f_r*) if only reflections, see Equation 2.3)*
$d\sigma\perp(\omega_i)$	$=$ $\lvert cos\theta_i \rvert d\sigma(\omega)$*, "projected" solid angle on the surface*

2.2.2 Monte Carlo Integration

For practical applications, exact computation of high dimensional integrals through computer CPU or GPU code execution is very complicated. Fortunately, using a mathematical method known as "Monte Carlo Integration" [Sob94], we can efficiently approximate any integral function, no matter how complex it is. In computer graphics, any required form of the rendering equation is typically estimated using the Monte Carlo integration. In fact, the path tracing rendering technique typically refers to this approach of implementation.

According to the Monte Carlo integration, to approximate a definite, n-dimensional integral, random samples are chosen over the integration domain, and the integral is calculated based on the contribution of each of those samples weighted by their probability of selection. As a result, integrals of any function f(x) in a domain Ω can be approximated using random N samples weighted by their probability p as depicted in Equation 2.10.

$$I = \int_{\Omega} f(x)dx \Rightarrow < I >= \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{2.10}$$

To optimally apply Monte Carlo integration in path tracing, we use the tool known as "Importance Sampling" [BB88]. In importance sampling, we generate samples according to a pdf that is proportional to the integrand. Overall, choosing a proper pdf biases the sample selection towards more significant directions, thus reducing variance. On the other hand, if the pdf is not properly chosen then we will have an increase in variance, which translates to slower convergence rates.

That being said, by applying the Monte Carlo integration and importance sampling the rendering equation (Eq 2.10), can be now approximated by Equation 2.11.

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \frac{1}{N}\sum_{i=1}^{N}\frac{L(x,\omega_i)f_s(x,\omega_i,\omega_o)|cos\theta_i|}{p(\omega_i)} \tag{2.11}$$

where:

θ_i = *light's angle of incidence*
$p(\omega_i)$ = *probability of light's incidence direction*

2.2.3 Sampling and Ray Spawning

Direct Light Sampling

From the rendering equation, it is clear that the only source of illumination inside a scene is the set of emissive surfaces (term $L_e(x, \omega_o)$), which are typically described by specific material properties of surfaces. In path tracing, using Monte Carlo integration and importance sampling, non-uniformly-generated random paths are traced to contribute to the convergence of the currently rendered segment of a virtual scene. Given enough samples, the total illumination of a scene can be estimated to produce the desired photorealistic result. However, this non-direct and "brute-force"[5] approach can be very slow to converge, as many of the randomly chosen paths usually do not reach light emitters and end up contributing nothing.

To overcome this issue, direct light sampling needs to be performed. In this technique, light emitters are defined as different entities than the ones describing geometrical objects. Knowing their exact position and properties enables us to directly sample them at each rendering equation evaluation step, which in turn optimizes the implementation's performance and the algorithm's convergence rate. The integral form of the rendering equation becomes Equation 2.12. Additionally, in the L_i of $L_{Indirect}$, L_e from the points hit by the rays in -ω_i direction is not taken into account, otherwise the equation would sample light sources twice. Monte Carlo sampling is also utilized to integrate over the total surface area of a light inside the scene (Eq 2.13). This approach is generalized for multiple concurrent lights even when sampling a single path at a time (Eq 2.14).

$$L_o(x, \omega_o) = L_{Direct} + L_{Indirect} = \int_{A_L} L_e(y, \omega_i) f_s(x, \omega_i, \omega_o) G(x, y) dA(y) + \int_{\Omega} L_i(x, \omega_i) f_s(x, \omega_i, \omega_o) d\sigma\perp(\omega_i) \tag{2.12}$$

where:

A_L = *the solid angle subtended by the contributing light source*
$G(x,y)$ = *geometry term*
$dA(y)$ = *area of light's geometry at point y*

$$< L_{Direct}(x, \omega_o) >= \frac{1}{N_L} \sum_{i=1}^{N_L} \frac{L_e(y_i, \omega_i) f_s(x, \omega_i, \omega_o) V(x, y_i) G(x, y_i)}{p(y_i)} \tag{2.13}$$

[5] Exhaustively checks all possible cases

where:

N_L = *samples over the light's surface*
y_i = *point on the light's surface*
$V(x, y_i)$ = *visibility check, 1 ⇒ light source fully visible to x*
$G(x,y_i)$ = *geometric coupling term*
$p(y_i)$ = *probability of a sample*

$$< L_{Direct}(x, \omega_o) >= \frac{1}{N_L} \sum_{i=1}^{N_L} \frac{L_e(y_i, \omega_i) f_s(x, \omega_i, \omega_o) V(x, y_i) G(x, y_i)}{p(k_i) p(y_i|k_i)} \tag{2.14}$$

where:

k_i = *a light source*
$p(k_i)$ = *probability of selection of a light source* k_i
$p(y_i|k_i)$ = *probability of a surface point* y_i, *on a light source* k_i

Path Termination

Path termination is an important part of every ray tracing algorithm. In the case of path tracing, generated paths could potentially be extremely, or even infinitely, deep (recursion/iteration). This results in a never ending execution of the program's evaluation of the rendering equation. Enforcing a fixed length, e.g. by cutting off paths after a specific amount of evaluations, introduces bias to the generated imagery as important light transport operations may be ignored. Unfortunately, this is unavoidable when certain performance demands must be met.

Russian roulette (RR) termination addresses this problem whilst managing to produce unbiased results. It succeeds so by applying a hit/miss strategy that avoids the probabilistic bias. Using a reasonable probability p to cast a ray, and generating a uniform random number ξ, if $\xi < p$, the ray is then cast and weighted by $\frac{1}{p}$, otherwise the path is terminated. Value p is often related to either material properties of currently intersecting surface or path characteristics such as depth and total contribution.

Sampling the BSDF

In a single path implementation of path tracing, at each intersection point, apart from direct lighting and path termination determination, another major decision the algorithm is responsible of making is whether to transmit or reflect the next path segment. Material attributes such as roughness, metalness, etc. typically dictate this next path direction. According to those same attributes we may also choose different directional sampling distributions from which our algorithm generates this next path segment.

2.2.4 Denoising: Achieving Real-time Photorealistic Results

Undoubtedly, for most light transport phenomena, path tracing is among the best techniques to obtain photorealistic results. Unfortunately, due to current computational limitations, its is mainly utilized in offline rendering. However, this does not mean that it cannot achieve real-time performance if it is approached in a different, unconventional way. In fact, recent studies and research shows that through machine learning, it is possible to construct adequately clear results from noisy images. NVIDIA [LMH+18] trained a neural network to remove noise using only noisy inputs during the training procedure. Awe-inspiring results shown in Figure 2.22 depict that even in the case of under-sampled outcomes of path tracing, such a technique could be used for denoising. From [SSK+17], we see that by using a specialized spatio-temporal, variance-guided filtering technique, it is possible to achieve real-time path tracing with photorealistic results using a single sample per pixel.

Figure 2.22: Denoising. (Sources: Top: [LMH+18], Bottom: [SSK+17])

2.3 HMD-based Virtual Reality

As already noted, immersive VR experiences require convincing graphics and complex scenes rendered stereoscopically at high refresh rates. Many recent optimizations and improvements both focused at the computing and tracking capabilities of VR systems and their software-side implementation have enabled the production of several VR video games and media. Many popular game engines have integrated HMD-based VR in their

own rendering pipeline allowing game developers to experiment with stereo rendering and tracking-based orientation and movement. As a result, digital game distribution platforms were enriched with even more software that utilizes available HMDs on the market.

2.3.1 VR Rendering Optimization Methods

Reprojection

As noted in Section 1.2.4, naive approaches of stereo rendering effectively double the amount of operations the renderer has to perform. This directly affects graphics quality and permitted complexity of the VR scene. Reprojection is based on the fact that much of what is seen from one eye may have a lot of similarities to information observed from the other eye. Typically, in the context of computer graphics, during reprojection, the left and right eye views are rendered sequentially with information projected from the first, avoiding shading calculations in compatible pixels (see Figure 2.23).

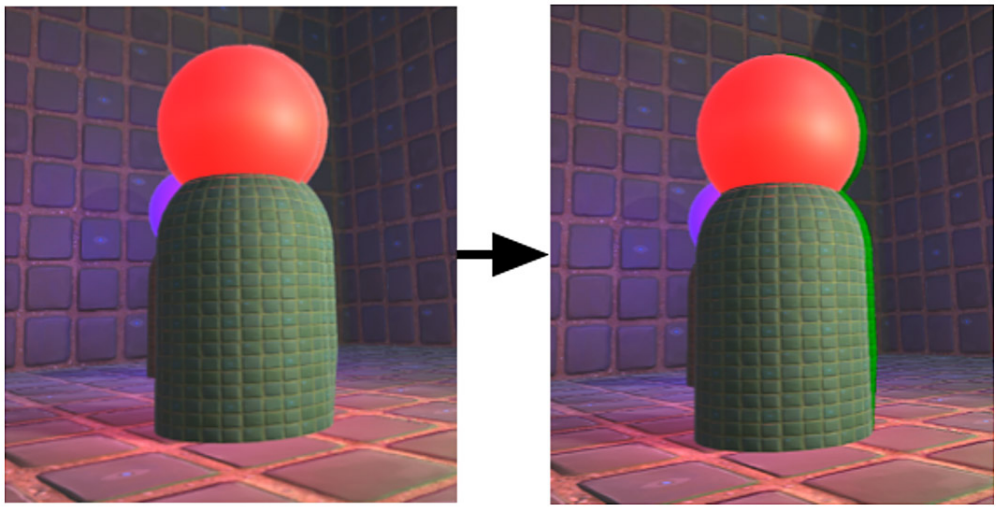

FIGURE 2.23: Reprojection. Incompatible pixels (marked green) of the second eye are shaded normally. (Sources: [ZG17])

In [ZG17] it is discussed how reprojection is integrated into the Unity game engine. The basic procedure is described as follows. Initially, the depth (see Figure 2.24) and color buffers of the left eye are computed and stored. Next, only the depth buffer of the right eye is computed and stored. Then, a reprojection pass is applied during which, by using the depth values, the color of the left eye is projected to the right eye's position. Simultaneously a pixel culling mask is generated to point out pixels that are not compatible. Using this mask, the remaining, unmatched pixels of the right eye are shaded to complete its view.

FIGURE 2.24: Depth buffer. Depth buffer values (0 to 1), determine the depth of the pixel. Darker pixels typically represent geometry that is closer. (Image rendered using XEngine [PV13])

The way matching between compatible eye pixels is performed is by reconstructing the world-space position of the right eye's pixels using its depth buffer and its inverse view projection matrix[6]. Next, using the left eye's view projection matrix, the world space coordinates are expressed into its own screen space coordinates. This way, correspondence between compatible pixels and reprojection of color information of the left eye to the proper pixel of the right eye is achieved (see Figure 2.25).

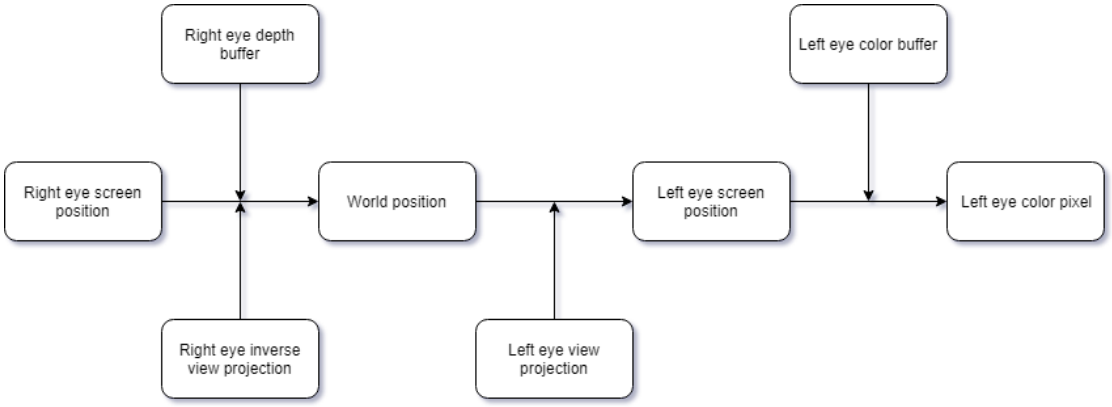

FIGURE 2.25: Reprojection matrix chain (Sources: [ZG17] (Adapted))

As we have discussed in Section 1.2.4, each eye perceives the world from a different point of view. This is reason behind incompatibly of some of the pixels of the eyes during reprojection. A major issue that this technique faces occurs when looking at effects that are highly view-dependant. For example, projecting information that originates from specular or glossy reflections is not necessarily correct since each eye should perceive a slightly different mirroring of the surface's surrounding environment. The solution that

[6]Using the inverse view projection matrix, we reconstruct the world-space from the screen-space coordinates (see Figure 2.10 reversed order)

[ZG17] suggests for this problem, is to provide the ability to disable reprojection on a per-material basis.

Lens-matched Shading

Recently, Oculus SDK added support for NVIDIA's "VRWorks Lens Matched Shading" [Hug18]. This is a concept that aims to compensate for efficiency losses due to lens distortion canceling applied on rectilinear images that typical VR applications supply HMDs with. Specifically, most VR rendering pipelines, including those that utilize the Oculus SDK, apply a barrel distortion to the rendered frames during post-processing, to cancel out the aforementioned lens distortion effect (see Figure 2.26).

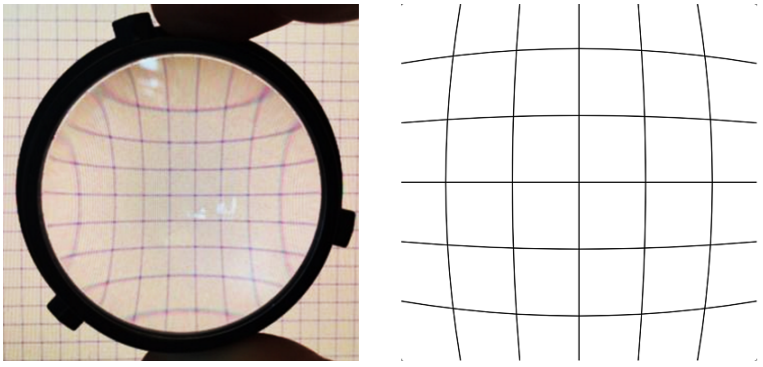

FIGURE 2.26: Lens and barrel distortions. Left: Lens distortion (pincushion distortion). Right: Barrel distortion. (Sources: Left: [Liu17])

Distortion parameters depend on the relative positions of the lenses of the HMDs and that of the user's eyes. However in general what occurs is that the central region of the image is magnified whereas the information found in the periphery is squashed. Figure 2.27 shows the effect barrel distortion has on a rectilinear image. Moreover, viewers often focus at the center of the screen. Consequently, we should avoid spending an equal amount of time and effort to calculate those two areas.

FIGURE 2.27: Effects of lens distortion on a rectilinear image. (Sources: [Liu17])

To combat this issue, the Oculus API permits renderers to upload octilinear images. As shown in Figure 2.28, those images are described by four different tilted sub-viewports that aim to match the resolution gradient of the lens. This way, more pixels are utilized for information that belongs to the central region instead of the periphery. Additionally, this approach also avoids rendering pixels that would be discarded from the HMD (see black surrounding spots in Figures 2.27, 2.28).

