UNIVERSIDAD DE INVESTIGACIÓN DE TECNOLOGÍA EXPERIMENTAL YACHAY

Escuela de Ciencias Matemáticas y Computacionales

TÍTULO: Real-Time Ray Tracing in OpenGL

Trabajo de integración curricular presentado como requisito para la obtención del título de Ingeniero en Tecnologías de la Información

Autor:

Mateo David Coello Andrade

Tutor:

Francesc Antón Castro, Ph.D.

Urcuquí, Noviembre 2024

Autoría

Yo, **Mateo David Coello Andrade**, con cédula de identidad 0107870768, declaro que las ideas, juicios, valoraciones, interpretaciones, consultas bibliográficas, definiciones y conceptualizaciones expuestas en el presente trabajo; así cómo, los procedimientos y herramientas utilizadas en la investigación, son de absoluta responsabilidad de el autor del trabajo de integración curricular. Asímismo, me acojo a los reglamentos internos de la Universidad de Investigación de Tecnología Experimental Yachay.

Urcuquí, Noviembre - 2024.

Firmado electrónicamente por:
MATEO DAVID COELLO
ANDRADE

Mateo David Coello Andrade
CI: 0107870768

Autorización de publicación

Yo, **Mateo David Coello Andrade**, con cédula de identidad 0107870768, cedo a la Universidad de Investigación de Tecnología Experimental Yachay, los derechos de publicación de la presente obra, sin que deba haber un reconocimiento económico por este concepto. Declaro además que el texto del presente trabajo de titulación no podrá ser cedido a ninguna empresa editorial para su publicación u otros fines, sin contar previamente con la autorización escrita de la Universidad.

Asímismo, autorizo a la Universidad que realice la digitalización y publicación de este trabajo de integración curricular en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación.

Urcuquí, Noviembre - 2024.

Firmado electrónicamente por:
MATEO DAVID COELLO
ANDRADE

Mateo David Coello Andrade
CI: 0107870768

Dedication

Endings are devastating, worrying, yet inevitable.
Fear is the one playing on stage,
seconds remain till the curtain falls again.
Logic aka consciousness is ready to present.
Although nervousness wants to participate,
happiness prevents him from getting on stage.
The curtain loosely falls and lights are right on.
The poetry of memories is announced,
declaimed by yours truly self-confidence.
Every moment is a different story,
each one having its photography, and
marked with its own time and dedicatory.
Thanks to every essence that was part of this story.

Mateo David Coello Andrade

Resumen

Los gráficos por computadora son una de las ramas más amplias de estudio dentro de las ciencias computacionales. Su objetivo principal es sintetizar una imagen de un modelo 2D o 3D, cuyo proceso se conoce formalmente como renderización. Uno de los métodos más importantes de renderización 3D es el trazado de rayos, que se distingue por su capacidad de simular el comportamiento de la luz de forma realista. Para lograrlo, se apoya principalmente en ramas como la radiometría y la geometría óptica, con el fin de modelar correctamente el comportamiento de la luz cuando interactúa con superficies. Asimismo, se toma en cuenta la teoría de microfacetas, la cual modela una superficie como una colección de pequeños espejos distribuidos aleatoriamente. La incorporación de estas nociones en los gráficos por computadora se conoce como renderización basada en la física. Al ser combinada con la técnica de trazado de rayos, se pueden simular de forma realista los distintos comportamientos y efectos de la luz, tales como la reflexión, refracción, sombras, sangrado de color, entre otros. Otro aspecto fundamental de los gráficos por computadora es su interactividad, lo cual se logra mediante una cámara capaz de moverse dentro de la escena, pudiendo observar distintas perspectivas mientras todo se renderiza en tiempo real. La renderización en tiempo real es posible gracias al uso de la unidad de procesamiento gráfico (GPU), la cual nos permite paralelizar la tarea de renderizado, generando varias imágenes por segundo. Este trabajo incorpora los conceptos mencionados para implementar un renderizador de trazado de rayos en tiempo real, haciendo uso de OpenGL, una API para el manejo de la GPU.

Keywords: Tiempo Real, Trazado de Rayos, Renderización Basada en la Física, Iluminación Global.

Abstract

Computer graphics are one of the broadest fields of study within computer science. Its main objective is to synthesize an image from a 2D or 3D model, a process formally known as rendering. One of the most important 3D rendering methods is ray tracing, which is distinguished by its ability to simulate the behavior of light realistically. To achieve this, it primarily relies on fields such as radiometry and optical geometry, in order to accurately model the behavior of light when it interacts with surfaces. Additionally, the microfacet theory is taken into account, which models a surface as a collection of small mirrors randomly distributed. The incorporation of these concepts into computer graphics is known as physically-based rendering. When combined with ray tracing, various light behaviors and effects, such as reflection, refraction, shadows, color bleeding, among others, can be realistically simulated. Another fundamental aspect of computer graphics is its interactivity, which is achieved through a camera capable of moving within the scene, allowing different perspectives to be observed while everything is rendered in real time. Real-time rendering is made possible by using the graphics processing unit (GPU), which allows us to parallelize the rendering task, generating several images per second. In conclusion, this work incorporates the aforementioned concepts to implement a real-time ray tracing rasterizer using OpenGL, an API for managing the GPU.

Keywords: Real-Time, Ray Tracing, Physically Based Rendering, Global Illumination.

Contents

Dedication iii

Resumen iv

Abstract v

Contents vi

List of Tables ix

List of Figures xi

1 Introduction 1

- 1.1 Background 1
- 1.2 Problem Statement 3
- 1.3 Objectives 6
  - 1.3.1 General Objective 6
  - 1.3.2 Specific Objectives 6

2 Theoretical Framework 7

- 2.1 The Pinhole Camera Model 7
- 2.2 Modification of the Pinhole Camera 8
- 2.3 The Behavior of Light 9
- 2.4 Perception of Color 10
- 2.5 Tracing Rays 12
- 2.6 Matrix Transformations 16

2.7 Homogeneous Coordinates and Projective Space 18
2.8 The View Frustum 20
2.9 Coordinate Systems 21

3 State of the Art **32**
3.1 Hybrid Ray Tracing Rendering 32
3.2 Acceleration Structures 34
3.3 Denoising Techniques 36
3.4 Summary 37

4 Methodology **39**
4.1 Implementation Details 39
    4.1.1 Shaders and the Rasterization Pipeline 40
    4.1.2 Ray Tracing Pipeline 43
4.2 Primitive Rendering 47
    4.2.1 Sphere Rendering 47
    4.2.2 Triangle Rendering 49
    4.2.3 Axis-Aligned Bounding Box/Ray Intersection 51
4.3 Intersection Acceleration 54
4.4 Bounding Volume Hierarchies 56
    4.4.1 BVH Construction 57
    4.4.2 Primitives Bounding Boxes 58
    4.4.3 Split Methods to Construct a BVH 58
4.5 The Global Illumination Problem 61
4.6 Radiometry 62
    4.6.1 Light Source Representation 68
    4.6.2 Light-Surface Interactions 69
    4.6.3 Reflectance and BRDF 70
    4.6.4 Diffuse and Specular Reflections 72
4.7 The Rendering Equation 73
4.8 Shading Models 74
    4.8.1 Phong Model 75

4.8.2 Blinn-Phong Model 77

4.9 Physically Based Models 78

4.9.1 The Fresnel Equations 78

4.9.2 Torrance-Sparrow Model 80

4.9.3 Cook-Torrance Model 82

4.10 Path Tracing 84

4.10.1 Indirect Illumination 86

4.11 Summary 91

5 Results and Discussion **92**

5.1 Traversal of Bounding Volume Hierarchies 92

5.2 Spheres Rendering Comparison 93

5.3 Analyzing Global Illumination Effects 99

5.3.1 Color Bleeding 101

5.3.2 Soft Shadows and Specular Reflections 102

5.3.3 Combination of Effects 103

5.3.4 Performance Analysis 105

5.4 Rendering a Complex Model 108

5.5 Limitations 111

6 Conclusions **115**

Bibliography **118**

Appendices **135**

A Main GitHub Repository 135

B Ray Traced Sphere 135

C Ray Traced Triangle 135

D BVH Construction 135

E Distance-Based Traversal 136

F Ray-Direction Sign-Based Traversal 137

G Science Fiction Building Model - Normal Map 138

List of Tables

2.1 Coordinate Spaces used in 3D Rendering [1, 2] 22

4.1 Implementation details of the renderer including general development aspects, platform support, and libraries. 41

4.2 Base reflectance F_0 values for different materials. Each value is expressed as a normalized RGB triplet in the 'Linear' column and its respective integer 8-bit representation in the 'Texture' column. Values with just one term indicate a triplet with the same value in all components. Moreover, only one value was picked for those materials whose reflectances are represented in ranges. All values were obtained from [2, 3]. 80

4.3 Alternative functions for computing the statistical distribution of microfacet normals D, and the geometric attenuation factor G. For the computation of D, the vector $\boldsymbol{m}$ represents the microfacets normals. When computing G, the vector $\boldsymbol{s}$ represents either $\boldsymbol{v}$ or $\boldsymbol{l}$ to account for the effect of masking or shadowing, respectively. The value of α is equal to the squared roughness. 84

5.1 Specifications of the system on which the tests were performed 92

5.2 Spheres Rendering Comparison - Testing Parameters 94

5.3 Average FPS comparison across five runs for the sphere scenarios rendered in Test 1. 96

5.4 Average FPS comparison across five runs for the sphere scenarios rendered in Test 2. 98

5.5 Global Illumination Comparison - Testing Parameters 100

5.6 Average FPS comparison of the global illumination scenes for the five runs. 106

5.7 Complex Model Rendering Parameters 109
5.8 Average FPS comparison of the science fiction building model renders across five runs. 111

List of Figures

2.1 Pinhole camera model illustrating how incoming light rays produce the final inverted image. 7

2.2 Vertical field of view representation. 8

2.3 Modification of the pinhole camera model for computer graphics that exhibits the eye of the viewport as a grid of pixels. 9

2.4 Visible Light Spectrum and Cone Response to Light Wavelengths. The visible light spectrum was adapted from [4]. The cone response was adapted from [5], the real cone curves are presented in the original paper. 11

2.5 The RGB color model representation. 12

2.6 Parametric line form by the points $\mathbf{a} = (0, 2, 3)$ and $\mathbf{b} = (3, 2, 0)$. 13

2.7 Forward ray tracing variant that shows some photons will never reach the viewport. The Cornell box render belongs to [6]. 14

2.8 Backward ray tracing variant that illustrates the different types of rays based on Glassner's [7] description. The Cornell box render belongs to [6]. 15

2.9 Result of applying a translation vector $v = (4, 1, -5)$ to a cube. 18

2.10 View frustum volume comprised of a near plane (green) and a far plane (gray), with the projection plane (blue) in between. Each plane is perpendicular to the viewing direction of the camera. Adapted from [8]. 21

2.11 Model matrix transformation of a cube into world space comprised by: (1) scale down transformation, (2) rotation, and (3) translation. 22

2.12 Camera's coordinate axes formed by the right, up, and forward vectors according to its angle of view. 23

2.13 Transformation of one coordinate system to another given by the translation vector $\boldsymbol{t} = (3, 1, 0)$. The point $\mathbf{p} = (0, 0, -1)$ in the world coordinate system has the representation $\mathbf{p} = (0, 0, -1) - (3, 1, 0) = (-3. - 1, -1)$ in the view coordinate system. 24
2.14 Rotation of the camera coordinate system by an angle of 45° respect to the y world space axis, followed by a rotation around the x-axis of 35.26°. . . . 24
2.15 Top and side views of the frustum according to the view space coordinates of the camera. Adapted from [9]. 26
2.16 Mapping x_p to x_n 28
2.17 Mapping y_p to y_n 28
2.18 Coordinate spaces based on the ray tracing rendering process. 31