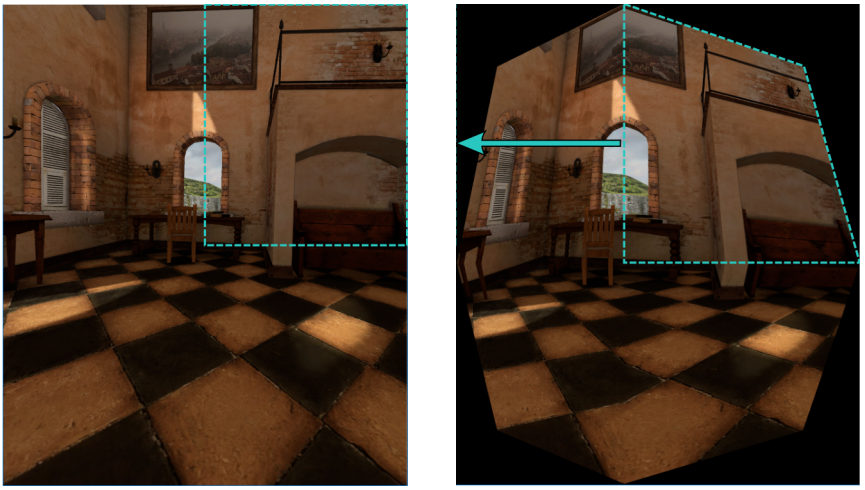

FIGURE 2.28: Creating an octilinear image from a rectilinear one. (Sources: [Hug18] (Adapted))

Stereo Instancing

Stereo instancing is a method utilized in rasterization pipelines to optimize VR rendering, especially in cases where a multitude of draw calls are issued. A draw call is a self-contained and indivisible command batch to pass a stream of data (primitives) through the graphics pipeline. If several individual parts comprise the scene's geometry with different material properties, geometric transformations or generally a different "state", then multiple draw calls must be issued as a state cannot change during a draw call. Other methods of optimization include the ones described in Section 2.1.3.

Rather than rendering the scene twice (rasterization and shading), stereo instancing prepares both left and right eye textures simultaneously using a single draw call for each object inside a scene. In the case of the Unity game engine [Sri17], eyes split the workload reducing graphics state changes. The GPU iterates over the geometric objects once and issues a single draw call to ultimately present them in both eyes.

Typically, in computer graphics, geometric instancing refers to reusing information of a 3D mesh under different transforms to speed up preparation and rendering time, as illustrated in Figure 2.29. Therefore, stereo instancing could be viewed as reusing the scene geometry adjusted at different offsets to match each eye's perspective.

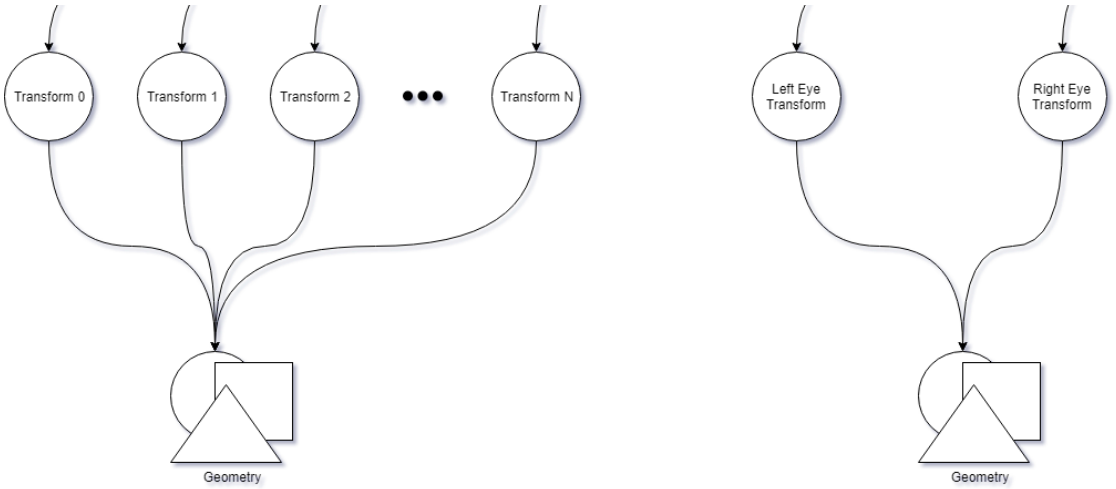

FIGURE 2.29: Scene-graphs utilizing geometric instancing. Left: Representation of typical geometry instancing. Right: Representation of stereo instancing.

2.3.2 VR Latency Correction Techniques

Asynchronous Timewarp

Screen tearing is a visual artifact that occurs when a display device's refresh rate is not matched with the framerate input from a video feed. To combat this issue, vertical synchronization (v-sync) is used. V-sync is an option that prevents the GPU from constructing more frames than the monitor is capable of displaying. Specifically, the GPU is given a certain amount of time to produce each frame depending on the refresh rate of the monitor. If the GPU completes preparation of the frame before the v-sync deadline, the GPU operations are halted until the deadline is reached and the next frame creation stage begins. If the GPU misses the v-sync deadline, the next frame will be submitted for presentation at the v-sync deadline closest to the frame completion time. This has a substantial effect on the framerate. For example, even if the GPU barely fails to submit a frame, missing every other v-sync deadline, the frame rate is essentially halved.

Asynchronous timewarp (ATW) [Ant15] is an Oculus Rift proprietary technique that reduces latency by operating independently and asynchronously from the rendering loop. As shown in Figure 2.30, rendering time of left and right eye frames may take too long and miss the v-sync deadline (Frame2). In this case ATW has already generated a new image for submission using data from the earlier frame (Frame1). This image is warped to reflect the rotation of the user's head. As seen from Figure 2.31 (extreme example), ATW may construct an incorrect result. This is due to the fact that, ATW's only source of information is the last submitted frame. However, typically this is not very noticeable. ATW results in lower latency between the HMD orientation changes and the updates inside the virtual scene. This way, motion appears smoother to the user.

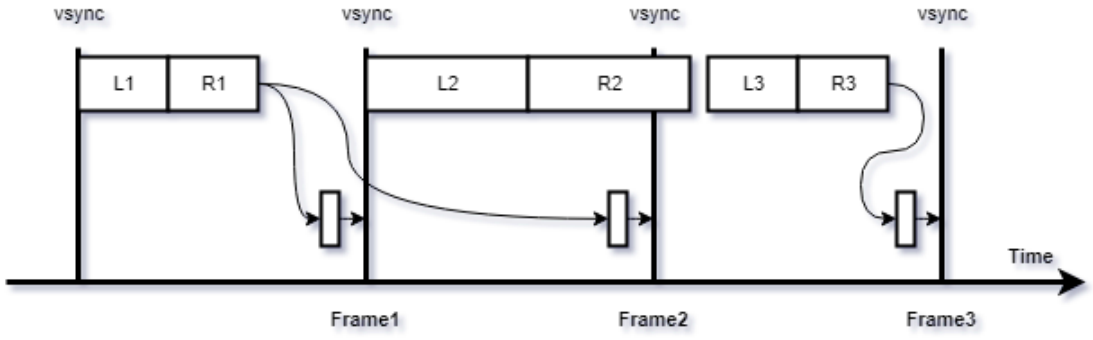

FIGURE 2.30: Asynchronous timewarp.

FIGURE 2.31: ATW image warping. (Sources: [eVR14])

Asynchronous Spacewarp

As mentioned in the previous paragraph, ATW is a mechanism that supplies the application with orientation-corrected intermediate frames when rendering of next frames takes longer than expected. Asynchronous spacewarp (ASW) [BHP16] is a concept similar to ATW. In fact, ASW also creates extrapolated frames from data found in previously composed frames. However, it goes a step further to smooth out the whole VR experience. This includes user's positional head movement and motion of visible objects. It succeeds so by performing animation detection and compensation with real-time GPU and CPU animation analysis. Figure 2.32 depicts the way ATW and ASW cope (autonomously) in various frame rate profiles.

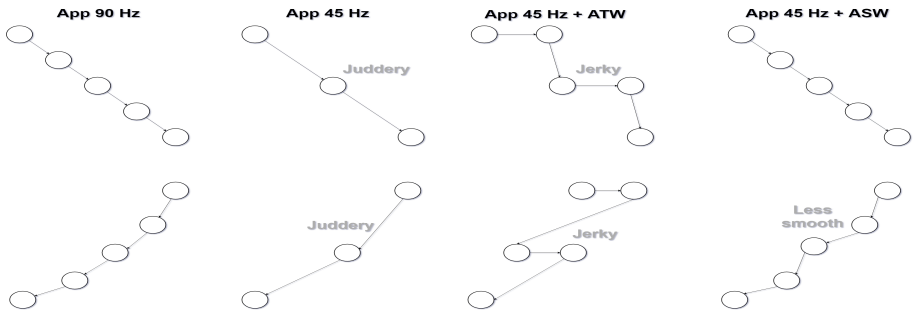

FIGURE 2.32: ATW and ASW in various frame rate profiles. Top Row: Linear animations during HMD rotation. Bottom Row: Non-Linear animations during HMD rotation. (Sources: Adapted from [Ped16])

Asynchronous Spacewarp 2.0

Oculus released an update to ASW which uses depth information to enhance the optimization capabilities of this technique. Contrary to ASW 1.0, this new version can perform adequately even at lower than half the frame rate target of the HMD. Given a depth buffer of the currently visible segment of a scene (see Figure 2.24), ASW 2.0 [AB19] performs object separation to solve reprojection errors that may occur. As shown in the following Figure 2.33, those errors can be very apparent when ASW 1.0 is employed.

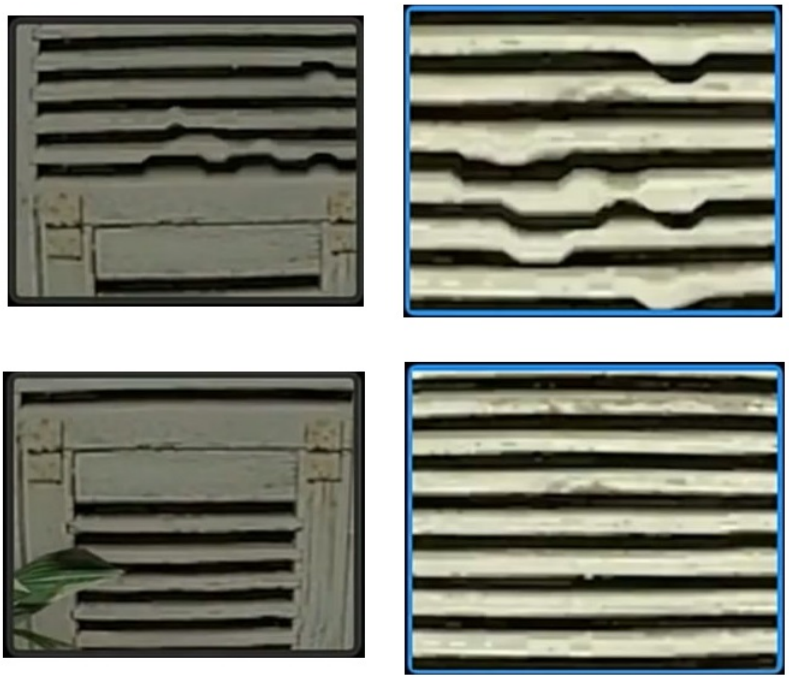

FIGURE 2.33: ASW version comparison. Top Row: ASW 1.0. Bottom Row: ASW 2.0. (Sources: [AB19])

Chapter 3

Implementation

From a technical standpoint, creating a platform able to display frames inside an HMD is not necessarily challenging. The straightforward nature of the Oculus SDK and the hardware and software abstractions it provides are more than helpful in our task. Real difficulties arise when our platform is required to support intricate graphics techniques, complex interactive worlds and elaborate input systems. Without those features, it would be impossible to construct an interactive VR environment.

To achieve an adequate, believable level of photorealism, it is important to have a rendering pipeline with well defined programmable and fixed stages. The reduction of the total workload of the render loop, the lessening of the scene primary and graphics memory consumption and the acceleration of load times are also essential for performance. Finally, handling of multiple hardware input devices such as game-pads and touch controllers generally require a more intricate management system, capable of performing synchronized propagation of their input events to the structural elements that comprise the scene.

In this thesis, we are given the opportunity to create a complete and extensive backend, used for experimentation and testing for research that may go beyond the scope of our work. On that account, a graphics engine was created from scratch using modern programming tools and techniques. For the purposes of this thesis, we use this graphics engine as a host for our VR path tracer. Moreover, this renderer acts as a platform to test several optimizations on mainly offline photorealistic rendering techniques and how well they cope in a stereoscopic rendering context. This work is part of a greater series of academic research on photorealistic VR that focuses on increasing photorealistic rendering performance via VR-based graphics optimizations.

3.1 Platform

3.1.1 RT-XEngine

Typically, proper hardware utilization for resources such as 3D models, textures and shaders needed to test complex graphics techniques, requires the use of a middle-ware commonly known as a game or graphics engine. This program generally manages assets, creates and updates scene-graphs[1], and hosts one or multiple rendering pipelines. In their most complex forms, they feature world and material editors, script-able behaviour of objects and entities and other additional functionality, which may server their purpose. The RT-XEngine, mentioned in the introductory section of this chapter, is a newly crafted graphics engine, developed in modern C++17. In it, developers can integrate any rendering API and specification, e.g. OpenGL, Vulkan, D3D, etc. and target any interoperable rendering surface. For the purposes of this thesis, this engine is primarily used to combine the above-mentioned OptiX framework and the Oculus SDK in an optimized fashion.

Asset Management

Asset management in RT-XEngine is performed for both CPU and GPU-side assets. Specifically, this engine employs multi-key hash maps to tie any resources to their unique properties. Those may include information such as system filepath, target dimensions and even other assets that may be part of them. Therefore, resources can be cached and reused, and that results in reduction of engine loading and preparation time. Moreover, the lifetime (loading/unloading) of those assets is handled automatically through reference counting leaving no room for memory leaks or erroneous accesses. Last but not least, asset typing is managed in a meta-programming[2] fashion, which dramatically reduces the complexity of their hierarchical structure and the amount of code required to request them.

Worlds & Renderers

A world in RT-XEngine is a container and accessor of scene nodes. In reality, worlds are nodes themselves and operate as roots of the scene-graphs that describe the hierarchical structure of said scene nodes. Multiple concurrent worlds may be active at a time and each of them may be targeted by one or more renderers. Those possess the ability to act as world observers and change state according to updates inside their target world.

Object oriented implementations of graphics engines typically render a scene by iterating its scene-graph. In RT-XEngine's case though, worlds provide several data structures

[1] Graphs that hierarchically describe the scene
[2] In our case, C++ templates are used for compile-time type resolution

that contain every geometry, light and camera pointer listed in them. Each renderer, based on its programming, will access any world information deemed necessary. For example, a renderer may request only initial camera properties and geometric information to render a static image. Others may operate as observers of user positional changes and transformations of entities inside the world to render dynamic and animated scenes.

It should be noted that having multiple worlds and renderers is mostly beneficial for real-time comparisons of results, in terms of quality and photorealism. Alas, performance is negatively affected, usually in a non-linear fashion when using a single graphics unit and multiple rendering APIs in the same process, as sometimes context switching is required to avoid race conditions.

Multi-Renderer & Front-end Window System

The RT-XEngine, at the present stage of development, is comprised of a back-end API[3] and a front-end window system, both developed in C++17. The front-end uses abstractions that the "Simple DirectMedia Layer" (SDL) version 2.0.9 [sdl01] library provides. It also includes several engine module abstractions, i.e. engine, world and renderer wrappers, and a render loop manager.

In a typical case example, the engine context is first initialized to act as a container to every engine sub-module, i.e. file-system, worlds and renderers. Worlds are then loaded using a simple XML scene description file. Next, one or more renderers may be created using parameters including world target and renderer type. Those are then added to the render loop manager, which in turn chooses the appropriate window object to use as their rendering surface target. Windows initiate the rendering context and command renderers bound to them to initialize their scene (using information found in their respective target world). Whenever worlds, renderers and windows are ready, the application begins its render loop during which, rendering calls and input state changes are issued to every active window. Those in turn may perform context switching and input information propagation to any engine modules associated with them.

3.1.2 NVidia OptiX Ray Tracing Engine

The NVidia OptiX [PBD+10] ray tracing engine was employed to simplify the construction process of our rendering algorithm, as well as to utilize the RTX technology for accelerated geometry intersection and AI-denoising (see [Mor18] for more details). OptiX is a framework that assists graphics engineers with the task of executing and parallelizing ray tracing techniques inside the GPU. It contains several functions to help alleviate math complexity behind intersections and shading operations. Additionally,

[3] Application programming interface

by abstracting GPU driver calls, OptiX reduces code boilerplate that may be required for GPU-accelerated calculations.

OptiX 6.0 is a synergistic framework that is divided into two main parts. The first is an API in which geometry and texture data, acceleration structures such as the aformentioned BVH (see Section 2.1.3), ray spawning entry points and intersection graph nodes are defined. The second one is a CUDA C++-based programming system, where ray intersection, additional ray spawning (apart from entry point), miss and exception behaviour is described. OptiX is not exclusively used for graphics application, as other simulation systems can benefit from ray-surface intersections, e.g. collision detection, sound propagation, etc.

Ray-Tracing

OptiX consists of host[4] and device[5] code. Typically, its host-side is executed in an early run-time stage during which several data structures are uploaded from disk or ram to the device memory. These structures act as nodes that assemble an interconnected graph. They define geometry, material, shader, acceleration, transform and group information and their hierarchy affects intersection order, data instancing and performance. The host is also in charge of creating an entry point which acts as the root of the aforementioned hierarchy graph and issues one or more "context launches", i.e. calls spawning the primary rays.

In a context launch, the GPU executes shader/program code that defines behaviour during intersections and additional ray spawning. To be more precise, after constructing the primary ray, a ray generation program issues a trace call which in turn begins the hierarchical graph node traversal. Using ray information, e.g. origin and direction, and the acceleration structures defined during construction of the graph, OptiX tests whether each ray hits or misses.

Regarding those events, there are programmable miss, closest hit and any hit shaders. As their name suggests, miss shaders are called whenever a ray misses. On the other hand, hit shaders are divided between closest hit and any hit programs. The latter are not limited to intersection queries that return the nearest primitive but instead they are called for any primitive intersection that occurs in the ray's path. In our case, this can be utilized by shadow rays to minimize their computational cost. This is due to the fact that we can halt the ray traversal after discovery of any intersection between the current hit point and the light source. Last but not least, there is an exception program directly associated with one or more entry points that is mainly used for debugging purposes.

[4]Executed in CPU
[5]Executed in GPU

Finally, as the example illustrates in Figure 3.1, OptiX graphs usually have a two-level acceleration data structure scheme. This means that, typically, there are group nodes that contain a collection of geometry sub-groups (3D models/objects) nodes. Both group nodes and their children are assigned different acceleration structures. If the children motion is rigid, then only the group's acceleration structure needs to be adjusted. This way we avoid reconstruction of every single BVH of the children nodes.

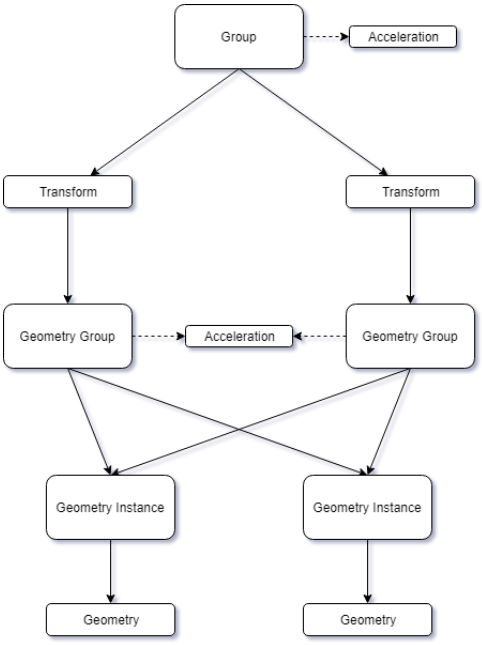

FIGURE 3.1: Example of an OptiX graph. In this example, geometry groups that share an underline geometry, also share an acceleration structure to maximize efficiency.

AI-Denoising

OptiX 6.0 also provides a post-processing framework and a few built-in post-processing stages. One of them is the deep-learning-based denoiser. This is a neural network that was trained with a large group of rendered images in various stages of convergence. This way, given a noisy image, e.g. an under-sampled path tracing result, the denoiser is able to estimate the final, converged result.

The implementation of this denoiser is based on the work of [CKS+17]. Although, as seen in Chapter 4, this denoiser manages to produce real-time, photorealistic results, it is not temporally stable, meaning that in a sequence of denoised results, flickering and other visual artifacts may become noticeable. This is due to the fact that this AI-based implementation denoises frames in isolation and does not retain a memory of its previous results. Moreover, as its default training set was not tuned for such purposes, it fails to denoise objects behind transparent materials and produces blurred results when attempting to denoise high frequency surfaces (e.g. hair).

3.1.3 Oculus VR Development Kit

Oculus SDK & VR Rendering

First of all, the Oculus SDK is responsible for handling distortion rendering, GPU synchronization, frame timing and presentation to the HMD. The scene must be rendered onto one or more textures. Furthermore, the SDK also gives access to tracking state and eye pose information, data that originates from hardware sensors placed inside and outside the HMD. Developers should accurately emulate user's motion that is recorded by the HMD, as well as any other input devices, e.g. touch controllers, inside the virtual world and update necessary entities and actors that describe objects affected by those actions. Finally, the SDK provides a mirror texture. As its name suggests, this texture mirrors Rift's contents on the application window. This is a useful tool for debugging purposes as it can display pre and post-distortion results, given the proper flag parameters. Overall, as detailed in the Oculus SDK samples [Hea14], the following Algorithm 1 describes the general approach of integrating Oculus VR rendering into a non-VR application.

Algorithm 1 Oculus SDK Integration

```
1: Declare the Oculus texture(s)
2: Initialize SDK and engine
3: Configure VR
4: Create mirror texture
5: Initialize scene, models and cameras
6: for main loop reading inputs do
7:     Update scene using inputs
8:     Get tracking information
9:     for each eye do
10:        Set eye render target
11:        Get matrices
12:        Render models
13:    end for
14:    Distort and present
15:    Render mirror texture on the application's window
16: end for
17: Release mirror texture
18: Release Oculus texture(s)
19: Release SDK
20: Release engine
```

DK2 Characteristics

For the visualization purposes of our work, we utilized the Oculus Rift DK2. This HMD features a 5.7 inches pentile amoled display with a resolution of 1920 x 1080 (960 x 1080 per eye) and a refresh rate of 75Hz. It also includes a gyroscope, an accelerometer and a magnetometer and is capable of providing orientation and position tracking with 6 degrees of freedom (see Figure 3.2).

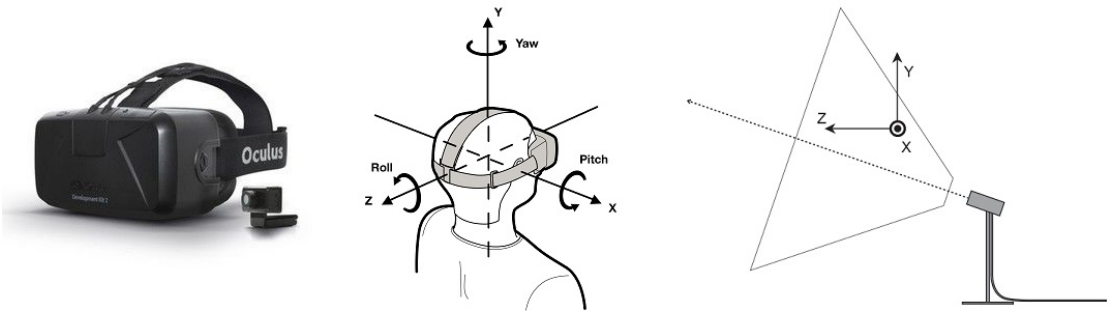

FIGURE 3.2: Oculus Rift DK2 tracking features. Left: Oculus Rift DK2 and tracker. Middle: DK2's orientation tracking. Right: DK2's trackers frustum. User must be inside otherwise positional tracking is negatively affected. (Sources: [VR16])

3.1.4 Hardware

Development, experimentation and testing were performed in a high-end personal computer comprised of the following hardware components: an "Asus Maximus VIII Ranger" (LGA1151) motherboard, an "Intel Core i7 6700K" @ 4.00GHz CPU, 32.0GB @ 2133MHz ram, an "Nvidia GeForce RTX 2080 Ti" graphics card and an "Oculus Rift Developers Kit (DK) 2".

3.2 Oculus SDK & OptiX Engine Interoperability

As already mentioned, the Oculus SDK is responsible for presenting eye frames to the Rift HMD. Overall, this is performed by passing one or more textures into the API. Those textures can be created from either a Direct3D 11 or 12, an OpenGL or a Vulkan rendering context. Specifically, Oculus SDK provides interoperability with those rendering APIs and specifications through use of an "ovrTextureSwapChain" object, which represents eye buffers, in an API-specific way. In reality, this object contains multiple buffers that are swapped between each successive frame presentation to the HMD, to avoid stuttering and tearing. To submit a frame that will be displayed by the HMD, the application needs to pass one or more texture swap chains within an "ovrLayerEyeFov" structure. This describes a layer that specifies a monoscopic or stereoscopic view. There are two options, as far as storing of textures swap chains inside the layer structure is

concerned. The first one, is more straightforward and uses two textures, one for each eye. The second one uses a single texture with two defined viewports, each referring to sub-rectangular regions inside the texture that correspond to each eye. After preparation of this layer, the application submits it for distortion rendering and display.

3.2.1 Oculus Platform Integration

The Oculus SDK is mainly integrated into RT-XEngine's front-end system. It is placed inside the window hierarchy as an inheritor of properties and methods of OpenGL-based rendering surfaces. This is justified as OptiX, which, as mentioned above, is the second component used for our specialized renderer, is able to interoperate with OpenGL. The structure responsible for maintaining and managing a VR session between the HMD and our application is named "GLOculusWindowSingleTexture". In it, operations are divided across specific functions to work as an abstraction mechanism for Oculus SDK's integration stages mentioned in Section 3.1.3.

Initialize

During this procedure, parameters are prepared and passed to the Oculus SDK initialization function. Those include flags that override default behaviour, SDK version requests, user-specified logging and the time in milliseconds that the application awaits for HMD connection. Next, an SDK function is called to create a handle to a VR session. Using this handle, HMD information such as its recommended and maximum FOV (field of view) for both eyes, its resolution and its display refresh rate becomes available to the application.

Next, the eye texture swap chain is constructed based on HMD description and a few adjustments. For simplification purposes, we use symmetric FOVs for the eyes, using the maximum values of their original up, down, left, and right tangents. The SDK calculates the recommended viewport size needed for rendering the eyes within the HMD given their FOV cone (described by said tangents) and the correspondence of pixels per display pixel. Higher FOV requires larger textures otherwise the result is negatively affected.

Finally, initialization of this window includes the creation and configuration of the aforementioned mirror texture. This texture is given user-specified width, height, format and a few initialization flags. Using those flags, we can display only one of the eye views, visualize post-distortion results, force symmetric FOVs and enable or disable mirroring of the Oculus system notifications and menus.

Handle Events

As part of the window hierarchy, during event handling, "GLOculusWindowSingleTexture" propagates input information to its corresponding renderer and world. However, contrary to other windows, it additionally conveys HMD tracking state as specialized input events to update the user's position and orientation inside the virtual world. Specifically, our application requests status information from the session handle. Using this data, it initially checks whether the HMD's tracking should be re-centered inside the tracking frustum. Session status also examines if the process has VR focus and thus is visible in the HMD. If all is set, the application designates that it is ready to begin rendering. Next, the predicted display time of next frame is queried from the API. The value obtained in return is used to calculate HMD's predicted pose. Using this data, head, left and right eye position and orientation is extracted and registered on specific update events that are later propagated through the engine's event system to the user node and its sub-nodes inside the scene.

Render

Before requesting from the back-end renderer (the VR path tracer in our case) to produce its results, the texture swap chain updates its internal GL textures that essentially hold information used by the HMD. After rendering on that texture, the texture swap chain commits any pending changes and advances its current texture indices. To display the final frame, our application needs to submit layers using a specific end frame SDK function call. Layers are drawn in order regardless of their type. In the implementation of GLOculusWindowSingleTexture, we use a single texture swap chain that contains color and depth information split vertically in the middle to be used by both eyes. In the end, after submission of the frame, the mirror texture is updated with Rift's result. The application copies this information to finally display it on a window.

3.2.2 Achieving Fastest Possible Interoperability

When created under an OpenGL context, Oculus texture swap chains alternate between two or more GL textures. OptiX supports interoperability for OpenGL and, to be more specific, for GL buffer objects. Minimizing transfer operations and benefiting from direct memory access (DMA) requires a "bridge" between the Oculus SDK and our OptiX-based renderer. Thankfully, OptiX buffers can be constructed using GL-generated PBOs. From [HA18], it is understood that by using those buffer objects, texture transfers are performed inside OpenGL controller memory (see Figure 3.3). Therefore, by applying those direct memory transfers to the Oculus textures used for HMD submission, we manage to completely isolate most of their preparation process inside the graphics module.

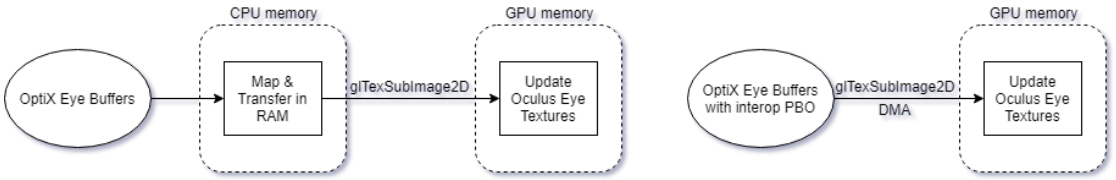

FIGURE 3.3: Direct memory access principles. Left: No DMA. Map buffers into main memory and then copy them into GPU memory. Right: DMA. Copy buffers inside GPU memory

3.3 Virtual Reality Path Tracer

The VR path tracer was developed to test a variation of real-time path tracing using VR-based optimizations. Current implementation consists of the renderer's scene construction, or more specifically, RT-XEngine's world parsing to generate OptiX graph nodes and extract any other information required for ray tracing and shading. It also includes a basic model of path tracing adjusted in a way that when we apply denoising on top of it we are able to achieve real-time performance. It should be noted, that the VR path tracer is part of a greater series of academic work on photorealistic virtual reality and its main objective is to examine how well such a rendering technology and paradigm behave in a VR context, as well as to point out any issues and concerns that may arise in the process.

3.3.1 Scene Building Stage

RT-XEngine's worlds are currently loaded from an XML scene description file that defines all appropriate camera, light, environment (sky), geometry, transform and grouping information. Those are denoted as nodes or meta nodes, i.e. structures comprised of multiple nodes and their inner relations, and are stored both sequentially and as a tree graph in their respective world container. During each iteration of the render loop, a world caches global transformation data to every node that has been invalidated due to some change, e.g. geometrically transformed. Renderers are solely responsible for defining how to parse and observe world information needed for their own scenes. Consequently any world, considering it contains the proper entities, may be used, or in better words, observed, by many renderers concurrently.

The renderer created for the purposes of this thesis attempts to combine a real-time version of path tracing and VR to operate as a stepping stone for future research on optimizations between those two concepts. At the time of writing, the nodes detailed in the next paragraphs are exposed to OptiX as scene graph nodes.

Triangle Mesh Geometry Node

The triangle mesh geometry node carries information needed to load vertex and material data of a 3D model, which is composed of triangles. It also provides a hint about whether geometry is going to be structurally unchanged (static) or dynamically updated. Yet again, each renderer is responsible to decide how to act on any information received as an input. In VR path tracer's case, each and every triangle mesh geometry node is loaded sequentially and their hierarchical status is ignored. Their cached world transformation data is used to place the geometry in a correct location inside the scene and according to the aformentioned hint the renderer decides whether to mark the top BVH structure for refiting or not. If it is set to refit, after the geometry has being invalidated, OptiX only re-adjusts the node bounds of the bounding volume hierarchy instead of constructing it from scratch, resulting in a very fast BVH update without sacrificing too much ray tracing performance. See Figure 3.4 below for corresponding OptiX sub-graph result.