4.1 Rasterization pipeline for rendering a simple triangle. Because the coordinates are within the viewport the clipping stage is not illustrated. 43
4.2 Workgroup hierarchy used to instantiate compute shaders and parallelize a task. 44
4.3 Ray tracing pipeline for rendering a scene. The compute shader pipeline is used to render a ray-traced frame. The frame is stored as an image and bound to a texture. Finally, using the rasterization pipeline, the texture containing the rendered image is mapped to a triangle that covers the entire viewport and then displayed on the screen 46
4.4 Ray-sphere intersection illustrating no solution, one solution, or two solutions. 48
4.5 Visual representation of Möller and Trumbore algorithm for ray-triangle intersection. Adapted from [10]. 51
4.6 Ray-box intersection presenting the two intersections with the AABB. The AABB is represented by its min_B and max_B (B stands for bound) coordinates following a left-hand orientation of the coordinate axes. 52
4.7 Visual representation of the slab method to determine the intersection of a ray with a 2D bounding box (BB). Three cases are presented, a) the ray hits a corner of the BB b) a ray intersects the BB at two points, and c) the ray does not intersect the BB. 54

4.8 Bounding Volume Hierarchy for a scene comprised of two spheres and three triangles (a) For each primitive, a bounding box is computed and then these are added together based on proximity forming a larger bounding volume. The largest bounding volume encloses all other bounding volumes of the scene. The example also shows the intersection of two bounding volumes enclosing the purple and cyan triangles. (b) The bounding volume tree hierarchy with the root node representing the largest bounding volume, followed by the internal nodes, and finally the leaf nodes containing the primitives. . 57
4.9 Representation of the differential solid angle for a sphere of radius r. 63
4.10 Radiant intensity (I) of a light source. 65
4.11 Irradiance incident to an infinitesimal patch S with area dA. 66
4.12 Radiance representation in terms of radiant flux defined as the radiant power per unit foreshortened surface area per unit solid angle along the direction of reflection $\boldsymbol{d}_o$ 68
4.13 Bidirectional Reflectance Distribution Function (BRDF) formed by the zenith angles, θ_i and θ_o, and the azimuth angles, θ_i and θ_o. A BRDF f_r is said to be 4-dimensional, but can also be represented in terms of the direction vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_o$, such that $fr(\theta_i, \theta_o, \theta_i, \theta_0) = fr(\boldsymbol{d}_o, \boldsymbol{d}_i)$[11, 12]. 71
4.14 Diffuse, specular, and glossy reflections. 72
4.15 Blinn-Phong model where the half vector $\boldsymbol{h}$ prevents from calculating the reflection vector $\boldsymbol{r}$. 77
4.16 Microfacet representation of a surface based on the Torrance-Sparrow model. The incident radiance direction $\boldsymbol{d}_i$ is represented as $\boldsymbol{l}$. The masking effect is shown on the left side while the shadowing effect is on the right side, notice that $\boldsymbol{l}$ and $\boldsymbol{v}$ are interchanged but $\boldsymbol{h}$ remain the same. 82

5.1 Representation of the four scenarios rendered for Test 1 94
5.2 Average frame rendering time comparison for each scenario in the absence of an acceleration structure (NA), using a BVH with the distance-based method (DB), and employing the ray-direction sing-based traversal (RDSB). A logarithmic scale is used to represent the results of the five runs. 95

5.3 Representation of rendered scenarios for Test 2 97
5.4 Average frame rendering time comparison for Test 2 using the DB and RDSB traversal algorithms. The results represent the average across the five runs. 98
5.5 Differences between metallic, specular, and roughness properties in surfaces. 100
5.6 Color bleeding effect inside the Cornell box. 101
5.7 Soft shadows and reflections caused by a yellow diffuse box and a gray metallic box placed on the left and right, respectively, inside the Cornell box. 102
5.8 Combination of color bleeding, soft shadows, specular reflections, and glossiness effects. 104
5.9 Average frame rendering time comparison of the different scenarios of global illumination. The results represent the average of the five executions for the three Cornell box scenes presented earlier. 106
5.10 Average frame rendering time comparison of the science fiction building model across the five runs. 110
5.11 Triangle shown on grids of three different resolutions. On the discrete grid, the smooth geometry turns into stair steps denominated jaggies. No matter how high the resolution, the jaggies will not disappear, but will only become smaller. 112
5.12 Renders of the science fiction building model. 114

Chapter 1

Introduction

1.1 Background

Computer graphics comprise an extensive branch of study focused on developing algorithms, methods, and techniques for computer-based image synthesis. One of the major research areas is rendering, which describes the process of generating an image from a set of data. This data could represent anything from a time series of air pollution levels to a single frame in a 3D animated movie, or the layout of a user interface for a web browser. As such, graphics can be categorized into 2D and 3D according to the data they depict. In the realm of 3D graphics, rendering is further divided into two main approaches: photorealistic and non-photorealistic [3, 13]. Photorealistic rendering aims to recreate reality as closely as possible by considering all visual aspects of light and the objects in the scene imitating the photography process. In turn, non-photorealistic rendering takes a more creative approach, using multiple artistic styles to produce simplified, stylized, or illustrative representations of 3D models.

Although non-photorealistic rendering presents vast opportunities for exploration, photorealism is widely valued across numerous industries [3, 13]. In film, animation, and visual effects, photorealistic rendering enables balancing the lighting of virtual elements added during production [14]. Architectural visualization also leverages photorealism to recreate realistic indoor and outdoor lighting, enhancing the visual aspect of architectural designs [13]. Product advertising benefits significantly as well, allowing for lifelike renderings of future products, while reducing costs by providing a controlled environment where materials, lighting, and setups can be easily adjusted [13]. Photorealism in flight and driving simulations is essential to ensure proper atmospheric conditions or accurate street lighting [14]. The video game industry has also evolved largely, where photorealistic rendering has become crucial to create increasingly immersive 3D worlds [3, 13, 14].

Accurately simulating light and its different phenomena is central to photorealistic rendering, as it is essential for achieving a convincing level of realism. Light itself is particularly intriguing due to its wave-particle duality, meaning it can behave both as waves and as particles [15]. This complexity becomes even more evident when considering light interactions (absorption, reflection, or transmission) with objects in an environment. Such interactions determine how we can distinguish objects by their colors and the characteristics of their constituent materials. Additionally, these interactions give rise to shadows, reflections, and refractions, which produce various effects on surfaces as light bounces or bends. With this in mind, the question arises: How can light and its interactions be simulated?

A widely discussed concept in modern computer graphics and photorealistic rendering is ray tracing. This technique, which has been studied for several decades, offers an alternative solution to the rendering problem. Its origins can be traced back to 1967 when Appel [16] introduced a method to determine the visibility of the contour lines that define a surface. Appel's approach involved shooting rays through the facets of a 3D object to identify which ones were visible from a given viewpoint. This foundational idea paved the way for more advanced rendering techniques, enabling the accurate depiction of 3D objects from different perspectives.

In 1971, Goldstein and Nagel [17] proposed a ray tracing method that introduced a new way to render 2D images of 3D scenes using three essential components: a camera, a light source, and the objects in the scene. This approach mirrors how real cameras operate, where light rays bouncing off of objects pass through the lens and are captured on photographic film forming the image. The primary difference lies in the direction of the rays: in ray tracing, rays are cast from the camera's viewpoint into the 3D scene to identify objects within the line of sight. The resulting images displayed the facets of the three-dimensional objects shaded according to the camera's perspective and the light source's position. This advancement laid the groundwork for further research in ray tracing and photorealistic rendering, where the light physical principles became crucial in achieving realistic results.

Today, ray tracing is known as a solution for photorealistic rendering that generates realistic representations of shadows, reflections, refractions, and direct/indirect lighting, among other phenomena [18, 19]. These effects collectively create captivating photorealistic visuals, where its high level of detail enhances the meaning and realism in simulations, animation frames, advertisements, product designs, and architectural models, to name a few use cases. Recently, ray tracing has also become integral to augmented and virtual reality [20, 21]. The inclusion of photorealism in augmented and virtual devices ensures realistic visuals of the 3D environments that are created with these technologies. Photorealism in these technologies allows for realistic visualizations of 3D environments, benefiting various fields, such as healthcare, where VR simulations of surgeries can support training and assist during procedures [22, 23].

Other sectors leveraging photorealism with augmented and virtual reality include automotive [24], architecture and construction [25, 26], and industrial design [27]. Ray tracing's impact across various applications, technologies, and industries is critical, adding depth and realism to visual representations, which enhances the clarity and meaning of the visuals. An additional key to achieving photorealism is physically based rendering (PBR), a field in computer graphics dedicated to simulating the interaction of light with materials in accordance with physical laws [3, 11]. Combined with ray tracing, PBR more accurately models how light reflects off surfaces, capturing details like reflections, refractions, roughness, and color in a realistic way [11]. Consequently, advancements in photorealistic rendering through PBR and ray tracing are driving forward innovations in multiple fields, setting new standards for any visual application where photorealism is essential.

1.2 Problem Statement

Interactive computer graphics are an essential part of any visual application or system, enabling users to interact in real-time with graphical elements and observe responses to their inputs. Rendering is closely linked to interactivity, as any adjustments in the 3D scene require re-rendering to display the changes. These changes might include adjusting light conditions, shifting the camera, repositioning 3D models, or replacing the object colors and materials. Consequently, the generation time of the newer image depends largely on the rendering algorithm, scene complexity, and hardware specifications. Longer rendering times directly impact system interactivity, reducing responsiveness.

Real-time rendering aims to generate highly interactive graphics by rapidly synthesizing images of a 3D scene [3, 12]. During interaction, viewers respond to the current image, and their input influences the next generated image. This rapid rendering-interaction cycle creates a dynamic experience, enhancing viewer immersion. The rate at which images are generated is measured in frames per second (FPS), with 1 FPS indicating minimal interaction. From 6 FPS onwards, the viewer's sense of interactivity becomes noticeable [3]. Movies, for example, are shot at 24 FPS but use a shutter system to display each frame multiple times, reducing flickering [3, 13]. Video games generally target a frame rate of 30 FPS, but as response latency becomes critical, frame rates may exceed 60 FPS [28]. In virtual reality, the IEEE 3079 standard recommends at least 90 FPS for interactive content in head-mounted displays [29].

The benefits of real-time rendering are extensive: it enables immediate layout and lighting adjustments, rapid prototyping, adaptable pre-visualization for film and television, and virtual environment creation [13]. Additionally, it reduces lengthy rendering times and enhances the realism of interactive architectural visualizations, immersive gaming, and virtual reality experiences. As such, the combination of real-time and photorealistic rendering allows for highly interactive and realistic graphics across diverse applications. This level of performance is made possible through graphics acceleration hardware. To illustrate, a common introductory resource for ray tracing is *"Ray Tracing in One Weekend"* by Peter Shirley [30], where a CPU-based ray-tracing renderer is built from scratch. Upon completion of the rendering process, a single frame or image of the 3D scene is generated. However, as the scene complexity increases, rendering a single frame can take seconds, minutes or hours to complete.

One approach to reducing rendering times is to parallelize the ray-tracing code to leverage the multiple cores of modern CPUs. While this can lower rendering times, it remains insufficient for achieving interactive, real-time graphics. Early solutions attempted to distribute the ray-tracing workload across multiple processors or computer clusters, using specialized communication protocols to ensure efficient data distribution and task parallelization [31–33]. However, these solutions had limitations due to high implementation costs and limited consumer accessibility, making them less feasible for widespread use.

The paradigm shifted significantly with the introduction of graphics processing units (GPUs). Research in computer graphics advanced rapidly, with real-time and photorealistic rendering seeing substantial progress. The thousands of cores in a GPU, along its SIMD (Single Instruction, Multiple Data) architecture, can handle the massive number of computations required by ray tracing algorithms, drastically reducing rendering times [34–36]. With advancements in GPU architectures and technologies, such as NVIDIA's RTX series, real-time ray tracing has become possible even in complex applications like video games, virtual reality, high-fidelity simulations, film and television, design, among others [36, 37]. In this sense, GPUs have driven a complete revolution in computer generated graphics, improving photorealism and real-time rendering techniques.

The primary APIs for interacting with GPUs include OpenGL [38], DirectX [39], Vulkan [40], Metal [41], and CUDA [42]. Among these, OpenGL and DirectX are the most widely recognized due to their long-standing presence in the computer graphics industry. However, DirectX is a proprietary Microsoft API, restricting its use to Microsoft platforms only. Similarly, Metal and CUDA are proprietary APIs developed by Apple and Nvidia, respectively. This leaves OpenGL and Vulkan as the main cross-platform, open-source solutions. Vulkan, developed by the Khronos Group, is the successor to OpenGL and was designed to address the limitations of OpenGL. As a lower-level API, Vulkan offers greater control over GPU processes, leading to improved performance [43].