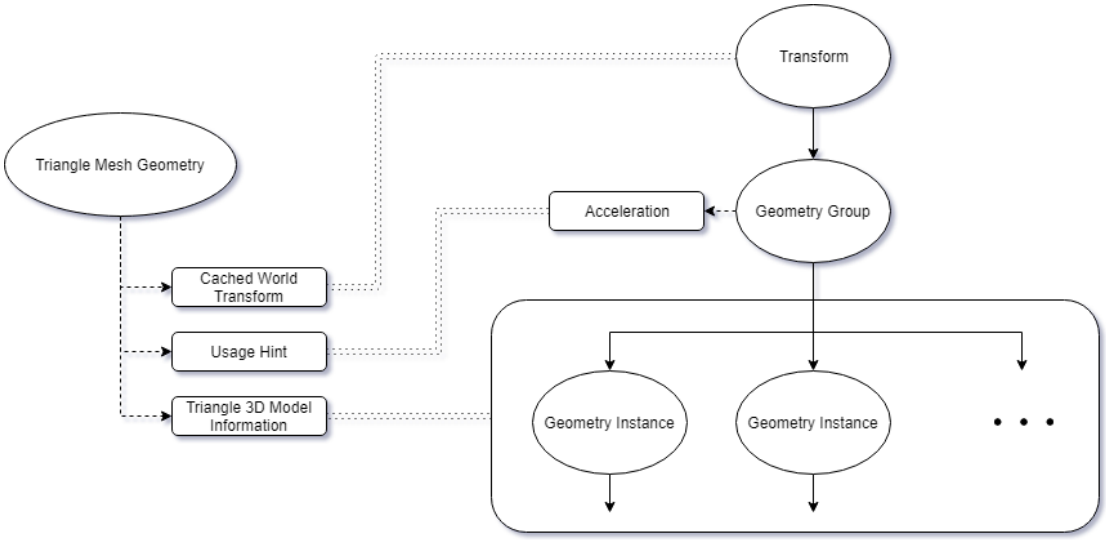

FIGURE 3.4: Triangle mesh geometry node to OptiX sub-graph. Note: Geometry instances are produced depending on the amount of materials found in the triangle 3D model information. In OptiX 6.0 with RTX enabled, triangle geometry instances can only support a single material.

Instanced Triangle Mesh Geometry Meta Nodes

Instanced triangle mesh geometry nodes contain instancing information about a certain 3D triangle model. Renderers that support geometry instancing, i.e. uploading of the geometric data of a 3D mesh once and reusing it under different transforms, can generally benefit from those nodes. The VR path tracer is one of them, as it utilizes OptiX geometric instancing capabilities to improve performance. See Figure 3.5 below for the corresponding OptiX sub-graph result.

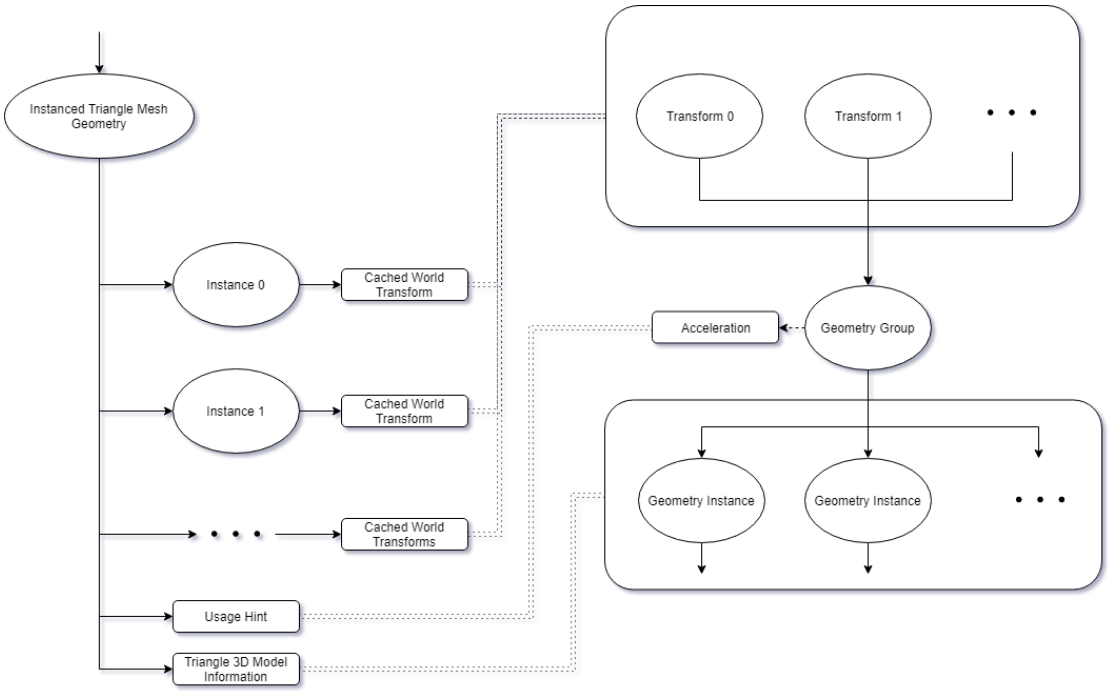

FIGURE 3.5: Instanced triangle mesh geometry meta node to OptiX sub-graph. Note: Geometry instances are produced depending on the amount of materials found in the triangle 3D model information. In OptiX 6.0 with RTX enabled, triangle geometry instances can only support a single material.

Sky Nodes

To represent the distant environment RT-XEngine's worlds may use any of the available sky nodes. The first one is the sky cube node which in practise is a 3-dimensional box of textures, also known as a cube map. During rendering, those textures are used to encase the camera inside them. The second one uses an HDR panorama image that warps around the camera spherically. Both nodes and their corresponding techniques create the illusion that the viewer is positioned inside an enormous sphere, in the same way a person perceives the real surrounding environment (Figure 3.6 portrays those methods).

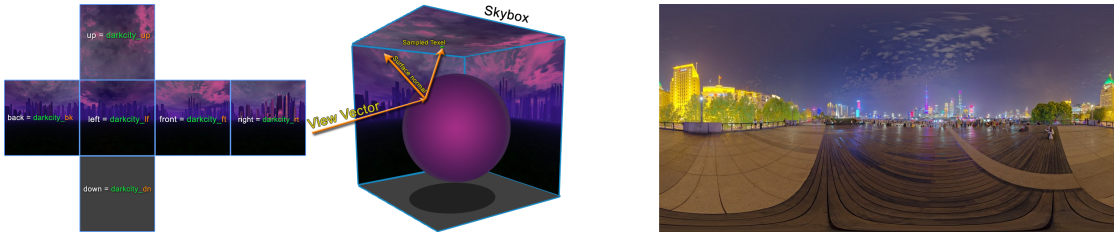

FIGURE 3.6: Environment information. Left: Cube map. Right: HDR panorama (Sources: Left (Skybox): [Low11] Right: [Zaa18])

The VR path tracer currently uses the second method. Specifically, for a ray direction **r** and using the set of Equations 3.1, the renderer can sample, using the u and v values, the HDR image during miss events. Therefore, paths that do not interact with any geometry

simply draw radiance from the environment which is depicted in the panorama. Surfaces from where there are direct paths that end up in miss events, depending on their material properties, may display specular or glossy reflections of the HDR image.

$$\begin{aligned} \theta &= \arctan 2(r_x, r_z), \\ \phi &= \frac{\pi}{2} - \arccos(r_y), \\ u &= \frac{\theta + \pi}{2\pi}, \\ v &= \frac{1 + \sin\phi}{2}. \end{aligned} \tag{3.1}$$

Light Node

Light nodes are responsible for defining entities used in the direct lighting model, as analyzed in Section 2.2.3. They describe lights using information such as position, direction, color and flux. The VR path tracer uploads them to the device shader part of the implementation, packed inside a buffer to support multiple lights. Currently, all light entities operate as punctual omni-directional lights (Figure 3.7) to reduce the amount of samples required for convergence. However, as it will be discussed in Chapter 4, we aim to enrich direct lighting in VR path tracer to incorporate more types of lights and a more sophisticated sampling model.

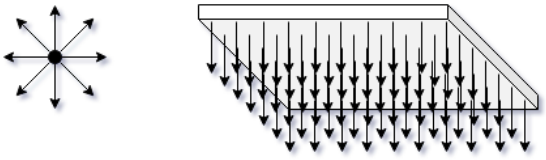

FIGURE 3.7: Different light entities. Left: Punctual omni-directional light. Sends light at all directions equally. Right: Area light. Sends light uniformly from its surface area.

Oculus User Node

The Oculus user node utilizes the graph hierarchical structure contained inside the virtual world to correctly synchronize and handle movement of head and eyes as well as other simulated body parts that may correspond to hardware devices (touch controllers etc.). In this present implementation, the front-end Oculus Window system propagates HMD tracking information used by the world to transform the Oculus user node and its sub-nodes in order to match user's position and orientation. The VR path tracer is mainly interested in the transformations of the eyes, which are in fact handled as virtual cameras (see Figure 3.8). In reality, the VR path tracer ignores most of the user data and mostly uses information inside those two camera nodes to trace the scene from their perspective.

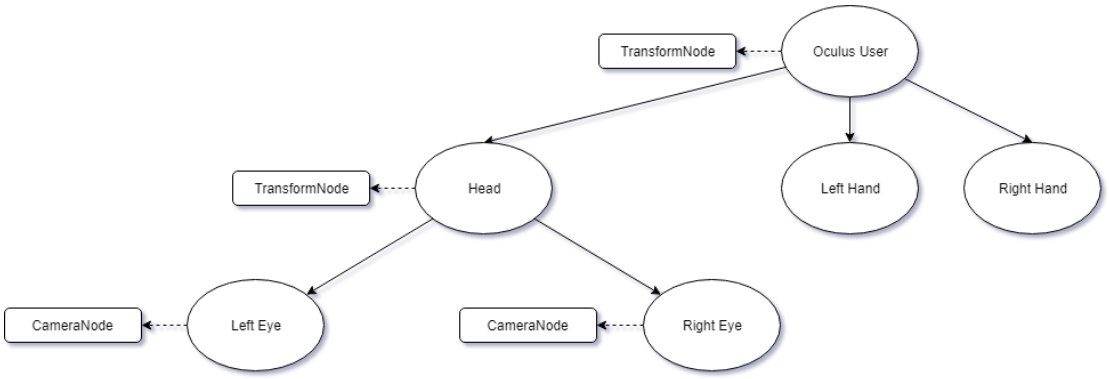

FIGURE 3.8: Oculus user sub-graph.

3.3.2 Path Tracing Stage

After expressing RT-XEngine's scene nodes to OptiX nodes and shader input, the renderer is ready to begin construction of its results. Cameras describe their position **p** as 3-dimensional vectors. Primary rays use this point in 3D space as their origin **o**. To find out the primary rays' direction **r** however, we need to perform several calculations. Starting from the pixel coordinates in the frame buffer of height h and width w of each camera/eye, we use the camera's reference frame definition to compute the world-space ray direction passing through each pixel. The camera's orientation is described by its **n** (front), **v** (up) and **u** (right) vectors. Additionally, to sample the proper viewport that corresponds to our camera, we need to take into consideration the camera's FOV and focal length (see Figure 3.9). Finally, to construct the primary rays using the aforementioned camera information, including the horizontal and vertical FOV half angles ϕ and θ and the camera's focal length f we use the following set of Equations 3.2.

$$
\begin{aligned}
v' &= v \cdot tan\theta \cdot f, \\
u' &= u \cdot tan\phi \cdot f, \\
d_i &= 2\frac{i}{h} - 1, \\
d_j &= 2\frac{j}{w} - 1, \\
o_{ij} &= p, \\
r_{ij} &= \frac{u' \cdot d_i + v' \cdot d_j + n}{|u' \cdot d_i + v' \cdot d_j + n|}.
\end{aligned}
\tag{3.2}
$$

where:

$i =$ *vertical pixel index*
$j =$ *horizontal pixel index*

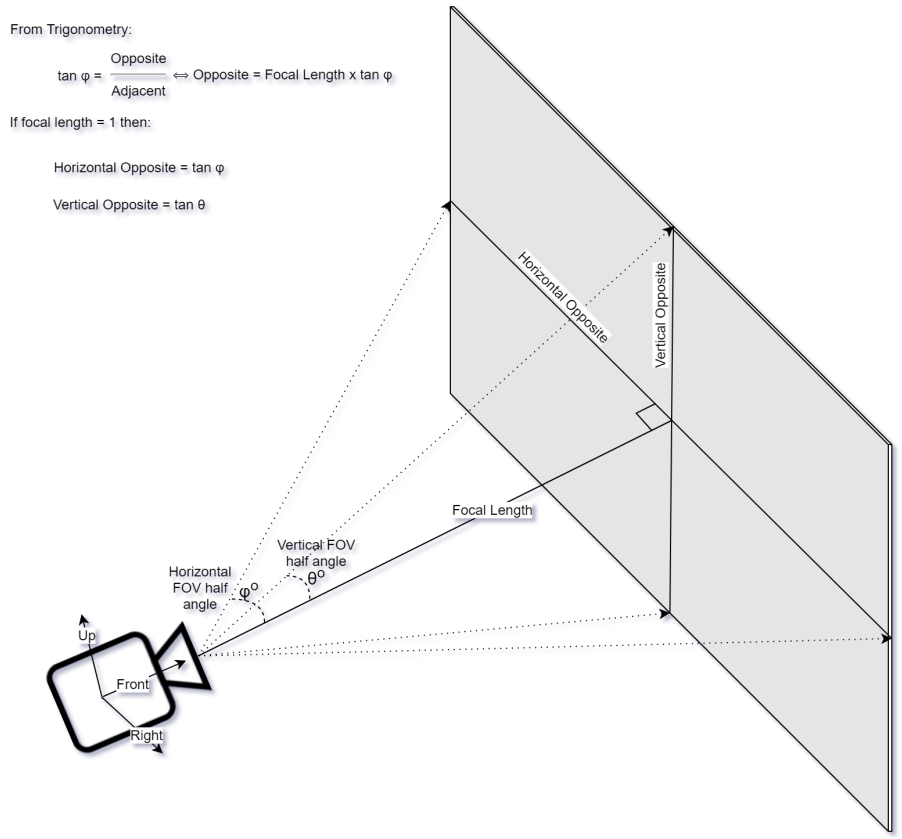

FIGURE 3.9: Camera viewport/film plane.

To avoid deep recursion, shading is performed in an iterative fashion inside the ray generation program. Primary rays are spawned from the eye positions and are shot inside the scene passing through the film planes, similarly to how it is described in the previous paragraph. In our case, jittering inside the area of pixels is performed. As shown in Figure 3.10, this technique reduces aliasing on the final result. Apart from origin and direction, rays are given initial depth, radiance, throughput, random seed and a "done" flag for early termination due to ray misses. The main ray generation and tracing loop is described in Algorithm 2.

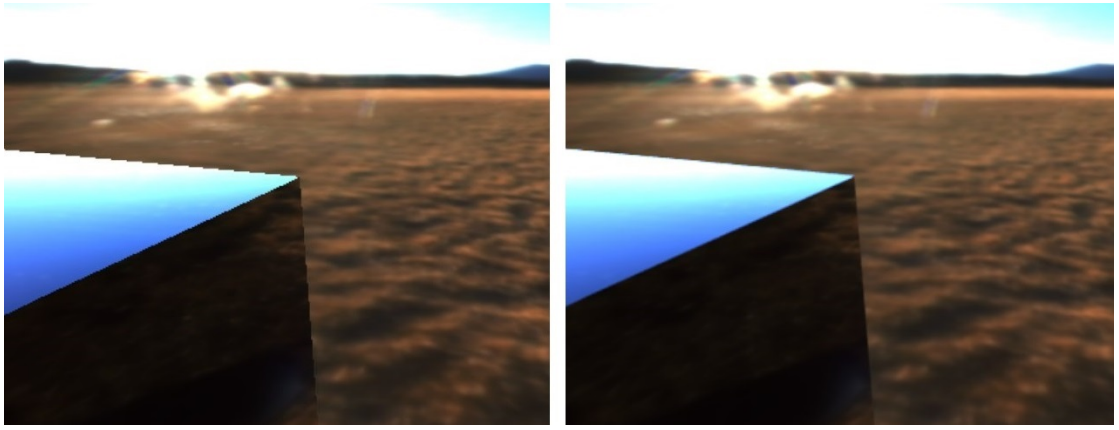

FIGURE 3.10: Jittering. Left: No jittering. Right: Jittered result.

Algorithm 2 Ray Generation

1: Generate jittered rays adjusted to the appropriate eye parameters // See set of Equations 3.2
2: **for all** generated rays **do**
3: **for all** *samples_per_pixel* **do**
4: Initialize path payload
5: **for all** bounces **do**
6: $trace(ray_{ij})$ // i and j correspond to the pixel coordinates through which the primary ray of this path entered the scene
7: **if** $ray_{ij}\_payload\_done$ **then**
8: break // Early termination due to ray miss
9: **end if**
10: **if** $ray_{ij}\_payload\_depth \geq 5$ **then**
11: $p = max_{xyz}(ray_{ij}\_payload\_throughput)$ // Max RGB channel value
12: **if** $rand[0-1] \geq p$ **then**
13: break // RR termination
14: **end if**
15: $ray_{ij}\_payload\_throughput \backslash= p$
16: **end if**
17: $ray_{ij}\_payload\_depth$ += 1
18: **end for**
19: $result$ += $ray_{ij}\_payload\_radiance$;
20: Update ray data for the next path segment
21: **end for**
22: $output\_buffer[i,j] = \dfrac{result}{samples\_per\_pixel}$
23: **end for**

During ray tracing, ray payload is updated depending on the hit or miss event that occurred. Those events cause OptiX to execute specific user-defined shaders. If a ray does not hit any surface, the miss program executes a set of Equations similar to 3.1 and adjusts the ray's radiance payload to match the environment lighting and subsequently flags this path for early termination.

Whenever there is intersecting geometry, closest hit programs (Algorithm 3) are used for each rendering equation evaluation step. Applying principles discussed in Section 2.2.3, this shading algorithm calculates direct lighting by sampling point lights using shadow rays to check whether there is any geometry between the intersected surface and the light emitter. Local illumination results are added to the ray's radiance payload. Algorithm 4 describes this process. Results are shown in Figure 3.11.

Algorithm 3 Closest Hit Program

1: Calculate shading normal from geometric normal and any other inputs (e.g. mesh data, normal maps)
2: $hit\_point = ray\_origin + t\_hit \cdot ray\_direction$ // t_hit is the reported distance from the closest hit intersection
3: Calculate direct illumination // See Algorithm 4
4: Calculate indirect illumination and next ray direction // See Algorithm 5
5: $ray\_payload\_origin = ray\_origin$ // Store inside path's payload for next bounce (see Algorithm 2 row 20)
6: $ray\_payload\_direction = \omega_i$ See Algorithm 5

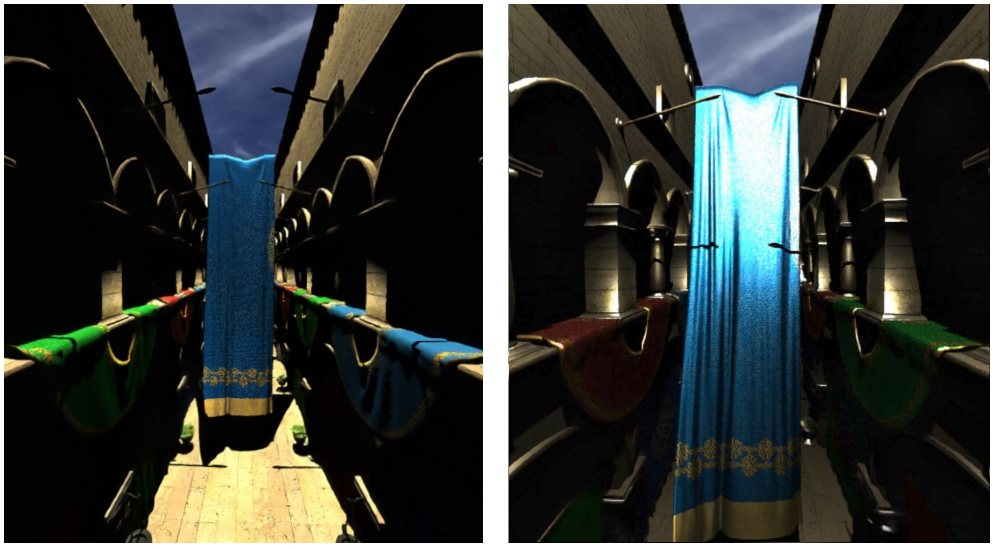

FIGURE 3.11: Examples of direct lighting. Left: Non-metallic and rough. Right: Metallic and smooth.

Algorithm 4 Direct Illumination

1: $light\_distance = length(light\_position - hit\_point)$
2: $point\_to\_light\_direction = normalize(light\_position - hit\_point)$
3: **if** $cos(\theta) > 0$ **then**
4: $shadow\_ray\_origin = hit\_point$
5: $shadow\_ray\_direction = point\_to\_light\_direction$
6: $trace(shadow\_ray)$
7: **if** $shadow\_ray\_payload\_in\_shadow$ **then**
8: $ray\_payload\_radiance = \frac{light\_color \cdot light\_intensity \cdot cos(\theta)}{light\_distance \cdot light\_distance}$
9: **end if**
10: Calculate direct $BRDF_{Specular}$ // See Equations 2.1, 2.5, 2.4, 2.6 and 2.7
11: Calculate direct $BRDF_{Diffuse}$ // See Equation 2.8
12: $ray\_payload\_radiance \cdot= ray\_payload\_throughput \cdot (BRDF_{Specular} + BRDF_{Diffuse})$
13: **end if**

Indirect illumination is a lot more involved. It currently uses the smooth normal of the intersecting surface to transfer operations from and to a local surface space to reduce their complexity. Depending on the material properties and the ray's seed payload, the shader randomly samples a new direction used for the next path segment. Its also responsible for updating the throughput payload that will be later used to attenuate the subsequent ray's contribution. Algorithm 5 describes how this is performed. Examples of indirect illumination are shown in Figure 3.12.

FIGURE 3.12: Examples of indirect illumination.

Algorithm 5 Indirect Illumination

1: Sample properties such as metallic, roughness and albedo from the material
2: Calculate Fresnel term "F" on this surface point // See Equation 2.5
3: **if** $rand[0-1] > F$ **then**
4: Sample a cosine weighted hemisphere for a new direction "ω_i" with "$pdf_d = \frac{cos(\theta)}{\pi}$„
5: $ray\_payload\_throughput \cdot= \frac{BRDF_{Diffuse} \cdot cos(\theta)}{\frac{cos(\theta)}{\pi}} = \frac{\frac{albedo}{\pi} \cdot cos(\theta)}{\frac{cos(\theta)}{\pi}} = albedo$ // See Equation 2.8
6: **else**
7: Sample new direction "ω_i" and its "pdf_s" based on the distribution of the GGX BRDF
8: Calculate $BRDF_{Specular}$ // See Equations 2.1, 2.5, 2.4, 2.6 and 2.7
9: $ray\_payload\_throughput \cdot= \frac{BRDF_{Specular} \cdot cos(\theta)}{pdf_s}$
10: **end if**

Finally, to leave space for future optimizations (such as reprojection mentioned in Section 2.3.1), eyes are rendered sequentially. This means that two separate OptiX context launches are issued. The ray generation shader performs ray tracing and shading for the left eye and stores the result inside the left half of the output buffer. Next, it renders the right eye, whilst being able to use any shading data calculated from the first eye. When

shading is completed, the output buffer is ready to be DMA-transferred (see Section 3.2.2) into the texture swap chain and be distorted and presented to the HMD through the Oculus SDK.

Overall, this is a basic path tracing model, however this does not mean that its inadequate. When using a lot of samples, our renderer is able to produce a good level of photorealism. It does so even if there is a lack of representation of more intricate phenomena. It is still not a viable option to expand our path tracer to include more sophisticated and demanding effects. For the time being, we should experiment on a more primitive model of path tracing to understand the limits and issues such an implementation introduces in a VR context.

3.3.3 Denoising Stage

In a best case scenario, our HMD is supplied with the maximum frames per second its refresh rate is able to support (75 in DK2's case). Many samples per pixel (SPP) and a potentially infinitely deep ray path have detrimental effects on the real-time rendering capabilities of the VR path tracer. Increasing samples per pixel greatly affects performance. Therefore, keeping SPP as low as possible (usually one or two) may be mandatory in order to achieve real-time results. Additionally, RR handles early path termination in a purely probabilistic way. This means that there may be paths that are long enough to cause framerate issues. In this case, implementations typically use only a few indirect bounces as a hard limit to the algorithm's depth.

Results shown in Figure 3.13 are achieved in real-time. Unfortunately, with so few samples, images are nowhere near an acceptable convergence. As already noted, OptiX features an AI-based denoiser that has been trained to clean those noisy and grainy images. The denoiser is integrated as a post-process step and is applied at the end of the VR path tracer's rendering pipeline. To be more specific, the renderer creates an OptiX command list, that contains all the required steps to produce a clean image. Apart from the denoiser, this list includes left and right eye context launches (see Section 3.1.2). When those two are done the denoiser uses information accumulated during the path tracing procedure to produce the final results that are displayed by the HMD.

Additional buffers for the denoiser are used to improve its outputs by giving a more detailed description of the surfaces currently visible in the input image. One of those is the albedo buffer. Values stored inside it can be thought as the surfaces color unaffected by any lighting. Since the VR path tracer uses PBR materials, this value is easily obtained from a surface's corresponding albedo map. For some objects however, such as perfect mirrors, to improve quality, it is generally better to use the albedo value of the subsequent surface hit, as the mirrored content is the one that must be denoised

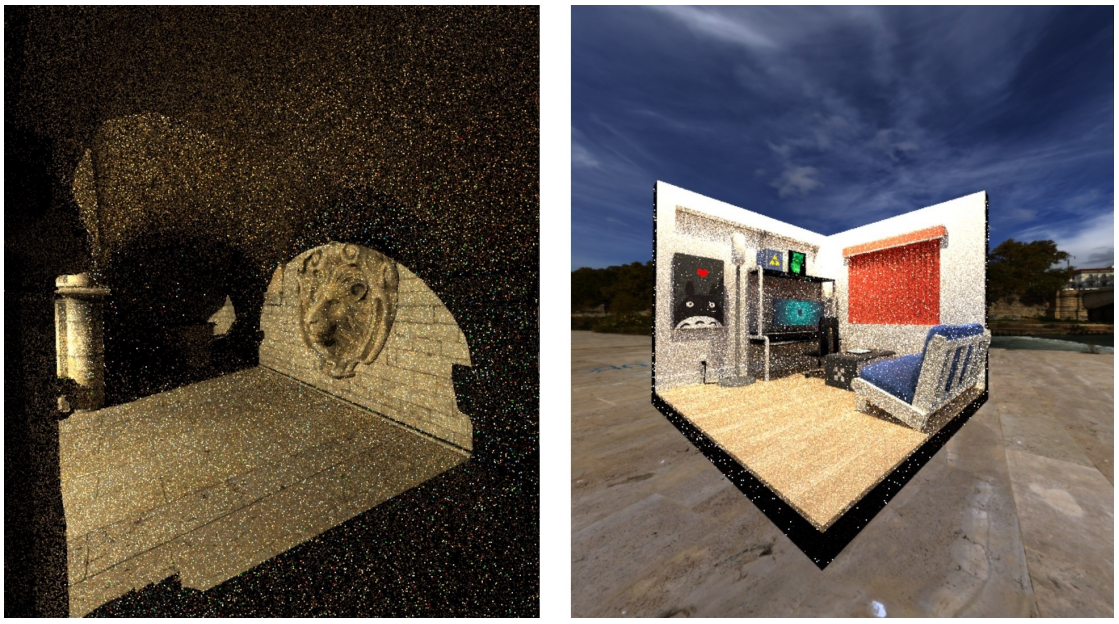

FIGURE 3.13: Unconverged real-time examples.

instead. The second input buffer contains visible surface normal information expressed in camera space. In this space, the camera's front vector looks down the negative z axis. As for the up and right vectors, those are aligned with the positive y and x image axes respectively.

To assemble the additional input surface data, adjustments to the algorithms described in the previous subsections need to applied. That is because, accumulation of camera space normals and albedo is performed during the path tracing stage. Therefore, during execution of closest hit programs, if the path depth is equal to zero, i.e. first direct hit, the path tracer should gather this information from the active surface material. When the path tracing stage is completed, all of the required and optional buffers are ready to be used in the denoising process (see Figure 3.14).

3.3.4 Lens-matched Shading/Sampling

As mentioned in Section 2.3.1, distortion rendering results in efficiency losses, when the HMD is supplied with rectilinear images such as those our VR path tracer produces. Lenses used by an HMD are typically sharper at the center and blurrier around it. Our renderer spends an equal amount of effort to shade peripheral areas to that it takes it to render the center of each eye. This means that it wastes valuable time to calculate data that will later be distorted. To combat this issue, we adjust the sampling rate of the central region and the periphery. In a ray tracer, this equates to changes in ray spawning density. In simpler words, for each virtual eye, pixels near the center of its film plane would be more densely distributed than the surrounding areas. To achieve

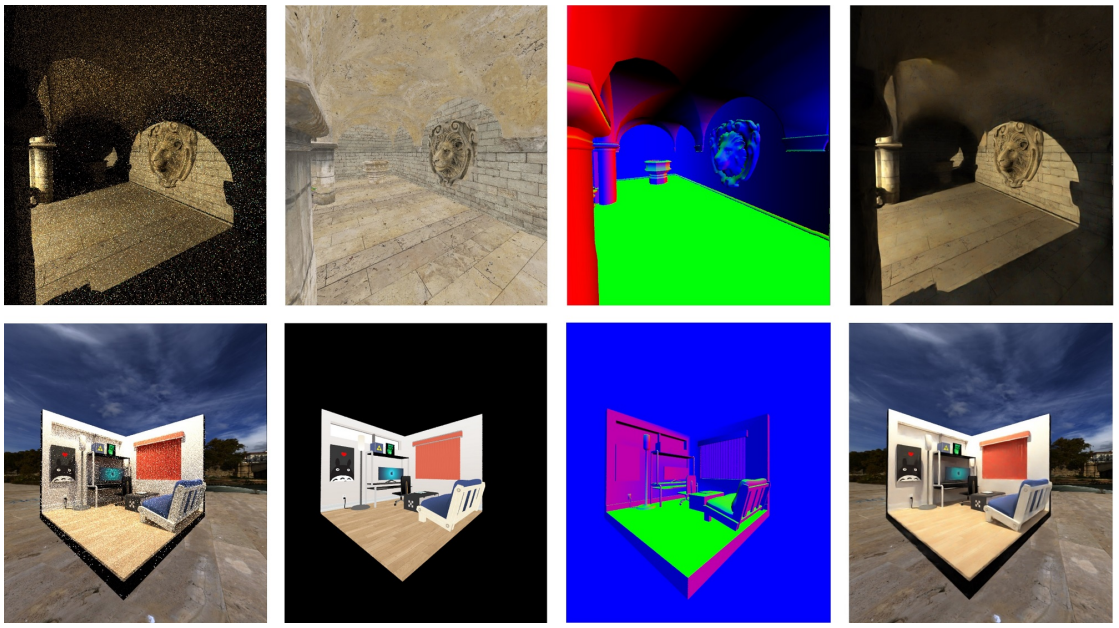

FIGURE 3.14: Denoising path traced results using albedo and camera space normal information. Left: Path traced buffers. Middle-Left: Albedo buffers. Middle-Right: Camera space normal buffers. Right: Denoised buffers (see for Figure 3.15 better resolution).

FIGURE 3.15: Denoised real-time examples.

this, we would have to override the distortion applied by the Oculus SDK after frame submission, otherwise the HMD would be given a doubly distorted input. Unfortunately, this is a functionality not yet available to the Oculus SDK.

This does not mean however, that the path tracer cannot still benefit by adjusting its samples per pixel in a lens-matched fashion. Figure 3.17 shows concentric circles drawn around the axis of projection for an eye. The figure demonstrates how after distortion the central areas are magnified. This is due to the barrel distortion applied on the

rectilinear image to cancel out the lens-based distortion occurring from the HMD. If we split the view in concentric zones and allocate a different number of samples per pixel in each one, with only one SPP in the periphery (darkest zone in the figure), we can save a significant amount of samples and gain a performance boost, while concentrating samples where they count most.