In summary, key aspects necessary for understanding interactive computer graphics have been covered. But what is their relevance in the context of ray tracing? One key objective in ray tracing is to generate a large number of frames that accurately simulate realistic lighting effects in a 3D environment, ensuring smooth interactions despite camera movement. In the current context, one can assume that rendering frames from a ray tracing renderer using the GPU will suffice, but this is not entirely optimal. Moreover, mastering and implementing all the state-of-the-art techniques for ray tracing requires significant time and effort. Therefore, this work focuses on the core principles necessary to develop a real-time ray-tracing renderer.

1.3 Objectives

1.3.1 General Objective

The main objective of this work is to implement a real-time ray-tracing renderer that handles realistic lighting in a scene. In doing so, it is expected to provide an interactive system that allows the visualization of different lighting effects independently of the camera's position and perspective of the 3D world.

1.3.2 Specific Objectives

- Handle common ray-tracing primitives such as spheres and triangles. Provide support for loading triangular meshes of 3D models in the wavefront.obj format.
- Optimize ray-primitive intersection operations using an acceleration structure.
- Incorporate physically based rendering (PBR) techniques to approximate the physical appearance of object materials under the incidence of light to their real-world counterparts.
- Provide a solution to the global illumination problem to render the scene with realistic lighting by means of path tracing.

Chapter 2

Theoretical Framework

2.1 The Pinhole Camera Model

To understand the ray tracing components, we should take a step back into what is known as the pinhole camera model. Angel and Shreiner [1] describe this type of camera model, illustrated in Figure 2.1, as an enclosed box with a photographic film at one side of the box and a small pinhole at the center of the opposite side. The incoming light rays that pass through the pinhole and hit the film will generate the image that captures the scene but rotated by 180°. As such, it is observed that point **f** is the projection of point **p** onto the photographic film.

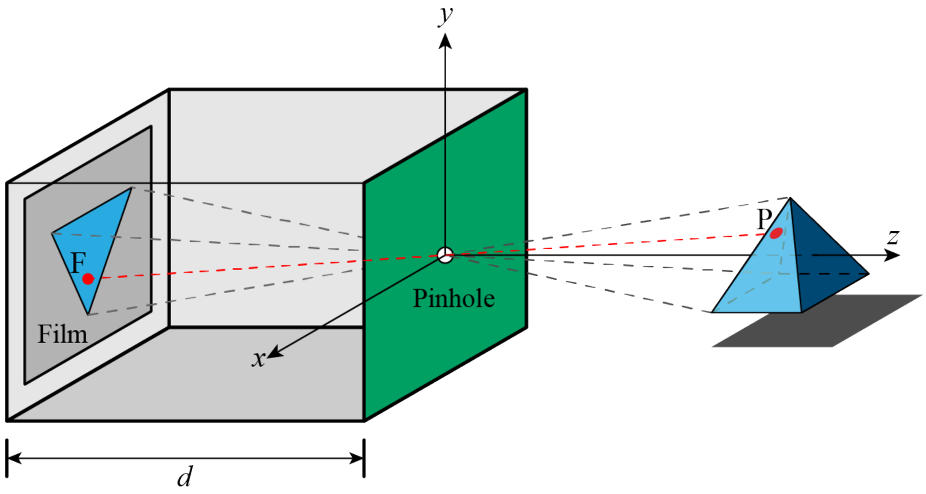


Figure 2.1: Pinhole camera model illustrating how incoming light rays produce the final inverted image.

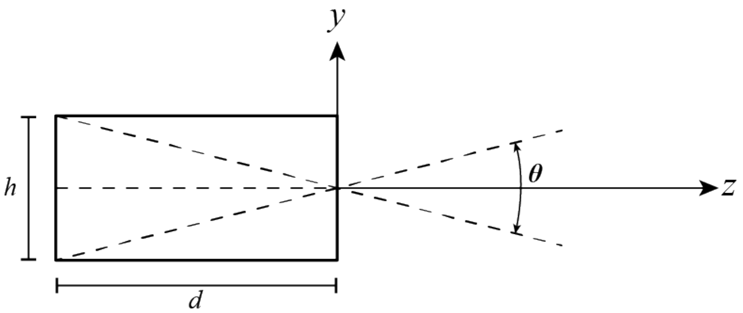


Figure 2.2: Vertical field of view representation.

Considering that the angles are congruent, the coordinates of the projected point $\mathbf{f}$ are calculated as follows. Let $\mathbf{f} = (\mathbf{f}_x, \mathbf{f}_y, -d)$ and $\mathbf{p} = (\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z)$, then:

$$\frac{\mathbf{f}_x}{-d} = \frac{\mathbf{p}_x}{\mathbf{p}_z} \rightarrow \mathbf{f}_x = -\frac{\mathbf{p}_x \cdot d}{\mathbf{p}_z}$$

$$\frac{\mathbf{f}_y}{-d} = \frac{\mathbf{p}_y}{\mathbf{p}_z} \rightarrow \mathbf{f}_y = -\frac{\mathbf{p}_y \cdot d}{\mathbf{p}_z}$$

The pinhole camera model also introduces the concept of the field of view, which is the largest possible angle the camera can observe from the world. Although it can be measured horizontally or diagonally, it is often calculated vertically. The vertical field of view (FOVy) is illustrated in Figure 2.2, and is given by:

$$\text{FOV}y\text{:}\ \theta = 2 \cdot tan^{-1}\left(\frac{h/2}{d}\right)$$

2.2 Modification of the Pinhole Camera

Despite the pinhole camera model being an interesting model, it requires some adjustments to become a suitable model for ray-traced rendering. Glassner [7] explains these modifications in detail, the new model is illustrated in Figure 2.3. First, the film plane is moved in front of the pinhole and receives the name of the *viewport*. The second distinction is that the pinhole will now be renamed to the *eye*. The eye represents the camera in the 3D scene with its coordinate system depending on its position and viewing direction.

Another concept introduced is that of a *pixel*. An excellent article to get more insight into this concept is the one by Smith [44]. Simply put, a pixel could be depicted as a small

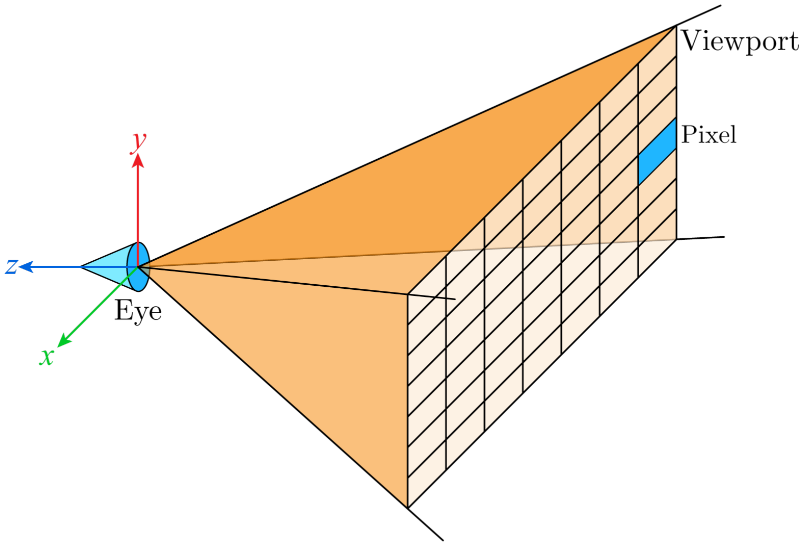


Figure 2.3: Modification of the pinhole camera model for computer graphics that exhibits the eye of the viewport as a grid of pixels.

rectangular window in the viewport. In turn, the viewport represents a discretization of the continuous space of photographic film as a grid of pixels. Therefore, if the pixel acts like a small window into the scene, what would be the best color to represent the objects visible through it?

A suitable solution is associating a small region of photographic film with a pixel. Once all the light rays have covered the pinhole and struck the small area of the film, the film will have accumulated enough rays to display a portion of the visible image. Consequently, an acceptable approximation for representing with a single color a pixel in the viewport will be to average the different colors of the light rays that hit it. Nonetheless, what are light rays and how can their colors be combined?

2.3 The Behavior of Light

Previously, it was mentioned that light could behave as both a wave and a particle. Young et al. [45] presented an overview of this topic. Energy transmitted by light waves is condensed into packets denominated photons. As such, although a wave model adequately describes the propagation of light, a particle behavior correctly explains the emission and absorption of light, and bodies whose particles are in thermal motion generate electromagnetic radiation, specifically thermal radiation. When particles have reached high enough temperatures, a body becomes luminescent and emits light.

Visible light can be explained by considering the wave model once again. Electromagnetic waves are propagated at a given frequency related directly to their wavelength. The visible light wavelengths range from about 380 to 750nm. During light propagation, a wavefront represents the instant when all wave points are at the same phase (position of the wave cycle relative to the origin). Rather than specifying the direction of light propagation with wavefronts, a ray is used. A ray represents no other than the path followed by photons, or a line perpendicular to the wavefronts of a wave.

2.4 Perception of Color

From the above, the following statements are drawn. Light sources and their interactions with an environment cause objects to be visible. The energy emitted by light sources travels in the form of photons. However, the frequency at which light waves propagate determines the emitted color. This brings us to the question of how we can perceive color. Kalloniatis and Luu [46] along with Land [47] provide a good explanation of the subject, an overview is given below.

When light rays reach the eye's retina as photons, these are processed by two photoreceptor cells known as rods and cones. Rods are sensitive to dim light allowing visibility under low light conditions. In contrast, cones are specialized for color discrimination in bright light conditions. Cones are divided into three categories based on their wavelength sensitivity to cover all the visible color spectrum of light. S-cones are sensitive to blue light and have a sensitivity peak at 440nm. M-cones are sensible to green colors peaking at a frequency of 535nm. Finally, L-cones are sensitive to the higher frequencies of light radiation (red colors) with their sensitivity peak at 565nm.

The capacity of cones to perceive different light wavelengths enables the so-called trichromatic human color vision [46]. The response of cones to light wavelengths together with the visible light spectrum is illustrated in Figure 2.4. Therefore, the combination of several wavelengths allows us to perceive different colors such as yellow, which is the response of M-cones and L-cones to wavelengths around 570nm and 585nm, or cyan which triggers the response of S-cones and M-cones to wavelengths between 580 and 590nm.

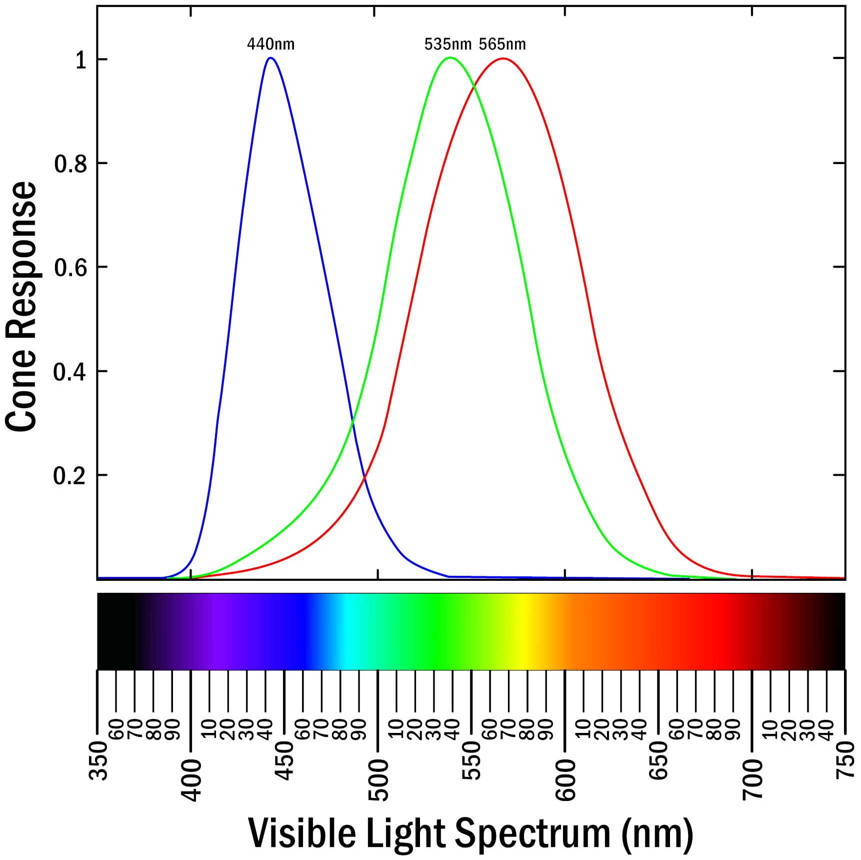


Figure 2.4: Visible Light Spectrum and Cone Response to Light Wavelengths. The visible light spectrum was adapted from [4]. The cone response was adapted from [5], the real cone curves are presented in the original paper.