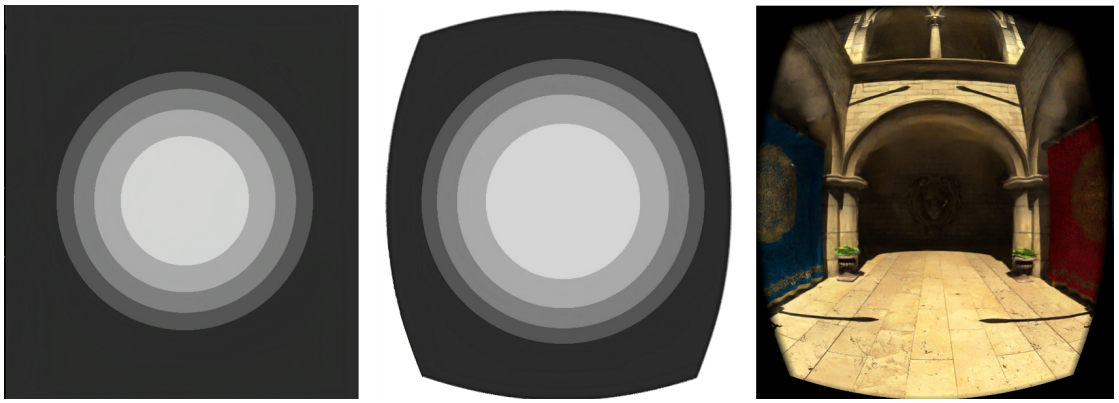

FIGURE 3.16: Lens-matched sampling example.

Algorithm 6 Lens Matched Shading Mask

1: $distance\_from\_center = length(pixel - eye\_center)$
2: $normalized\_distance\_from\_center = \dfrac{distance\_from\_center}{0.5 \cdot eye\_width}$ // 0 to 1
3: $samples\_per\_pixel = 1 + ceil(0.125 \cdot (1 - normalized\_distance\_from\_center) \cdot (max\_samples\_per\_pixel - 1)$
4: $magic\_number = 0.4 \cdot eye\_width$
5: **if** $distance\_from\_center \leq magic\_number$ **then**
6: $inner\_normalized\_distance\_from\_center\_squared = (\dfrac{distance\_from\_center}{magic\_number})^2$ // 0 to 1, inside the sampling mask
7: $samples\_per\_pixel+ = ceil(1 - inner\_normalized\_distance\_from\_center\_squared) \cdot (max\_samples\_per\_pixel - 1)$
8: **end if**

Note/Disclaimer: $magic\_number$ is adjusted so that the sampling mask covers most of the visible texture area inside the HMD which in DK2's case is roughly equal to $0.4 \cdot eye\_width$. This Algorithm does not operate based on actual lens characteristics of DK2 but rather on empirical evidence that emerged during experimentation.

Chapter 4

Evaluation & Future Work

4.1 Rendering Algorithm

4.1.1 VR Path Tracer

Although the tracing model used in the VR path tracer certainly lacks the representation of many phenomena, the high quality of its results, combined with the denoising capabilities of the Optix AI-Denoiser were quite encouraging (see Figure 4.1).

FIGURE 4.1: Photorealistic results of the VR path tracer (non real-time -however they were produced in less than 200 milliseconds). Left: Global illumination. Right: "Metallic Sponza".

On the other hand, however, due to the high resolution demands of VR, the cost of tracing path and the time it takes for it to be denoised (see Diagram 4.2), it seems that we are forced to sacrifice some of the quality that is key to the VR experience, in order to

achieve real-time performance. However, we need to take into account the fact that this is a pure ray tracing pipeline. This gives us the opportunity to experiment with several ray tracing based optimizations that we will discuss later in this chapter. Additionally, we could also test how different strategies would fare in terms of their effect on quality and performance, for example if we employed some form of hybrid rendering (see Section 2.1.3), primary rays could have been avoided if a rasterization pass identified the primary hits.

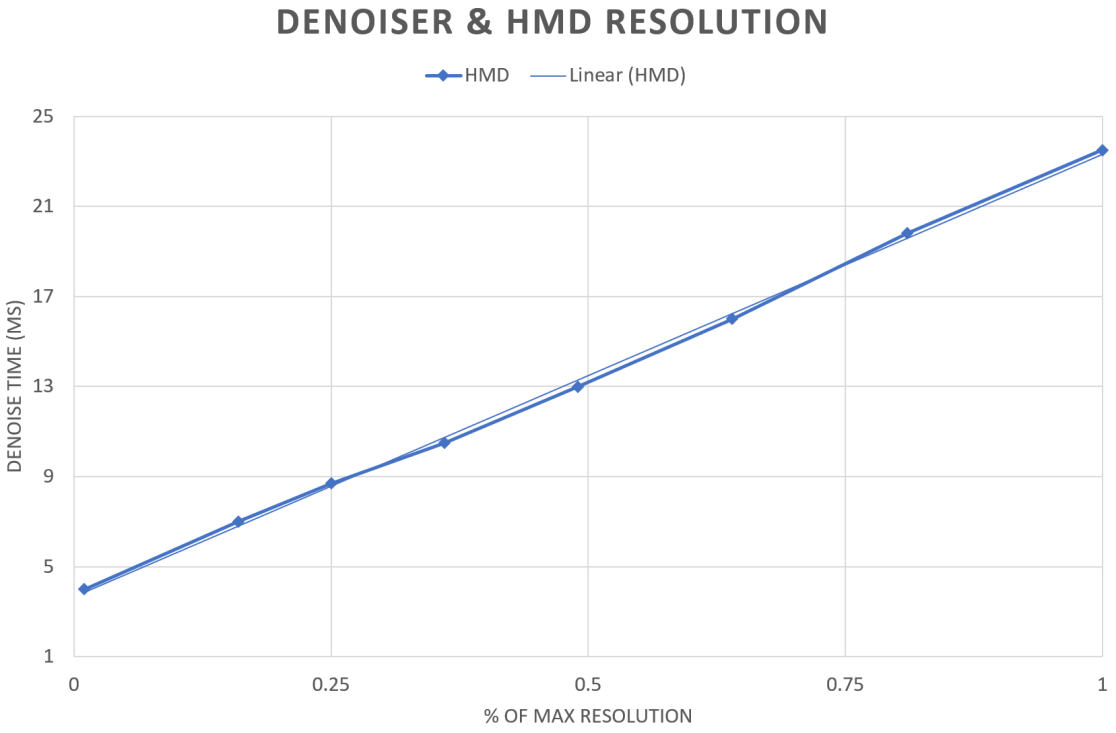

FIGURE 4.2: Denoise time based on pixel per HMD display pixels (maximum SDK-suggested texture resolution of the DK2 HMD: 2432x1472).

Theoretically, due to optimizations that smooth out VR (ASW 1.0 and ATW, see Section 2.3.2), when the v-sync deadline is barely missed, we should still have a stable experience without much noticeable judder and jerkiness. Therefore, for our evaluation, results that are presented at a constant frame rate of at least half the refresh rate of the HMD (37.5Hz for DK2) are considered to be real-time. Figure 4.3 showcases two real-time examples in comparison to a result that uses many more SPP and indirect bounces. All of the above used $\frac{1}{4}$ of the maximum resolution of the eye buffer.

In Diagram 4.4, we record several tests made inside the "Sponza" [tey14] scene. The way we performed those tests was by measuring the average frame preparation time with and without denoising based on one or two SPP and zero to ten indirect bounces. Cases that managed to get an average preparation time of less than 13.3 milliseconds achieved the DK2's frame rate target. Ones between that and around 26.7 milliseconds (missing every other v-sync deadline), provided an adequately smooth experience due to optimizations such as ATW and ASW. After the fifth indirect bounce, RR termination

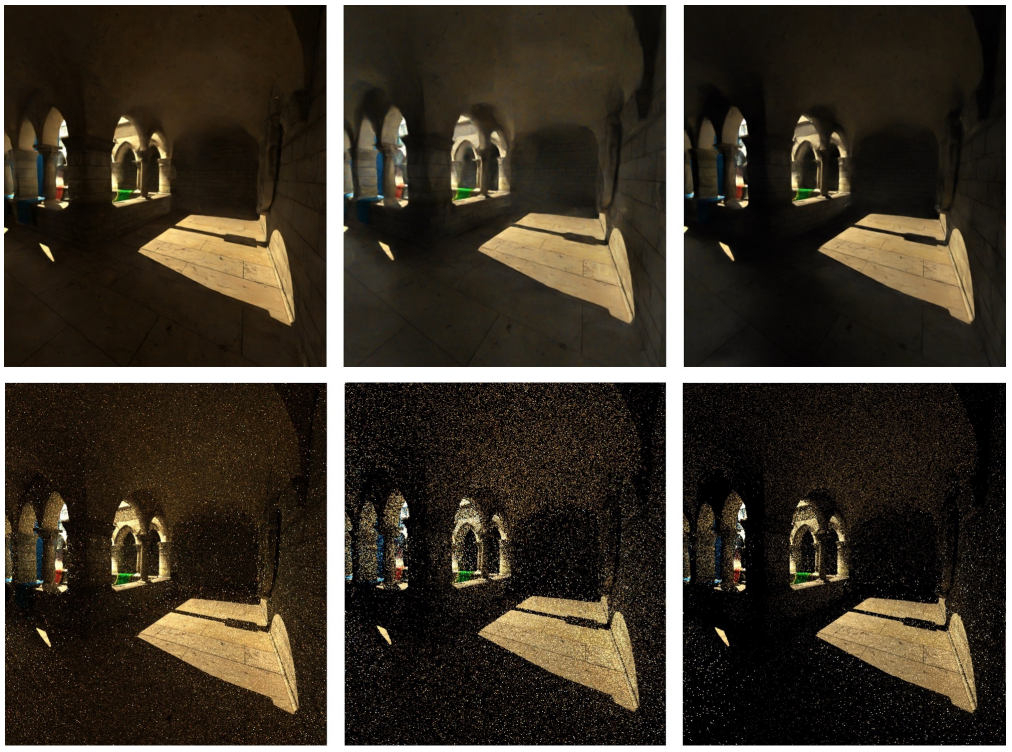

FIGURE 4.3: Comparison of denoised results in different levels of convergence and path depth. Left: Many SPP, RR terminated. Middle: 1 SPP, 2 indirect bounces. Right: 2 SPP, 1 indirect bounce.

checks are enabled, which is evident from the difference in performance scaling. Similarly to previous experiments, the eye buffer size was set to $\frac{1}{4}$ of its maximum resolution. As a result, as seen from the diagram (Diff 1 SPP and Diff 2 SPP), we managed to keep the denoising delay lower than 10 milliseconds.

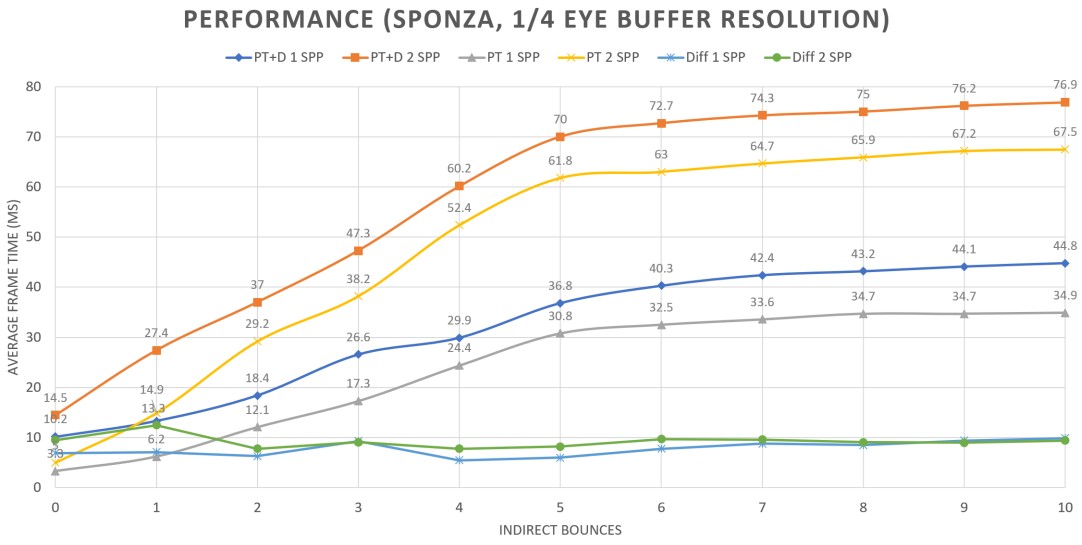

FIGURE 4.4: Frame delay. Notes: PT+D is the path tracing plus the denoising time which is approximately the application frame submission delay (other delays are negligible). Diff refers to the difference between PT+D and PT results and measures the denoising delay.

Finally, apart from its performance and denoising capabilities, our renderer should also be evaluated based on the physical correctness of its results. This equates to a proper representation of the interaction of light with certain materials. Since our model focuses on how light is reflected and/or diffusely scattered from surfaces we provide the following Figure 4.5 that shows how roughness values affect those phenomena on metallic and non-metallic surfaces. As seen from this figure, the VR path tracer is capable of producing a wide range of photorealistic results based on different material properties.

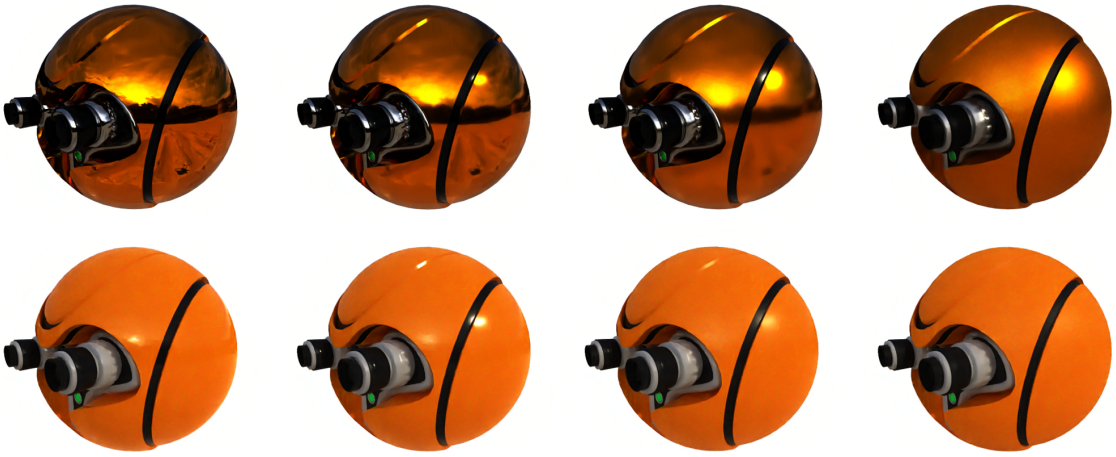

FIGURE 4.5: Metallic and non-metallic surfaces. Top row: Metallic surfaces. Bottom row: non-metallic surfaces. Note: Left to right increase in roughness, values are indicative and should not be compared between columns and rows.

Overall, the real-time performance of the VR path tracer was impressive, owing to the fact that it is still a somewhat early approach to real-time, stereoscopic and photorealistic rendering. The RTX-accelerated ray tracing and denoising capabilities of OptiX 6.0 (see Section 3.1.2) definitely played a major role in achieving these results. Be that as it may, it is also important to note, that a naive path tracing model would not be able to reach this level of photorealism, with so few samples.

4.1.2 Enriching The Model

- Area lights:

  Apart from point lights, our direct lighting model could also be extended to support area lights. As mentioned in Section 2.2.3, to sample those lights correctly, we need to integrate over their surface area. Typically, these are entities that carry information that describes simple geometrical shapes, such as rectangles or disks. Shadows cast inside a scene using area lights are softer and more realistic than the ones point lights produce. However, they generally require multiple shadow rays in order to converge. Since we are in scarcity of those, we should carefully evaluate whether area lights are a priority.

- Integration domain:

  As described in Section 2.1.2, the BRDF is a subset of the BSDF that describes light reflection on opaque surfaces. The BTDF on the other hand is a function that focuses on the other side of the surface, as its purpose is to define light transmission behaviour. As far as indirect lighting is concerned, our renderer uses the BRDF to describe illumination that originates from specular and diffuse reflections. This means that our model, currently integrates over a unit hemisphere (or non uniformly by the BRDF lobe, Section 2.2.2) above the surface of said point. For this model to be more complete, it should also be able to calculate contribution from transmission events such as refraction. Our renderer then, for a specific point, will be able to, based on the material properties, test indirect illumination by integrating over the point's surrounding unit sphere.

4.2 Ray Generation

4.2.1 Lens-matched Sampling

Lens-matched sampling is a simple technique that adjusts each path's SPP in a lens-matched fashion. As seen from Figure 4.6, by applying the sampling mask, shown in the top-middle of this figure, to the result shown in the image on the top-left we gain almost triple the performance. Moreover, as seen from the final outcome on the top-right, only the periphery is visibly affected (from inside the HMD this may vary depending on the quality of the peripheral vision of the user, and of course the user's focus which in a non-eye tracked HMD should be always at the center of the screen). In the bottom-right image we see that with LMS we manage to construct a real-time result that is almost equal in terms of quality with the non-real-time result shown in the bottom-middle. Additionally, in the bottom-left we see the current online rendering limitations of our renderer without use of LMS.

In Diagram 4.7, we record several tests with a similar setup to the ones we presented in the previous section. However in these experiments we calculated the frame preparation performance gains, measured in the frame delay difference, with and without applying LMS. In the case of LMS with one indirect bounce, the renderer achieved real time performance even when using six SPP, whereas in the case of two indirect bounces by using three SPP. Therefore, we conclude that by using lens-matched sampling, we can essentially increase the maximum number of samples per pixel (at the central zone), effectively improving the image quality of the important part of the eye view. Thus, we are able to supply the HMD with more converged results without losing real-time performance.

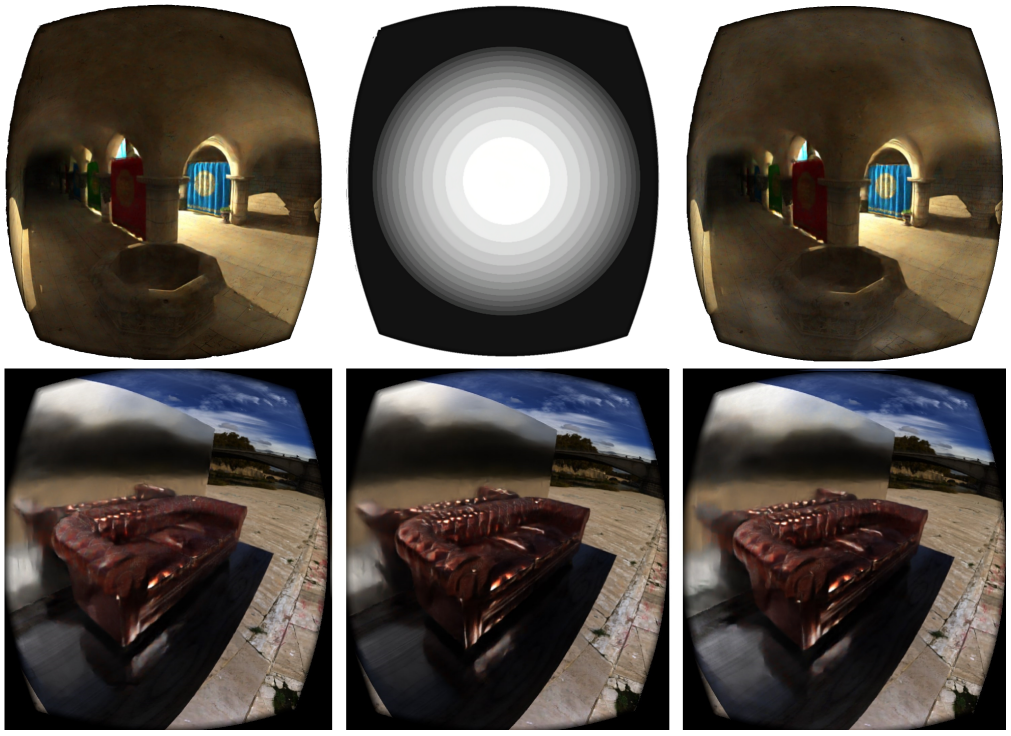

FIGURE 4.6: Lens-matched sampling. Top: Effect of the sampling mask. Bottom: LMS performance.

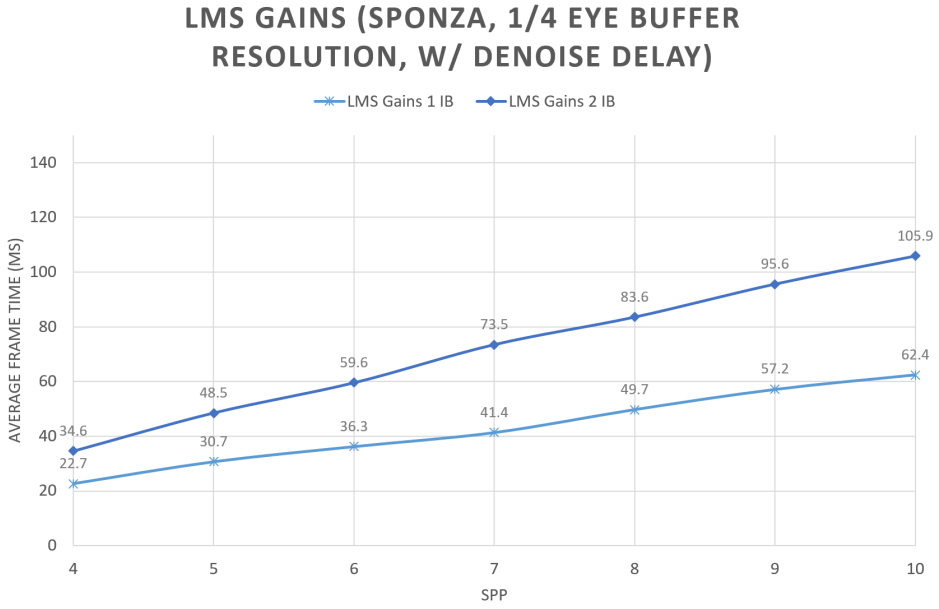

FIGURE 4.7: Performance gains using lens-matched sampling. Note: IB - indirect bounce of lights.

4.2.2 Optimizing the Rays

- Lens-matched ray generation (LMRG):

  Lens-matched sampling is a naive approach that uses empirical evidence to solve the efficiency losses that occur due to lens distortion. In fact, this technique would

pale in comparison to what a lens-matched ray generation approach would be able to achieve. An implementation of LMRG would densely generate rays closer to the center of the eyes and a thinner distribution on the periphery. This means that, even though the SPP of each pixel remains the same, the generated images will now have a resolution gradient that matches the sampling rate of the lens more properly. Knowing the exact lens and HMD properties, one could theoretically match the exact lens distortion from any HMD to avoid any efficiency losses described in Section 2.3.1. Nevertheless, this technique requires further research in terms of how the rays can be generated efficiently and if it could somehow be generalized for any HMD supplied with ray traced results.

- Ray/Path reprojection:

  As discussed in 2.3.1, with reprojection, information can be projected from one eye to another to effectively avoid many shading computations. In our case, this means that by using data calculated from the left eye, we could potentially avoid paths that are compatible to the right. Unfortunately, problems with reprojection arise when effects are highly view-dependant. For example, results that utilize the eye's view direction could potentially differ from eye to eye, such as highly-glossy or specular reflections. Therefore, we should probably reproject information originating from subsequent path segments (after the direct hit) as their contribution depends more on other factors (3D location, randomness, etc.).

- Asymmetric eye FOV:

  In [Zha18] it is explained that switching to asymmetric eye FOVs may increase performance due to reduction of around 20% of the total resolution of each eye buffer. For our renderer, this means fewer paths, and at this current implementation, faster denoising (see Figure 4.2).

4.3 Photorealistic Virtual Reality

4.3.1 Current VR Experience

Even when operating at half the FPS target of the HMD, our implementation can cause minor (subjective) dizziness and motion sickness. Additionally, the lower resolution of the eyes in conjunction with bad real-time denoise results can be rather unpleasant. However, as far as VR is concerned, priority should be given on matters that may be more disruptive to the whole experience. Following, we will discuss such issues and suggest possible counter-measures that may help alleviate their influence on the VR experience.

4.3.2 Improving the VR Experience

- Ray tracing stability:

  Apparent noise changes between frames produced by the path tracer are noticeable to the denoiser output and can be very disruptive to the whole VR experience (see Figure 4.8). The reason this occurs in the first place, has to do with the way random sampling is performed during path tracing.

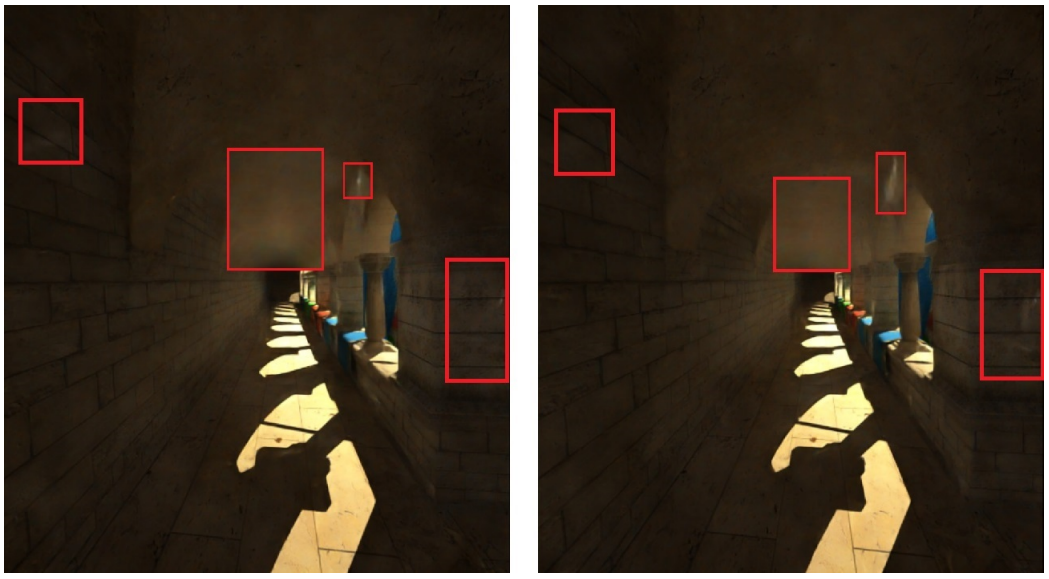

FIGURE 4.8: Consecutive frames (left eye).

  Typically, the way randomness is approached in shaders is by using pseudo-random generators. These generators are given a seed as a parameter to calculate a pseudo-random number. In our case, for paths to be consistent between frames, we supply our random generator with a seed derived from screen-space information (e.g. indices) of the pixel from which the path enters the scene. Having said that, random values on specific points on the surface of objects may change due to HMD movement. If the path traced result is well converged, then there will not be any visible differences in between denoised frames. However, if it is not converged, then even a small offset may be very apparent. Unfortunately, due to the HMD's constant position and orientation tracking updates, random values are almost never stable. [CSK+17] and [SSK+17] describe techniques that could be employed to increase our path tracer's stability.

- Noise generation:

  Other issues that are also very apparent to a denoised outcome that is less converged relate to the way we construct the seeds for our random generators. As mentioned before, in their payloads, rays carry seeds that are used during intersections to randomly sample the next ray's direction. Each time a seed is used by

a generator its value pseudo-randomly changes, in order to avoid generating the same value multiple times. As noted, the way those seeds are constructed is using index information from pixels. This way, between frames, the initial seed of a ray that entered the scene from a specific pixel, will always be the same. If our view remains stable, then pixels between frames will always result in the same value, giving some stability (even if the denoiser is temporally unstable).

In a stereoscopic context however, if the seed depends on the HMD's image buffer location, the noise disrupts the apparent disparity of the same location seen through the stereo image pair. As a consequence, parallax is negatively affected and leads to problems focusing on the displayed surfaces, after denoising. On the other hand, if the seed depends on each eye's image buffer location, almost the exact same noise will be generated to each eye in one-to-one pixel correspondence between them, making it appear monoscopic. This is not as discomforting as the previous case, however, it does negatively affect the stereoscopic experience. As a measure to alleviate those effects noise brings, we could try to stereoscopically match the seed construction from one eye to the other, similarly to as discussed for shading in reprojection (see Section 2.3.1).

- Depth information:

  When ASW 1.0 (see Section 2.3.2) is enabled, lower than half FPS produce noticeable reprojection artifacts (see Figure 4.9). Theoretically, this issue could be resolved if we used ASW 2.0 (see Section 2.3.2). In order to activate it, we need to manually construct an OpenGL (in our case) compatible depth buffer and supply it to the Oculus SDK. This is not yet implemented as calculation of depth information inside our path tracer was not used in some other, more vital computation.

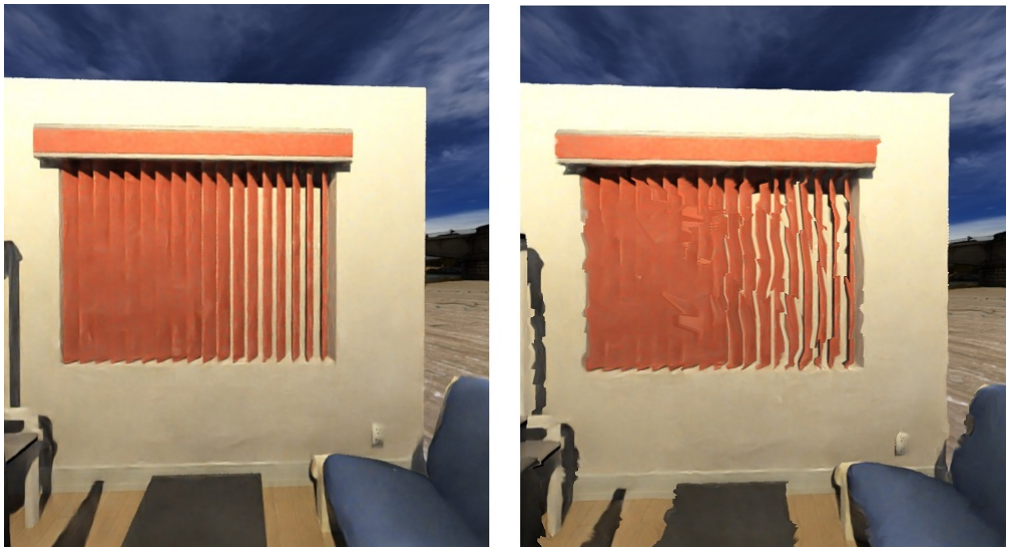

FIGURE 4.9: Reprojection artifacts occuring from ASW 1.0. Left: ASW off. Right ASW on.