However, the physical theory of color representation and reproduction is complex [48]. In consequence, many models have been created to give a proper representation of a color.

Within computer graphics applications, the model used is known as the RGB color model [49]. This model defines a color using a triple whose components are the three additive colors red, green, and blue, thus its name. In doing so, the RGB color model reproduces humans' trichromatic color vision [50]. The model is formally represented as a cube in 3D space (see Figure 2.5) with normalized red $(1, 0, 0)$, green $(0, 1, 0)$, and blue $(0, 1, 0)$ colors in the corners next to the origin corner. The origin corner represents the lack of color or black $(0, 0, 0)$, whereas white $(1, 1, 1)$ in contrast, is the complete contribution of the three. Other colors are obtained by combining red, green, and blue in different quantities. As an example, this shade of orange 🟧 is given by $(1.0, 0.8, 0.0)$ or this purple variant 🟪 is represented as $(0.66, 0.5, 1.0)$.

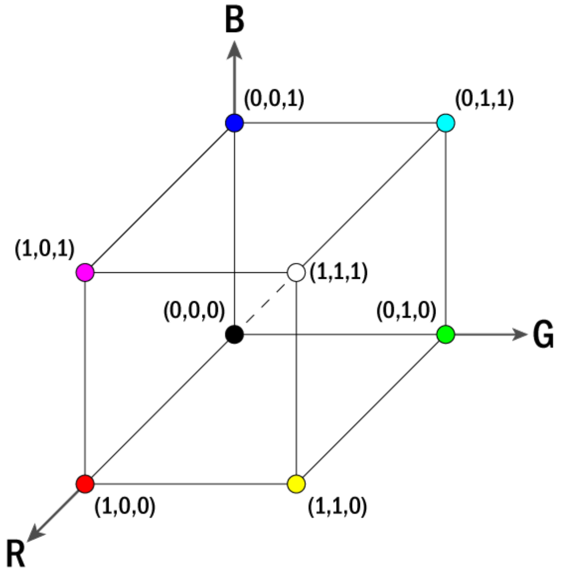


Figure 2.5: The RGB color model representation.

2.5 Tracing Rays

The concepts reviewed until now allow us to understand the physical behavior of light. Its purpose is fundamental in ray tracing to implement or approximate light concepts or their interactions. For instance, a proper mathematical definition could now be given to describe a light ray. Based on the particle model of light, Shirley et al. [51] define a ray as a line interval in 3D space. The parametric form of a three-dimensional line is given by

$$\mathbf{p}(t) = (1-t)\mathbf{a} + t\mathbf{b}, \quad \forall t \in \mathbb{R}$$

and represents a weighted average of points $\mathbf{a} = (\mathbf{a}_x, \mathbf{a}_y, \mathbf{a}_b)$ and $\mathbf{b} = (\mathbf{b}_x, \mathbf{b}_y, \mathbf{b}_z)$. In programming terms, it represents a function that receives as a parameter t and returns a point $\mathbf{p}$ along the line formed by the points $\mathbf{a}$ and $\mathbf{b}$ as illustrated in Figure 2.6. An alternative representation is to use a point and a vector. Such variation, allows us to define the origin of a ray as a point $\mathbf{o} = (\mathbf{o}_x, \mathbf{o}_y, \mathbf{o}_z)$, where $\mathbf{o} = \mathbf{a}$, and a normalized vector $\boldsymbol{d} = \dfrac{\mathbf{b}-\mathbf{a}}{||\mathbf{b}-\mathbf{a}||}$ as the direction of our ray. Therefore, a ray could be expressed as:

$$\mathbf{p}(t) = \mathbf{o} + t\boldsymbol{d}, \quad \forall t \in \mathbb{R}$$

Finally, for ray tracing a ray is also constrained by an interval $t \in [t_{min}, t_{max}]$ that defines a range for the intersections that are valuable during the rendering process. In other words, if a ray intersects an object at a point $\mathbf{p}(t)$ with $t \notin [t_{min}, t_{max}]$ then such object will not be rendered or its interaction will not be accounted for.

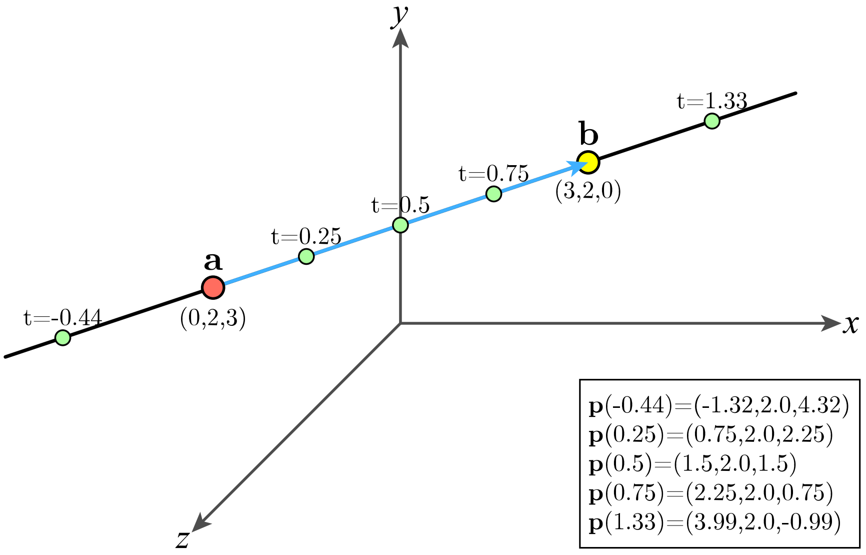


Figure 2.6: Parametric line form by the points $\mathbf{a} = (0, 2, 3)$ and $\mathbf{b} = (3, 2, 0)$.

Earlier it was discussed that the ray-tracing rendering process involves averaging all the light rays' colors intersecting a pixel. This poses a new problem in the rendering process which requires discovering the rays that intersect the viewport pixels. The solutions that were initially proposed are known as Forward and Backward Ray Tracing [7].

The forward variant follows the path of photons originating from the light source. Consider the example in Figure 2.7, where three photons are emitted. The first photon hits the red wall and bounces toward the metal sphere, but on its last bounce, it is lost in the scene. The second photon hits the gray back wall but unexpectedly bounces off and is also lost. The third photon is emitted, reaching the gray back wall first, bouncing into the gray floor, and when reflected off gets to intersect a pixel at the viewport. The photon paths presented raise a question: what will happen if none of them reach a pixel?

The previous example demonstrates the main drawback of forward ray tracing. Because photons can take infinite paths, and many will never reach the viewport. In consequence, the rendered image may present multiple missing pixels. One solution would be to increase the number of photon samples (at the cost of rendering time) until the user considers it

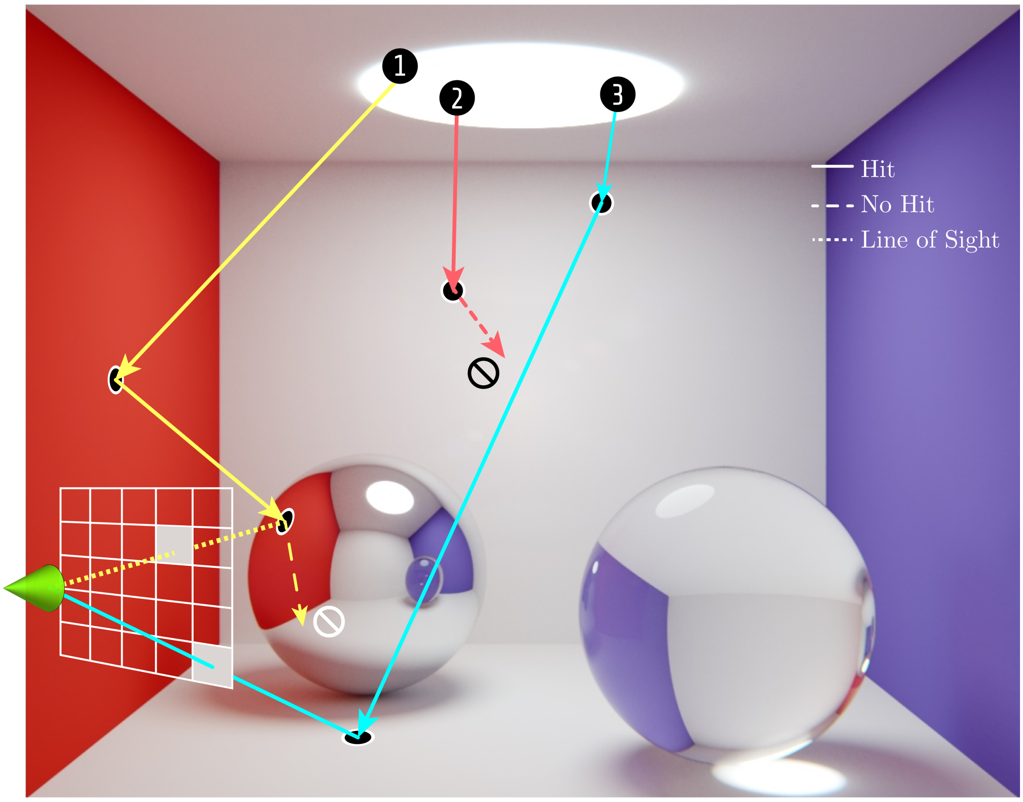


Figure 2.7: Forward ray tracing variant that shows some photons will never reach the viewport. The Cornell box render belongs to [6].

an acceptable image. These issues are addressed by approaching the rendering process from an inverse perspective. Instead of following the photons since they originated from the light source, a better solution is to backtrack them as if they had already reached the viewport. The previous concept gives place to what is known as backward ray tracing.

Backward ray tracing involves sending rays from the camera's eye into each pixel. The first object a ray hits after intersecting a pixel is interpreted as one stop in the photon's path as if it came from a light source. This guarantees that each pixel will contain information about the scene based on the camera's perspective. Furthermore, the backward variant makes four distinctions of light rays to render a scene. Glassner [7] presents them as eye rays, illumination rays, reflection rays, and transparency rays. Eye rays carry the light directly from the light's source to the camera's eye whereas illumination rays transport light to an object's surface. In contrast, reflection rays transport the light bouncing off a surface. Lastly, transparency rays transmit the light of those rays that have refracted from a surface.

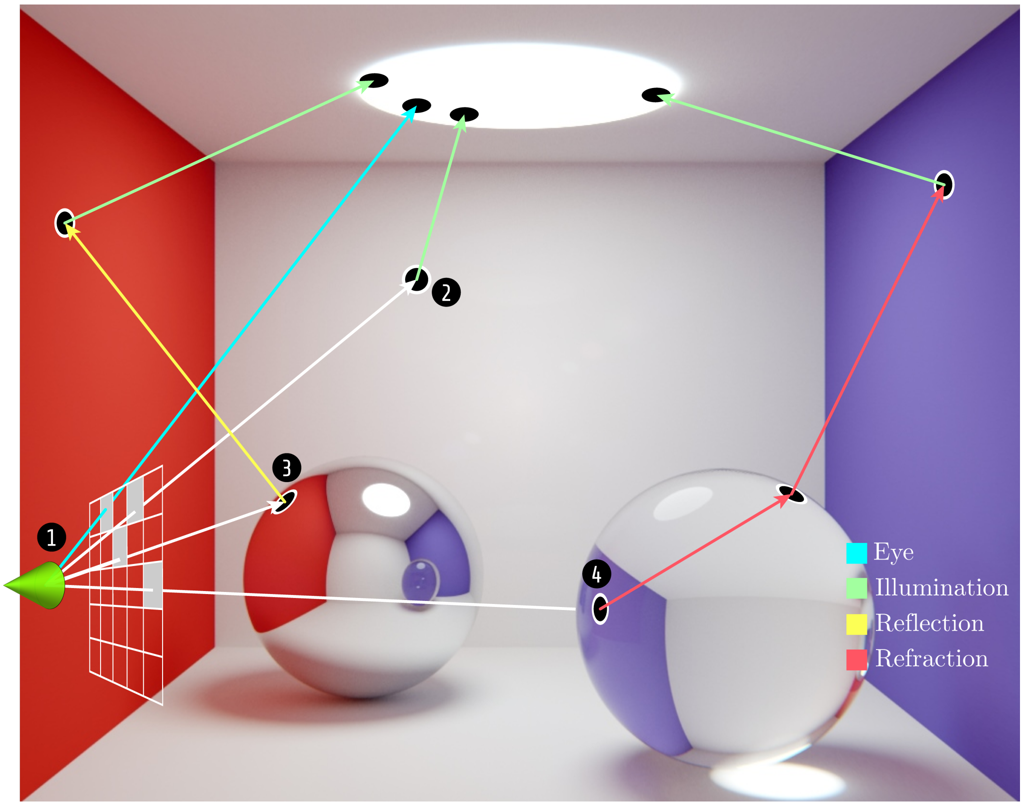


Figure 2.8: Backward ray tracing variant that illustrates the different types of rays based on Glassner's [7] description. The Cornell box render belongs to [6].

Figure 2.8 presents each type of ray described. The first ray corresponds to an eye ray since it is incident directly on the ceiling light. After hitting the wall, the second ray bounces directly off the light source, thus becoming an illuminating ray. The third ray hits the metal sphere and is reflected by the red wall. In turn, the red wall receives the light from an illuminating ray. Once refracted, the fourth ray enters the glass sphere and is refracted again after hitting another point inside it, to reach the blue wall. In turn, the hit point on the blue wall receives its light from an illumination ray.

It must be emphasized that the mathematical definition of any ray described is the same, however, their direction is modified depending on the surface they interact with. In other words, an object's material determines the propagation of incident light rays.

2.6 Matrix Transformations

So far, aspects of the functionality of a ray-tracing camera model have been reviewed. However, some mathematical tools are required to implement this camera model. Such tools are known as matrix transformations which alter the geometry and perspective of objects. The transformations required for our renderer are presented below:

- **Scaling** allows changing an object's size in each dimension. For example, to scale a vector it is multiplied by a scaling matrix as follows:

$$\begin{array}{ccccc} \text{Scaling Matrix} & & \boldsymbol{v} & & \text{Scaled } \boldsymbol{v} \\ \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} & = & \begin{bmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{bmatrix} \end{array}$$

- **Translation** involves changing an object's position on any axis. For instance, to translate a vector it is multiplied by a translation matrix, as shown below:

$$\begin{array}{ccccc} \text{Translation Matrix} & & \boldsymbol{v} & & \text{Translated } \boldsymbol{v} \\ \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} & = & \begin{bmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{bmatrix} \end{array}$$

- **Rotation** alters the current perspective of an object by rotating it against a specific axis. There are three basic rotation matrices to rotate a vector by an angle θ around a specific axis often abbreviated as $R_x(\theta)$, $R_y(\theta)$, and $R_z(\theta)$. These however are defined based on the orientation of the axes in three-dimensional space. Therefore, following a right-hand orientation, the rotation of a vector by any of these matrices is defined as follows:

$$\begin{array}{ccccc} R_x(\theta) & & \boldsymbol{v} & & \boldsymbol{v}\text{ rotated around X axis} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} & = & \begin{bmatrix} x \\ \cos\theta\cdot y - \sin\theta\cdot z \\ \sin\theta\cdot y + \cos\theta\cdot z \\ 1 \end{bmatrix} \end{array}$$

$$\begin{array}{ccccc} R_y(\theta) & & \boldsymbol{v} & & \boldsymbol{v}\text{ rotated around Y axis} \\ \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} & = & \begin{bmatrix} \cos\theta\cdot x + \sin\theta\cdot z \\ y \\ -\sin\theta\cdot x + \cos\theta\cdot z \\ 1 \end{bmatrix} \end{array}$$

$$\begin{array}{ccccc} R_z(\theta) & & \boldsymbol{v} & & \boldsymbol{v}\text{ rotated around Z axis} \\ \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} & = & \begin{bmatrix} \cos\theta\cdot x - \sin\theta\cdot y \\ \sin\theta\cdot x + \cos\theta\cdot y \\ z \\ 1 \end{bmatrix} \end{array}$$

All of these transformation matrices could be chained together by matrix multiplication. Careful though, as matrix multiplication is not commutative which changes the final transformation of our object depending on the order taken. A common rule of thumb is to first scale (S), then rotate (R), and finally translate (T) an object, where the order of transformations is applied from right to left ($T \times R \times S \times object$).

$$\begin{array}{cccccccc} \text{Translation Matrix} & & R_x(\pi/2) & & \text{Scale z-axis} & & \boldsymbol{v} & \\ \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \cdot & \begin{bmatrix} 2 \\ 3 \\ -1 \\ 1 \end{bmatrix} & = \end{array}$$

$$\overset{\text{Translation Matrix}}{\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \cdot \overset{R_x(\pi/2)}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \cdot \overset{\boldsymbol{v}_1}{\begin{bmatrix} 2 \\ 3 \\ -2 \\ 1 \end{bmatrix}} = \overset{\text{Translation Matrix}}{\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \cdot \overset{\boldsymbol{v}_2}{\begin{bmatrix} 2 \\ 2 \\ 3 \\ 1 \end{bmatrix}} = \overset{\boldsymbol{v}_3}{\begin{bmatrix} 4 \\ 3 \\ 6 \\ 1 \end{bmatrix}}$$

2.7 Homogeneous Coordinates and Projective Space

It follows from the above that all transformation matrices have different functions, but they all share something in common, a fourth row and a fourth column whose purpose has not been discussed. It may initially seem unnecessary when working only with scaling and rotation transformations. However, the reason relies on translations. An object can be translated by adding a translation vector to each of its vertices, as shown in Figure 2.9. Yet, a translation could no longer be chained as a matrix multiplication. For this reason, homogeneous coordinates appear in the scene to unify translation, rotation, and scaling as matrices [52].

As mentioned in [52], homogeneous coordinates were introduced by Mobius allowing calculations of graphics and geometry only possible in projective space. From [53], a point $\mathbf{p}$ is defined in the Euclidean space E^2 as a point with coordinates (x', y') or as a point

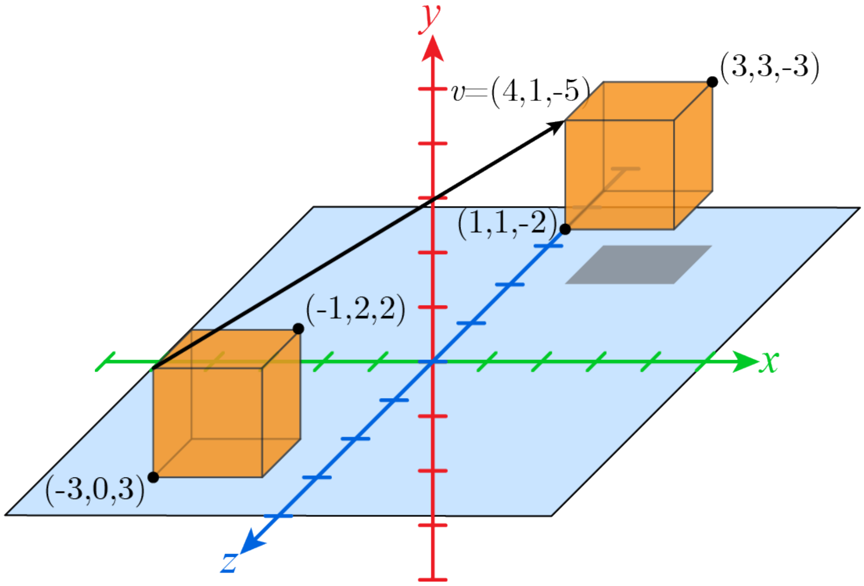


Figure 2.9: Result of applying a translation vector $v = (4, 1, -5)$ to a cube.

in homogeneous coordinates given by $(x : y : w)^T$ with $w \neq 0$ such that $x' = x/w$ and $y' = y/w$. At this point, one question is presented, why are these coordinates called homogeneous? Such an explanation can be inferred from the following examples:

$$
\begin{array}{ccc}
\text{Homogeneous} & & E^2 \\
(1:2:3) & \rightarrow & \left(\frac{1}{3}, \frac{2}{3}\right) \\
(2:4:6) & \rightarrow & \left(\frac{2}{6}, \frac{4}{6}\right) = \left(\frac{1}{3}, \frac{2}{3}\right) \\
(4:8:12) & \rightarrow & \left(\frac{4}{12}, \frac{8}{12}\right) = \left(\frac{1}{3}, \frac{2}{3}\right) \\
\vdots & & \vdots \\
(1a:2a:3a) & \rightarrow & \left(\frac{1a}{3a}, \frac{2a}{3a}\right) = \left(\frac{1}{3}, \frac{2}{3}\right)
\end{array}
$$

From the above, it is clear that all the homogenous points $(1 : 2 : 3), (2 : 4 : 6), (4 : 8 : 12), \ldots$ represent the same point $(1/3, 2/3)$ in E^2. The same concept can be extended to E^3, where a point or vector will be denoted by four components in homogeneous coordinates, $(x : y : z : w)$. Redirecting our attention to matrix transformations T, it is observed that the selection of $T_{4j} = [0, 0, 0, 1]$ and $T_{i4} = [0, 0, 0, 1]^T$ is not random. Neither is to set $w = 1$ in the homogeneous coordinate representation of a point or vector. Such configuration allows the resultant point or vector in homogeneous coordinates of our transformation to be the same in E^3 as $w = 1$.

Shafarevitch [54] gives the formal definition of a projective space as follows: Let V be a vector space of dimension $n + 1$ over the field k. The set of lines (that is, 1-dimensional vector subspaces) of V is called the $n-$dimensional projective space, and denoted by $\mathbb{P}(V)$ or $\mathbb{P}^n$. If we introduce coordinates $\xi_0, \cdots, \xi_n$ in V, then a point $\xi \in \mathbb{P}^n$ is given by $n + 1$ elements $(\xi_0 : \cdots : \xi_n)$ of the field k, not all equal to 0; and two points $(\xi_0 : \cdots : \xi_n)$ and $(\eta_0 : \cdots : \eta_n)$ are considered to be equal in $\mathbb{P}^n$ if, and only if, there exists $\lambda \neq 0$ such that $\eta_i = \lambda \xi_i$ for $i = 0, \cdots, n$. Any set $(\xi_0 : \cdots : \xi_n)$ defining the point ξ is called a set of homogeneous coordinates for ξ.

A common association of projective space is the problem of lines at infinity. Coxeter [55] mentions that two parallel lines in Euclidean space will never intersect even if they extend to infinity, however, when considering projective space they can intersect. For example, when observing a straight railroad line it appears that the two parallel rails converge at a point on the horizon. On the other hand, Hartley and Zisserman [56] comment that in computer graphics, the projective space $\mathbb{P}^3$ is used to represent a 3D scene of the real world. As a result, the rendered frames/images represent nothing more than the projection of the 3D objects in the scene onto a 2D plane that lies within the two-dimensional projective space.

2.8 The View Frustum

Recalling the modification of the pinhole camera reviewed in Section 2.2, it is time to introduce the view frustum. The view frustum represents the region of space containing all objects visible from the camera's perspective [1, 8]. This region is represented by a six-face volume as illustrated in Figure 2.10. The volume comprises a near plane and a far plane located at $z = -n$ and $z = -f$ respectively from the camera's position based on its local coordinate system. A third plane, the projection plane, is positioned at $z = -h$ from the camera's location between the near and far planes. Notice that the viewport resides at the projection plane. The rectangular shape of the view frustum is given by the aspect ratio s computed from the viewport's dimensions as $s = width/height = (r - l)/(t - b)$.