Content Credits

Models and images showcased in Chapters 3 and 4:

- Tiber Island HDR panorama [Zaa17]: Figures 3.10, 3.12, 3.13, 3.14, 3.15, 4.1, 4.5, 4.6, 4.9.
- Sponza 3D model [tey14]: Figures 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 4.1, 4.3, 4.6, 4.8,
- Old leather sofa model 3D model [Pix17]: Figures 3.12, 4.6
- Test Room PBR 3D model [nic19]: Figure 3.13, 3.14, 3.15, 4.9

Bibliography

[AB19] Volga Aksoy and Dean Beele. Introducing ASW 2.0: Better Accuracy, Lower Latency, April 2019. *https://www.oculus.com/blog/introducing-asw-2-point-0-better-accuracy-lower-latency/*. (cit. on 2.3.2, 2.33)

[Amo11] Evan Amos. Virtual-Boy-Set, September 2011. *https://commons.wikimedia.org/wiki/File:Virtual-Boy-Set.png*. (cit. on 1.10)

[Amo15] Evan Amos. Google-Cardboard, December 2015. *https://commons.wikimedia.org/wiki/File:Google-Cardboard.jpg*. (cit. on 1.6)

[Amo17] Evan Amos. Sony-PlayStation-4-PSVR-Headset-Mk1-FL, June 2017. *https://commons.wikimedia.org/wiki/File:Sony-PlayStation-4-PSVR-Headset-Mk1-FL.jpg*. (cit. on 1.12)

[Ant15] Michael Antonov. Asynchronous Timewarp Examined, March 2015. *https://developer.oculus.com/blog/asynchronous-timewarp-examined/*. (cit. on 2.3.2)

[Bae07] John Baeder. John's Diner with John's Chevelle, 2007. *https://commons.wikimedia.org/wiki/File:John%27s_Diner_by_John_Baeder.jpg* (CC BY-SA 3.0). (cit. on 1.1)

[Bar18] Dom Barnard. History of VR - Timeline of Events and Tech Development. *VIRTUALSPEECH*, August 2018. *https://virtualspeech.com/blog/history-of-vr*. (cit. on 1.2.2)

[BB88] H.-J. Bär and P. Bopp. M. h. kalos and p. a. whitlock:monte carlo methods, volume i:basics, john wiley and sons, new york, chichester, brisbane, toronto and singapore 1986, library of congress QA298.k35 1986. 186 seiten mit einem index, preis: £ 29.00. *Berichte der Bunsengesellschaft für physikalische Chemie*, 92(4):560–560, April 1988. (cit. on 2.2.2)

[BHP16] Dean Beeler, Ed Hutchins, and Paul Pedriana. Asynchronous Spacewarp, November 2016. *https://developer.oculus.com/blog/asynchronous-spacewarp/*. (cit. on 2.3.2)

[Bra19] Alan Bradley. The best games to show off your brand new graphics card. *PC GAMER*, February 2019. *https://www.pcgamer.com/the-best-games-to-show-off-your-brand-new-graphics-card/*. (cit. on 2.13, 2.20)

[CKS+17] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics*, 36(4):1–12, July 2017. (cit. on 3.1.2)

[Cor18a] Nvidia Corporation. NVIDIA VIRTUAL GPU. *https://www.nvidia.com/en-us/design-visualization/solutions/virtualization/*, 2018. (cit. on 1.1.2)

[Cor18b] Nvidia Corporation. RTX. IT'S ON. NVIDIA TURING. *https://www.nvidia.com/en-us/geforce/turing/*, 2018. (cit. on 1.3.1, 2.1.3)

[CSK+17] Alessandro Dal Corso, Marco Salvi, Craig Kolb, Jeppe Revall Frisvad, Aaron Lefohn, and David Luebke. Interactive stable ray tracing. In *Proceedings of High Performance Graphics on - HPG 17*. ACM Press, 2017. (cit. on 4.3.2)

[CT82] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics*, 1(1):7–24, jan 1982. (cit. on 2.1.2)

[CUM18] Nvidia Corporation, Aalto University, and MIT. Research at NVIDIA: AI Can Now Fix Your Grainy Photos by Only Looking at Grainy Photos. *https://www.youtube.com/watch?v=pp7HdI0-MIo*, 2018. [Online; posted 9-July-2018]. (cit. on 1.1.2)

[Dor17] Luke Dormehl. 8 virtual reality milestones that took it from sci-fi to your living room. *Digital Trends*, November 2017. *https://www.digitaltrends.com/cool-tech/history-of-virtual-reality/*. (cit. on 1.2.2)

[DS13a] Walt Disney and Animation Studios. BRDF Explorer - Development and analysis of bidirectional reflectance distribution functions (BRDFs). *https://www. disneyanimation.com/technology/brdf.html*, 2013. (cit. on 2.8, 2.9)

[DS13b] Walt Disney and Animation Studios. Disney's Hyperion Renderer. *https://www.disneyanimation.com/technology/innovations/hyperion*, 2013. (cit. on 2.19)

[Duv18] Julien Duval. Renderman Documentation - Shadows, October 2018. *https://rmanwiki.pixar.com/display/REN22/Shadows*. (cit. on 1.2)

[Ele15] Samsung Electronics. Samsung gear vr, November 2015. *https://www.samsung.com/ae/wearables/gear-vr-r322/*. (cit. on 1.12)

[ENSB13] Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. *Computer Graphics Forum*, 32(4):125–132, July 2013. (cit. on 2.19)

[Eri04] Christer Ericson. *Real-Time Collision Detection*. CRC Press, 2004. (cit. on 2.1.3)

[eVR14] eVRydayVR. Oculus Rift - How Does Time Warping Work?, 2014. *https://www.youtube.com/watch?v=WvtEXMlQQtI*. (cit. on 2.31)

[Fee19] Feelreal Inc. Feelreal - The World's First Multisensory VR Mask, 2019. *https://www.indiegogo.com/projects/feelreal-the-world-s-first-multisensory-vr-mask--2#/*. (cit. on 1.13)

[Gam16] Epic Games. Unreal Engine 4 Documentation - Reflections, 2016. *https://docs.unrealengine.com/en-US/Resources/Showcases/Reflections/index.html*. (cit. on 1.3)

[Gam18] Playground Games. *Forza Horizon 4*. Microsoft Studios, October 2018. (cit. on 1.1)

[Gmb19] Cybershoes GmbH. Cybershoes: A Step Into Virtual Reality Games, 2019. *https://www.indiegogo.com/projects/cybershoes-a-step-into-virtual-reality-games--3#/*. (cit. on 1.13)

[HA18] Song Ho Ahn. OpenGL Pixel Buffer Object (PBO), September 2018. *http://www.songho.ca/opengl/gl_pbo.html*. (cit. on 3.2.2)

[HC16] HTC and Valve Corporation. HTC Vive, 2016. *https://www.vive.com/us/product/vive-virtual-reality-system/*. (cit. on 1.12)

[Hea14] Tom Heath. Engine Integration, December 2014. *https://developer.oculus.com/downloads/package/oculus-sdk-for-windows/*. (cit. on 3.1.3)

[Hen08] Henrik. Ray trace diagram, April 2008. *https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg*(GNU FDL). (cit. on 2.16)

[Hof16] Alex Hoffman. Subsurface Scattering (sss), 2016. *https://cgblender.com/subsurface_scattering*. (cit. on 1.2)

[HR96] Ian P. Howard and Brian J. Rogers. *Binocular Vision and Stereopsis*. Oxford University Press, February 1996. (cit. on 1.2.4)

[Hug18] James Hughes. Oculus SDK support for NVIDIA VRWorks Lens Matched Shading, March 2018. *https://developer.oculus.com/blog/oculus-sdk-support-for-nvidia-vrworkstm-lens-matched-shading/*. (cit. on 2.3.1, 2.28)

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH 86*. ACM Press, 1986. (cit. on 2.2.1)

[Kau16] Hannes Kaufmann. Virtual and Augmented Reality, Lecture Slides, TU Wien, 2016. (cit. on 1.4)

[Kob16] Kobbaka. View-Master model L, June 2016. *https://commons.wikimedia.org/wiki/File:View-Master_model_L.jpg*(CC BY-SA 4.0). (cit. on 1.6)

[Lam60] J. H. Lambert. *Photometria, sive, De mensura et gradibus luminis, colorum et umbrae*. Augsburg, Germany: V. E. Klett, 1760. (cit. on 2.1.2)

[Liu17] Edward Liu. Lens Matched Shading and Unreal Engine 4 Integration, March 2017. *https://developer.nvidia.com/lens-matched-shading-and-unreal-engine-4-integration-part-1*. (cit. on 2.26, 2.27)

[LMH+18] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2Noise: Learning Image Restoration without Clean Data. *CoRR*, abs/1803.04189, 2018. (cit. on 1.1.2, 1.3.1, 2.2.4, 2.22)

[Low11] Colin Lowndes. Ely Darkcity, 2011. *http://www.custommapmakers.org/skyboxes/zips/ely_darkcity.zip*(GNU GPL2). (cit. on 3.6)

[Mor18] Keith Morley. NVIDIA OptiX Ray Tracing Powered by RTX, April 2018. *https://devblogs.nvidia.com/nvidia-optix-ray-tracing-powered-rtx/*. (cit. on 3.1.2)

[Mos17] Ioannis Moschos. Volumetric Lighting & Environment Mapping. Bacherlor's thesis, Athens University of Economics and Business, sep 2017. (cit. on 1.2, 4)

[MTUK95] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. titleaugmented reality: a class of displays on the reality-virtuality continuum/title. In Hari Das, editor, *Telemanipulator and Telepresence Technologies*. SPIE, December 1995. (cit. on 1.2.1, 1.4)

[nic19] nickheitzman. Test Room PBR, 2019. *https://hdrihaven.com/hdri/?c=night&h=shanghai_bund*(CC BY 4.0). (cit. on 4.3.2)

[NW09] Greg Nichols and Chris Wyman. Multiresolution splatting for indirect illumination. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D 09*. ACM Press, 2009. (cit. on 2.1.2)

[Ocu12] Oculus. Oculus Rift: Step Into the Game, August 2012. *https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game*. (cit. on 1.11)

[OKS16] Hiroya Oku, Tsutomu Kuroiwa, and Keiichi Sato. Gantz: O, October 2016. (cit. on 1.1)

[O'N19] Chris O'Neill. Bowlbo: The Quest for Bing Bing, 2019. *https://www.youtube.com/watch?v=_NC4sU6jCKw* [Online; posted 25-March-2019]. (cit. on 1.1)

[Pap99a] Dave Pape. VPL DataSuit, October 1999. *https://commons.wikimedia.org/wiki/File:VPL_DataSuit_1.jpg*. (cit. on 1.9)

[Pap99b] Dave Pape. VPL Eyephone and Dataglove, October 1999. *https://commons.wikimedia.org/wiki/File:VPL_Eyephone_and_Dataglove.jpg*. (cit. on 1.9)

[Pap18a] Georgios Papaioannou. Ray Tracing, Lecture Slides, Athens University of Economics and Business, 2018. (cit. on 2.17, 2.18)

[Pap18b] Georgios Papaioannou. Stochastic Path Tracing, Lecture Slides, Athens University of Economics and Business, 2018. (cit. on 2.21)

[Pap18c] Georgios Papaioannou. Virtual and Mixed Reality, Lecture Slides, Athens University of Economics and Business, 2018. (cit. on 1.4)

[PBD+10] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29(4):66, 2010. (cit. on 3.1.2)

[Ped16] Paul Pedriana. Under the Hood of the Rift SDK Building for Touch, October 2016. *https://www.youtube.com/watch?v=eAl2l_1KfqQ* [Online; posted 12-October-2016]. (cit. on 2.32)

[Pix17] Pixel. Old leather sofa model, 2017. *https://hdrihaven.com/hdri/?c=night&h=shanghai_bund*(CC BY 4.0). (cit. on 4.3.2)

[PJH17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Monte carlo integration. In *Physically Based Rendering*. Elsevier, 2017. (cit. on 2.1.2)

[PV13] Georgios Papaioannou and Konstantinos Vardis. XEngine Documentation and API Reference Version 1.0, 2013. *http://graphics.cs.aueb.gr/graphics/downloads/xenginedoc/XEngine.html*. (cit. on 2.24)

[Rou12] Franz Roubaud. Panorama of Moscow battle in 1812, 1912. *https://commons.wikimedia.org/wiki/File:Battle_of_Borodino_part_of_panorama_by_Franz_Roubaud.jpg*. (cit. on 1.5)

[Sch94] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, August 1994. (cit. on 2.1.2)

[sdl01] Simple DirectMedia Layer - SDL version 2.0.9 (stable), 2001. *https://www.libsdl.org/download-2.0.php*. (cit. on 3.1.1)

[Sen18] Sensoryx AG. VRfree glove system, 2018. *https://www.indiegogo.com/projects/vrfree-glove-system#/*. (cit. on 1.13)

[Sha12] Kitsune Shakko. Battle of Borodino panorama by Shakko, 2012. *https://commons.wikimedia.org/wiki/File:Battle_of_Borodino_panorama_06_by_shakko.jpg*(CC BY-SA 3.0). (cit. on 1.5)

[Ska18] Skarredghost. Donald Greenberg: virtual reality is still a baby, we'll need 20 years for photorealism, November 2018. *https://skarredghost.com/2018/11/02/donald-greenberg-virtual-reality-is-still-a-baby-well-need-20-years-for-photorealism/*. (cit. on 1.3.2)

[Smi67] B. Smith. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, September 1967. (cit. on 2.1.2)

[Sob94] Ilya M Sobol. *A Primer for the Monte Carlo Method*. CRC Press, 1994. (cit. on 2.2.2)

[Squ16] Black Squirrel. Sega VR, June 2016. *https://segaretro.org/Sega_VR*. (cit. on 1.10)

[Sri17] Rob Srinivasiah. How to maximize AR and VR performance with advanced stereo rendering, November 2017. *https://blogs.unity3d.com/2017/11/21/how-to-maximize-ar-and-vr-performance-with-advanced-stereo-rendering/*. (cit. on 2.3.1)

[SSK+17] Christoph Schied, Marco Salvi, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, and Aaron Lefohn. Spatiotemporal variance-guided filtering. In *Proceedings of High Performance Graphics on - HPG 17*. ACM Press, 2017. (cit. on 2.2.4, 2.22, 4.3.2)

[Sta13] Sebastian Stabinger. Oculus Rift - Developer Version - Front, December 2013. *https://commons.wikimedia.org/wiki/File:Oculus_Rift_-_Developer_Version_-_Front.jpg*(CC BY 3.0). (cit. on 1.11)

[Sut65] Ivan E Sutherland. The Ultimate Display. Citeseer, 1965. (cit. on 1.2.2)

[tey14] teybeo. Sponza, 2014. *https://sketchfab.com/3d-models/sponza-2a75779d772c41f19c87188e367735ac*. (cit. on 4.1.1, 4.3.2)

[Tra06] Gilles Tran. Glasses, pitcher, ashtray and dice (POV-Ray), January 2006. *https://commons.wikimedia.org/wiki/File:Raytraced_image_of_several_glass_objects.png*. (cit. on 1.3)

[TS67] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces∗. *JournaloftheOpticalSocietyofAmerica*, 57(9) : 1105, *sep*1967. (*cit.on* 2.1.2)

[VR16] Oculus VR. Initialization and Sensor Enumeration. Developer Center — Documentation and SDKs, 2016. *https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-sensor/*. (cit. on 3.2)

[Vri16a] Joey de Vries. container2, 2016. *https://learnopengl.com/Lighting/Lighting-maps*(CC BY 4.0). (cit. on 2.1)

[Vri16b] Joey de Vries. container2_specular, 2016. *https://learnopengl.com/Lighting/Lighting-maps*(CC BY 4.0). (cit. on 2.1)

[Vri16c] Joey de Vries. coordinate_systems, 2016. *https://learnopengl.com/Getting-started/Coordinate-Systems*(CC BY 4.0). (cit. on 2.10)

[Vri16d] Joey de Vries. deferred_overview, 2016. *https://learnopengl.com/Advanced-Lighting/Deferred-Shading*(CC BY 4.0). (cit. on 2.14)

[Vri16e] Joey de Vries. lighting_maps_specular_color, 2016. *https://learnopengl.com/Lighting/Lighting-maps*(CC BY 4.0). (cit. on 2.1)

[Vri16f] Joey de Vries. materials, 2016. *https://learnopengl.com/PBR/Theory*(CC BY 4.0). (cit. on 2.2)

[Vri16g] Joey de Vries. pipeline, 2016. *https://learnopengl.com/Getting-started/Hello-Triangle*(CC BY 4.0). (cit. on 2.11)

[vrs17a] Applications Of Virtual Reality. *Virtual Reality Society*, 2017. *https://www.vrs.org.uk/virtual-reality-applications/*. (cit. on 1.2.3)

[vrs17b] History Of Virtual Reality. *Virtual Reality Society*, 2017. *https://www.vrs.org.uk/virtual-reality/history.html*. (cit. on 1.2.2)

[Whe38] Charles Wheatstone. Contributions to the physiology of vision.–part the first. on some remarkable, and hitherto unobserved, phenomena of binocular vision. *Philosophical transactions of the Royal Society of London*, 128:371–394, 1838. (cit. on 1.2.2)

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980. (cit. on 2.2)

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques - SIGGRAPH 78*. ACM Press, 1978. (cit. on 2.1.3)

[WMLT07a] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet Models for Refraction through Rough Surfaces. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques*. The Eurographics Association, 2007. (cit. on 2.1.2)

[WMLT07b] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces, 2007. (cit. on 2.1.2, 2.1.2)

[Zaa17] Greg Zaal. Tiber Island, 2017. *https://hdrihaven.com/hdri/?c=skies&h=tiber_island*. (cit. on 4.3.2)

[Zaa18] Greg Zaal. Shanghai Bund, April 2018. (cit. on 3.6)

[ZG17] Jian Zhang and Simon Green. Introducing Stereo Shading Reprojection for Unity, August 2017. *https://developer.oculus.com/blog/introducing-stereo-shading-reprojection-for-unity/*. (cit. on 2.23, 2.3.1, 2.25, 2.3.1)

[Zha18] Jian Zhang. Tech Note: Asymmetric Field of View FAQ, April 2018. *https://developer.oculus.com/blog/tech-note-asymmetric-field-of-view-faq/*. (cit. on 4.2.2)