The distance h separating the viewport from the camera's eye is called the focal length [8, 30]. As the focal length increments, the field of view gets narrower while lowering allows for a larger view of the scene. Therefore, by varying the value of h a zoom-in/zoom-out effect is obtained. In addition, the angle formed by the left and right faces corresponds to the horizontal field of view FOVx. Meanwhile, the bottom and top faces form the angle for the vertical field of view FOVy. Following the parameters presented in Figure 2.10 the vertical and horizontal fields of view are calculated as follows:

$$\mathrm{FOV}x = 2\tan^{-1}\frac{s}{h} \quad \text{and} \quad \mathrm{FOV}y = 2\tan^{-1}\frac{1}{h}.$$

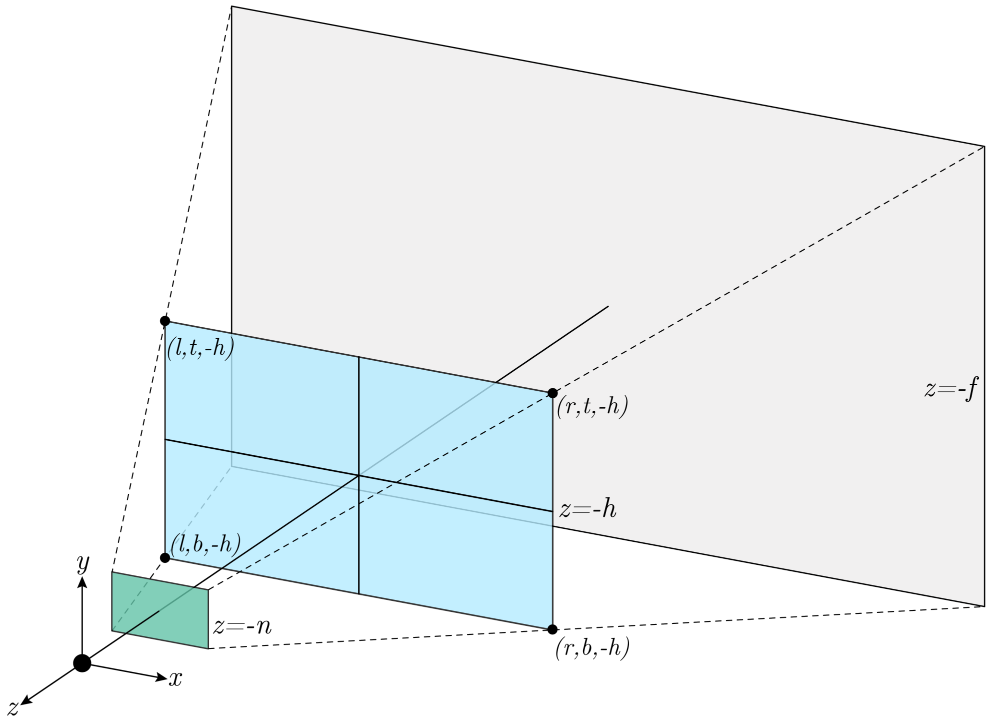


Figure 2.10: View frustum volume comprised of a near plane (green) and a far plane (gray), with the projection plane (blue) in between. Each plane is perpendicular to the viewing direction of the camera. Adapted from [8].

2.9 Coordinate Systems

The concept of coordinate systems is essential to properly render a scene from the camera's perspective in a 3D world. De Vries [2] does a good job explaining this concept. In summary, the coordinates of the objects, world, and view position should be transformed to determine the pixel coordinates in the final frame. The coordinate spaces of a rendering engine are presented in Table 2.1. Matrix transformations are employed to transition from one space to another, a detailed description of these matrix transformations is presented below.

The Model Matrix

The objects in world space are positioned by applying a model matrix to the local coordinates of objects. The model matrix consists of chaining rotations, scaling, or translation transformations to set the elements within the scene as desired. As observed in Figure 2.11 for instance, a model matrix can scale down an object, rotate it, and finally translate it

Table 2.1: Coordinate Spaces used in 3D Rendering [1, 2]

Coordinate Space	**Description**
Local Space	Coordinate space relative to the object. Often the origin coordinate of an object's 3D model is $(0, 0, 0)$.
World Space	Refers to the coordinates where the objects are positioned to comprise the scene world.
View Space	Also referred to as the camera or eye space, it corresponds to the view of the world and its objects from the camera's perspective.
Clip Space	Defines the range of all visible coordinates, coordinates outside this range are clipped, hence the name.
Screen Space	This space corresponds to the final mapping of pixels that represent the clip space of our scene from the camera's position and perspective.

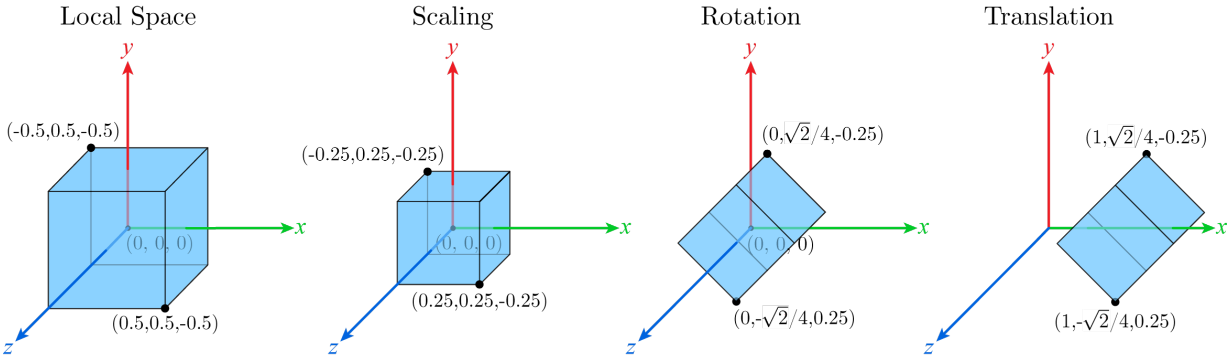


Figure 2.11: Model matrix transformation of a cube into world space comprised by: (1) scale down transformation, (2) rotation, and (3) translation.

into world space.

The View Matrix

The view matrix corresponds to the transformation applied to transition from world space to view space, providing the desired perspective from the camera's position and target direction [8]. The view matrix is computed from the camera's local coordinate axes: forward $\boldsymbol{\gamma}$, right α, and up β [2]. The forward axis is the normalized vector from the target to the camera's position, $\gamma = ||\mathbf{c}_{pos} - \mathbf{t}_{pos}||$. The right vector α is computed from the cross product between the forward vector γ and the canonical basis vector $\boldsymbol{j} = (0, 1, 0)$, $\alpha = \boldsymbol{j} \times \gamma$. Lastly, the up vector β, local to the camera's coordinate system, is obtained from the forward and right vectors cross product as $\beta = \gamma \times \alpha$. Consider that these vectors are computed following the right-hand rule convention for orienting the three-dimensional axes. Figure 2.12 presents the local coordinate axes of the camera.

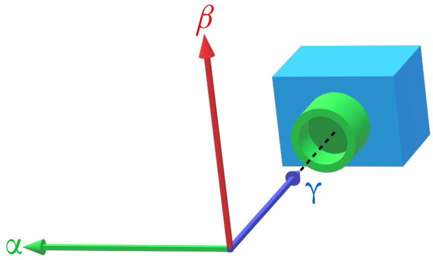


Figure 2.12: Camera's coordinate axes formed by the right, up, and forward vectors according to its angle of view.

The view space defines the world coordinates based on its coordinate system [1]. To do so, the view matrix comprises a translation matrix and a rotation matrix. Gambetta [57] provides an interesting analysis of the view matrix transform when the camera's coordinate system is not considered. Gambetta shows that keeping the camera fixed while translating and rotating the world in the opposite direction has the same effect as moving and rotating the camera in the world. If the camera's coordinate axes and world coordinate axes have the same orientation and scale but differ in their origin, then there is a translation vector that defines the transformation from one coordinate system to the other. Therefore, a point in world space can be expressed in view space coordinates by adding the opposite translation vector. Figure 2.13 illustrates the representation of the same point in the world coordinate system (red) and the view coordinate system (green).

The same intuition could be derived for the rotation of a coordinate system. Figure 2.14 shows two coordinate systems, the RGB coordinate axes belong to the world space whereas the black coordinate axes belong to the view space. Both coordinate systems share the same origin and scale and because the camera's target is the point $\mathbf{p} = (0, 0, -1)$ in world space the orientation of both coordinate axes is equal. Therefore, the point $\mathbf{p} = (0, -1, -1)$ has the same coordinates in the world and camera space.

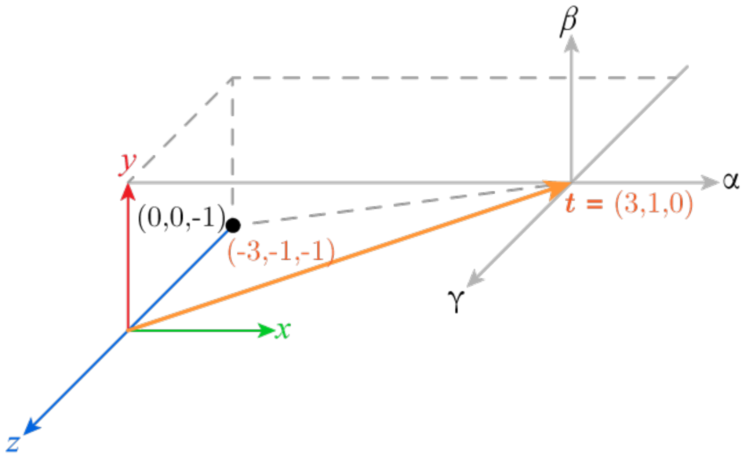


Figure 2.13: Transformation of one coordinate system to another given by the translation vector $\boldsymbol{t} = (3, 1, 0)$. The point $\mathbf{p} = (0, 0, -1)$ in the world coordinate system has the representation $\mathbf{p} = (0, 0, -1) - (3, 1, 0) = (-3. - 1, -1)$ in the view coordinate system.

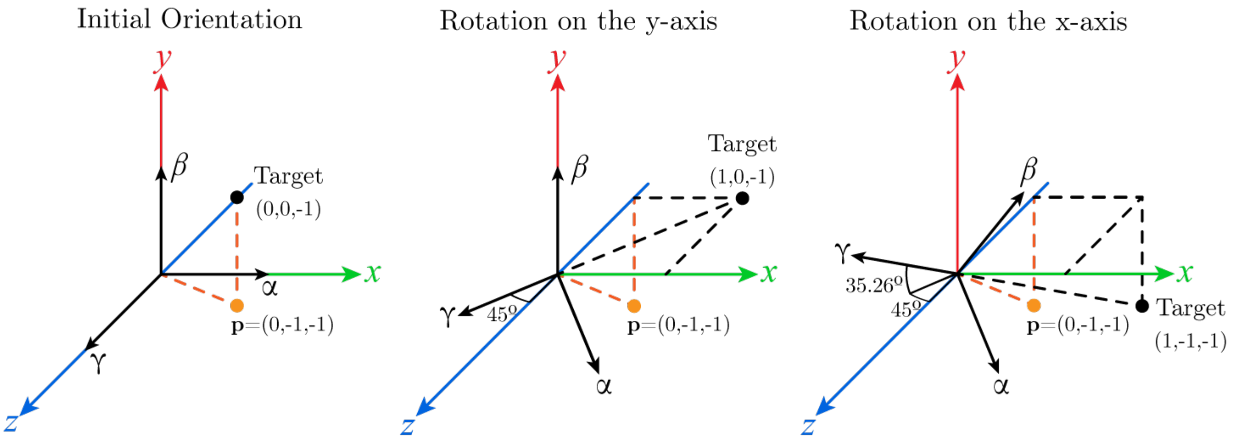


Figure 2.14: Rotation of the camera coordinate system by an angle of 45° respect to the y world space axis, followed by a rotation around the x-axis of 35.26°.

If the camera's target becomes the point $\mathbf{a} = (1, 0, -1)$ as shown in Figure 2.14, then the camera's vectors change: $\gamma = \text{norm}((0, 0, 0) - (1, 0, -1)) = (-1/\sqrt{2}, 0, 1/\sqrt{2})$, $\alpha = \text{norm}(\boldsymbol{j} \times \gamma) = (1/\sqrt{2}, 0, 1, \sqrt{2})$, and $\beta = \text{norm}(\gamma \times \alpha) = (0, 1, 0)$. As noticed, the camera's β vector remains the same, but γ and α have changed. The angle between the canonical vector $\boldsymbol{i} = (0, 0, 1)$ and the forward vector $\gamma = (-1/\sqrt{2}, 0, 1/\sqrt{2})$, is of 45° clockwise. In other words, the camera's coordinate axes have rotated 45° clockwise around the y-world space axis. The camera vectors themselves comprise the rotation matrix for rotating the camera coordinate system, without considering homogeneous coordinates it is defined as follows:

$$R_y = \begin{bmatrix} \cos 45 & 0 & \sin 45 \\ 0 & 1 & 0 \\ -\sin 45 & 0 & \cos 45 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 0 & 1 & 0 \\ -1/\sqrt{2} & 0 & 1/\sqrt{2} \end{bmatrix}$$

To determine the view space coordinates of point $\mathbf{p} = (0, -1, -1)$ lying in world space, it is multiplied by the rotation matrix R_y, such that $\mathbf{p}_V = R_y \cdot (0, -1, -1)^T = (-1/\sqrt{2}, -1, -1/\sqrt{2})$. The same analysis can be done if our target direction changes again as shown in the last part of Figure 2.14, where the camera's target direction is point $(1, -1, -1)$. The new camera vectors are: $\gamma = \text{norm}((0,0,0) - (1,-1,-1)) = (-1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{3})$, $\alpha = \text{norm}(\boldsymbol{j} \times \gamma) = (1/\sqrt{2}, 0, 1/\sqrt{2})$, and $\beta = \text{norm}(\gamma \times \alpha) = (1/\sqrt{6}, \sqrt{2/3}, -1/\sqrt{6})$. The angle between the previous and current forward vector is $\theta = \cos^{-1}((-1/\sqrt{2}, 0, 1/\sqrt{2}) \cdot (-1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{3})) \approx 35.26°$. These vectors form a new rotation matrix that rotates the camera regarding world space 45° around the y-axis and then $\approx 35.26°$ or ≈ 0.62 radians around the x-axis.

$$\begin{aligned} R &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos 35.26 & -\sin 35.26 \\ 0 & \cos 35.26 & \cos 35.26 \end{bmatrix} \cdot \begin{bmatrix} \cos 45 & 0 & \sin 45 \\ 0 & 1 & 0 \\ -\sin 45 & 0 & \cos 45 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sqrt{2/3} & -1/\sqrt{3} \\ 0 & 1/\sqrt{3} & \sqrt{2/3} \end{bmatrix} \cdot \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 0 & 1 & 0 \\ -1/\sqrt{2} & 0 & 1/\sqrt{2} \end{bmatrix} \\ &= \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 1/\sqrt{6} & \sqrt{2/3} & -1/\sqrt{6} \\ -1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \end{bmatrix} \end{aligned}$$

From the previous examples, it is noticed that to change the camera position around world space we applied a translation transformation. The motion column in the translation matrix is formed by the negative vector $\boldsymbol{t} = (t_x, t_y, t_z)$ to transition from the world space origin to the camera's position. In turn, a rotation transformation is applied to change the target direction of the camera. This rotation matrix is formed by the vectors forming the camera's coordinate axes Therefore, the view matrix transformation comprises a translation and a rotation matrix as presented below.

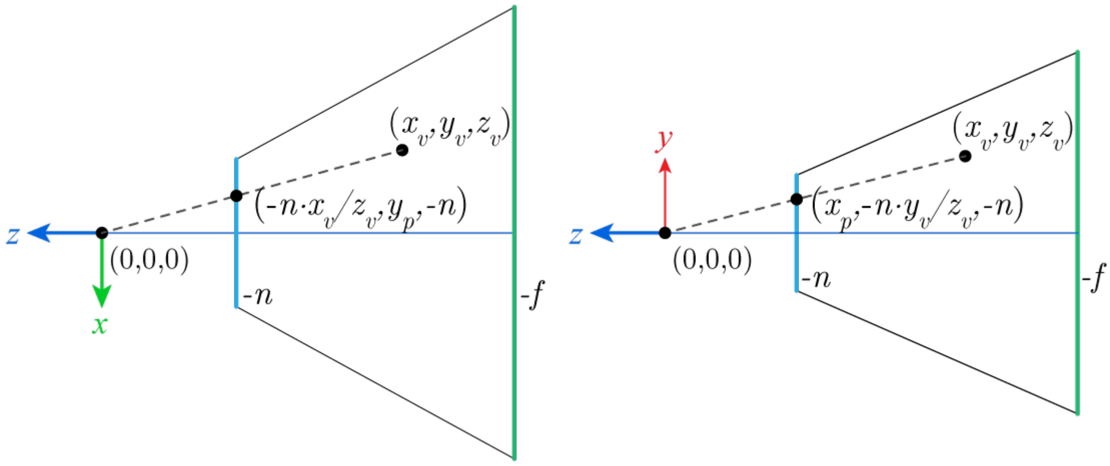


Figure 2.15: Top and side views of the frustum according to the view space coordinates of the camera. Adapted from [9].

$$V = \begin{bmatrix} \alpha_x & \alpha_y & \alpha_z & 0 \\ \beta_x & \beta_y & \beta_z & 0 \\ \gamma_x & \gamma_y & \gamma_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \alpha_x & \alpha_y & \alpha_z & -\alpha \cdot t \\ \beta_x & \beta_y & \beta_z & -\beta \cdot t \\ \gamma_x & \gamma_y & \gamma_z & -\gamma \cdot t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Perspective Projection Matrix

The projection matrix transforms coordinates from view space into clip space. The projections mentioned in the literature are perspective projection and orthographic projection [2, 8, 58]. Perspective projection looks to replicate the effect where objects farther away appear smaller from the view position [2]. In contrast, orthographic projections show objects the same size independently of distance [58]. This is useful in modeling, architecture, and engineering applications to depict the real dimensions of structures and components. However, perspective projection reproduces how depth perception works in human vision improving the realism of moving a camera around the scene.

The view frustum covered previously is related to perspective projection [8]. It allows projecting view space coordinates into clip space considering the camera's perspective. The projection matrix is built by considering the view frustum with the projection plane equal to the near plane. Figure 2.15 presents a top view and a side view of the frustum. The ratio of similar triangles is employed to find the coordinates of a projected point into the near plane [1, 9]. Let's start by finding x_p and y_p.

$$\frac{x_p}{x_v} = \frac{-n}{z_v} \qquad\qquad \frac{y_p}{y_v} = \frac{-n}{z_v}$$

$$x_p = \frac{-n \cdot x_v}{z_v} = \frac{n \cdot x_v}{-z_v} \qquad\qquad y_p = \frac{-n \cdot y_v}{z_v} = \frac{n \cdot y_v}{-z_v}$$

The division of both terms by $-z_v$ is known as perspective division and allows objects farther away from the camera to be represented smaller than those that are closer [8]. Furthermore, notice that after multiplying a view space coordinate by the projection matrix the result in clip space is still expressed in homogeneous coordinates. Therefore a clip space coordinate is divided by its w homogeneous component to be mapped into the range $[-1, 1]$ that comprises a space known as normalized device coordinates (NDC) [2]. The division by the w component is known as perspective division. After applying the perspective projection matrix and perspective division, the view frustum has been transformed into the canonical view volume whose coordinates are in NDC space [8].

$$\text{Projection Matrix} \cdot \begin{bmatrix} x_v \\ y_v \\ z_v \\ w_v \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \rightarrow \begin{bmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{bmatrix} = \begin{bmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{bmatrix}$$

Because x_p and y_p are inversely proportional to $-z_v$, then the w_c becomes $-z_v$ and the fourth row of the projection matrix becomes:

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_v \\ y_v \\ z_v \\ w_v \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

The clip space values of x_c and y_c are computed based on their mapping to NDC coordinates considering the linear relation where $[l, r] \rightarrow [-1, 1]$ and $[b, t] \rightarrow [-1, 1]$ [8, 9].

$$
\begin{aligned}
x_{NDC} &= \frac{1-(-1)}{r-l}\cdot x_p + \phi\\
\rightarrow 1 &= \frac{2r}{r-l} + \phi \quad \text{(substitute } (x_p, x_n) \text{ for } (r,1))\\
\rightarrow \phi &= 1 - \frac{2r}{r-l} = \frac{r-l}{r-l} - \frac{2r}{r-l}\\
&= \frac{r-l-2r}{r-l} = \frac{-r-l}{r-l} = -\frac{r+l}{r-l}\\
\therefore x_{NDC} &= \frac{2x_p}{r-l} - \frac{r+l}{r-l}
\end{aligned}
$$

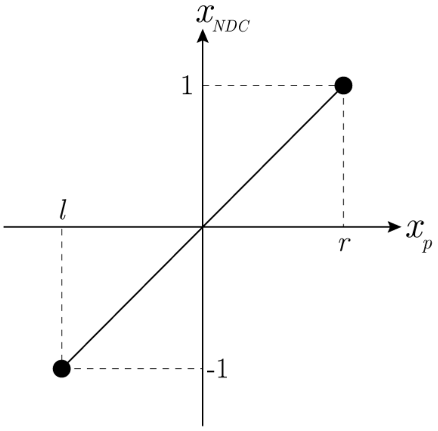


Figure 2.16: Mapping x_p to x_n

$$
\begin{aligned}
y_{NDC} &= \frac{1-(-1)}{t-b}\cdot y_p + \phi\\
\rightarrow 1 &= \frac{2t}{t-b} + \phi \quad \text{(substitute } (y_p, y_n) \text{ for } (t,1))\\
\rightarrow \phi &= 1 - \frac{2t}{t-b} = \frac{t-b}{t-b} - \frac{2t}{t-b}\\
&= \frac{t-n-2t}{t-b} = \frac{-t-b}{t-b} = -\frac{t+b}{t-b}\\
\therefore y_{NDC} &= \frac{2y_p}{t-b} - \frac{t+b}{t-b}
\end{aligned}
$$

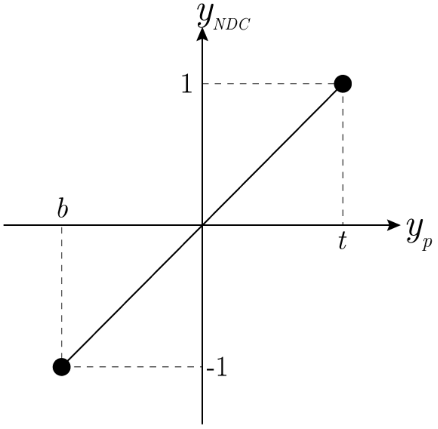


Figure 2.17: Mapping y_p to y_n

Then the values of x_p and y_p can be substituted based on their view space representation calculated previously, such that:

$$
\begin{aligned}
x_{NDC} &= \frac{2x_p}{r-l} - \frac{r+l}{r-l}\\
&= \frac{2\cdot\dfrac{n\cdot x_v}{-z_v}}{r-l} - \frac{r+l}{r-l}\\
&= \frac{2n\cdot x_v}{(r-l)(-z_v)} - \frac{r+l}{r-l}\\
&= \frac{\dfrac{2n}{r-l}\cdot x_v}{-z_v} + \frac{\dfrac{r+l}{r-l}\cdot z_v}{-z_v}\\
&= \underbrace{\left(\frac{2n}{r-l}\cdot x_v + \frac{r+l}{r-l}\cdot z_v\right)}_{x_c} / -z_v
\end{aligned}
$$

$$
\begin{aligned}
y_{NDC} &= \frac{2y_p}{t-b} - \frac{t+b}{t-b}\\
&= \frac{2\cdot\dfrac{n\cdot y_v}{-z_v}}{t-b} - \frac{t+b}{t-b}\\
&= \frac{2n\cdot y_v}{(t-b)(-z_v)} - \frac{t+b}{t-b}\\
&= \frac{\dfrac{2n}{t-b}\cdot y_v}{-z_v} + \frac{\dfrac{t+b}{t-b}\cdot z_v}{-z_v}\\
&= \underbrace{\left(\frac{2n}{t-b}\cdot y_v + \frac{t+b}{t-b}\cdot z_v\right)}_{y_c} / -z_v
\end{aligned}
$$

The numerators of the previous terms represent the x_c and y_c coordinates of a point in clip space. After applying perspective division, dividing the terms by $-z_v$, the coordinates are mapped into their canonical view volume representation x_{NDC} and y_{NDC}. Recalling our previous form of the projection matrix, the first and second rows could now be expressed as:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ . & . & . & . \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_v \\ y_v \\ z_v \\ w_v \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

Finally, the third row to compute z_c is left to discover. Notice that z_v is projected into $-n$, but there is the need for a unique z-value that accounts for clipping and depth testing [9, 59]. Moreover, because z_c does not depend on x_v or y_v, the w_v component is employed to find z_c. The variables that we need to solve for are A and B such that:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_v \\ y_v \\ z_v \\ w_v \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

and,

$$\begin{aligned} z_{NDC} &= \frac{z_c}{w_c} = \frac{Az_v + Bw_v}{-z_v} \\ &= \frac{Az_v + B}{-z_v} \qquad (w_v = 1 \text{ in view space}) \end{aligned}$$

Because the view space z-coordinate should also be transformed into NDC space, the linear relation $[-n, -f] \rightarrow [-1, 1]$ is employed to find A and B as follows.

$$\begin{cases} \frac{-An + B}{n} = -1 \\ \frac{-Af + B}{f} = 1 \end{cases} \rightarrow \begin{cases} -An + B = -n \\ -Af + B = f \end{cases}$$

After solving the system of equations, $A = -\frac{f+n}{f-n}$ and $B = -\frac{2fn}{f-n}$. Thus, it follows that

$$z_n = \frac{-\frac{f+n}{f-n} z_v - \frac{2fn}{f-n}}{-z_v}$$

and finally, the standard perspective projection matrix [59] is expressed as

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Screen Space

Once our coordinates are in NDC space, the only step left is to map them into their screen space representations. This process depends on the rendering pipeline, for instance in the rasterization pipeline [60, 61], the NDC coordinates are mapped into screen coordinates using a viewport transformation [2]. However, if we recall the ray-tracing rendering process in its backward variant, a ray is sent into the center of all viewport pixels. Notice carefully that in ray tracing the rendering process is inversed, that is, starting from screen space the objective is to reach world space coordinates. Moreover, defining the previous matrix transformations and coordinate spaces from a rasterization viewpoint helps to grasp the concepts more easily.

The rendering process starts by transforming the viewport pixel coordinates into its NDC space representations. Considering a viewport of 1920x1080 pixels (width × height), a pixel (i, j) where $0 <= i <$ width and $0 <= j <$ height has the following NDC representation.

$$x_{NDC} = \frac{2i - \text{width}}{\text{width}}, \qquad y_{NDC} = \frac{2j - \text{height}}{\text{height}}, \qquad z_{NDC} = -1$$

Notice that, i and j are transformed to be in $[-1, 1]$. Additionally, $z_{NDC} = -1$ because the viewport will be positioned at the near plane. Moreover, recall that perspective division is performed to transition from clip space to NDC space. As such, the w clip space component is set to 1 representing that perspective division has already been applied. In this regard, a clip space coordinate is represented as $(x_{NDC}, y_{NDC}, z_{NDC}, 1)$. Once in clip space, each coordinate is transformed by the inverse projection matrix into a view space coordinate. Finally, the inverse view matrix transforms every view space coordinate into its world space representation.

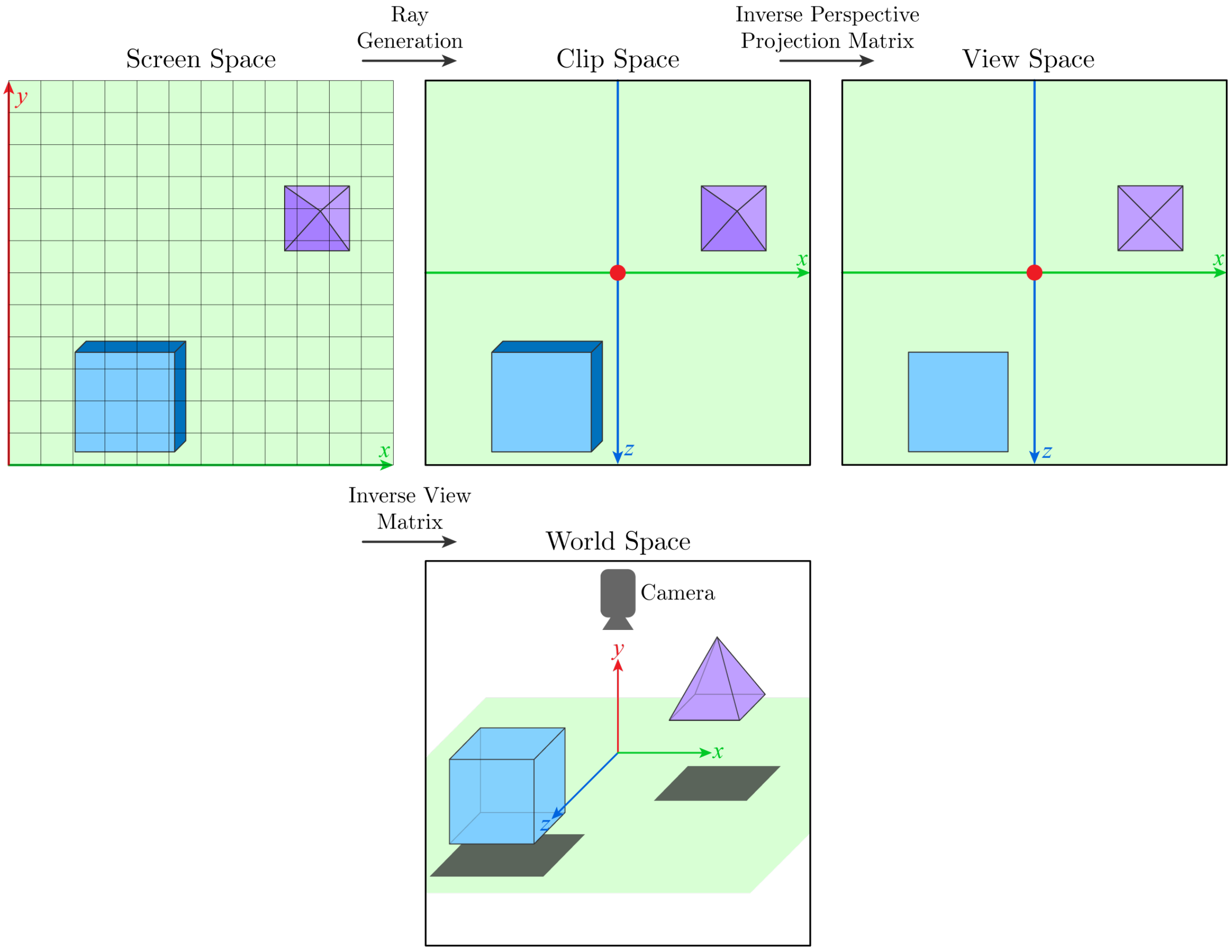


Figure 2.18: Coordinate spaces based on the ray tracing rendering process.

The overall ray tracing rendering process is presented in Figure 2.18. The model matrix is not considered in this pipeline, as the objects should rather be correctly placed into world space before sending rays into the scene. In any rendering pipeline, an object's model is represented by a 3D mesh of polygons often triangles [2, 60]. As such, when loading a model's mesh the model matrix should be applied to it. All the loaded meshes comprise the scene that will be rendered.

Chapter 3

State of the Art

Ray tracing has seen significant advancements in recent years, with improvements across various areas. Among these are hybrid rendering pipelines that integrate rasterization with ray tracing to optimize performance, and acceleration structures that minimize ray-intersection operations, enhancing speed and efficiency. Dedicated ray tracing hardware has also emerged, specialized for real-time rendering, while denoising techniques have been developed to reduce noise in complex renders. As performance improves, new methods have been introduced to model an even broader range of materials. However, a deep understanding of many of these techniques requires foundational knowledge and experience in computer graphics and is beyond the scope of this work. Thus, this chapter provides a summary of some key advancements for further exploration after reviewing the fundamentals covered here.

3.1 Hybrid Ray Tracing Rendering

One of the initial proposals of hybrid rendering was introduced by Beck et al. [62], who developed a CPU-GPU Real-Time Ray Tracing framework. The framework starts with a shadow map pass, a geometry identification pass, and a blur pass all performed in GPU. The triangle IDs resulting from the geometry pass are encoded as an RGB color in a framebuffer, along with the value of the shadow map pass stored in the alpha channel. The blur pass is used to modify the alpha channel to delimit the shadow boundaries. The resultant framebuffer is received by the CPU in which a ray-tracing algorithm generates the reflection and refraction rays. Finally, using the Phong shading model (discussed in Section 4.8), the final render is obtained.

Sabino et al. [63], in turn, propose a GPU-only hybrid pipeline combining both rasterization and ray tracing. In the first stage using deferred rendering [64, 65], a G-Buffer containing geometry information such as positions, normals, texture coordinates, and material parameters is computed. This stage avoids computing the initial ray-primitive intersection pass to determine object visibility. The second stage is responsible for computing shadows using the information of the G-Buffer and the light positions generating a shadow buffer. In the third stage, the effects of reflections and refractions are computed using a ray-tracing algorithm generating a third buffer. Finally, the last stage involves combining the buffers of each stage to render the final image.

A third solution is the PICA PICA real-time ray tracing experiment [66] showcasing self-learning agents in a procedurally generated 3D world. PICA PICA combines the common rasterization pipeline with the modern compute and ray tracing pipelines [36, 67, 68]. The rendering pipeline consists of eight stages, starting with computing a G-buffer with rasterization to subsequently compute a direct shadows buffer using a ray tracing algorithm or the rasterization process as well. Using a computer shader (discussed in Section 4.1) the effect of direct lighting is computed. The effects of reflections and global illumination, stages five and six, are then added by using either the ray tracing pipeline or the compute pipeline. Similarly, the effects of ambient occlusion and transparency/translucency are generated by ray tracing or compute pipelines. The last stage corresponds to post-processing, generating the final render.

There are several more examples of hybrid rendering such as HART [69], RenderMan [70], Unity High-Definition Render Pipeline [71], Unreal Engine 4 [72], the combination of Blender's EEVEE and Cycles rendering engines [73], among others. By combining these methods, hybrid rendering pipelines achieve a balance where rasterization handles the bulk of straightforward scene rendering, while ray tracing is selectively applied for effects that benefit most from its physical accuracy. This selective approach allows for photorealistic results with much lower computational cost compared to fully ray-traced rendering. As hardware and software optimizations continue to evolve, hybrid rendering is poised to become an integral part of achieving ever more immersive and realistic virtual experiences.

3.2 Acceleration Structures

Ray-intersection operations are some of the most performance-intensive tasks in ray tracing, as they determine object visibility within a scene and are required for shading calculations. With a fixed number of primitives n, a single ray-intersection test has a time complexity of $\mathcal{O}(n)$. However, considering the full viewport, multiple rays per pixel, and potential ray bounces, the computational load becomes prohibitive for real-time applications. To address this, acceleration structures are used to minimize intersection tests through two primary approaches: spatial subdivision (e.g., grid-based structures, k-d trees) and primitive subdivision (e.g., bounding volume hierarchies, or BVH) [74–76]. A formal introduction to these approaches is presented in the Methodology chapter, whereas this section discusses the improvements and variations of today's methods for further reading.

In contemporary GPU-based ray tracing, the bounding volume hierarchy (BVH) is the standard acceleration structure due to its efficient memory footprint, parallelization capabilities, compact traversal, rapid construction, and adaptability to dynamic geometry [67, 68]. The most common variant, an axis-aligned bounding box (AABB) BVH that uses the surface-area heuristic (SAH) for partitioning, significantly reduces the number of intersection tests and has inspired numerous enhancements to improve performance further [11]. One early improvement was introduced by Lauterbach et al. [77], who developed the linear BVH (LBVH) construction algorithm. This approach uses the GPU for parallel processing, organizing primitives using Morton codes [78]. Later, Pantaleoni and Luebke [79] extended this by combining LBVH with the surface-area heuristic to optimize the upper levels of the tree, an approach known as hierarchical LBVH (HLBVH).

Garanzha et al. [80] introduced a simpler and faster variant of HLBVH by incorporating the SAH binning algorithm [81]. Karras [82] proposed a different method, creating a parallel algorithm to build the entire LBVH structure using binary radix trees as foundational elements. Karras's approach includes an additional step to compute bounding boxes, which increases the overall cost of BVH construction. To address this, Apetrei [83] developed a method to simultaneously construct the topology and assign bounding boxes, reducing construction time. In turn, Chitalu et al. [84] introduced a binary ostensibly-implicit tree structure instead of a binary radix tree. This approach encodes the BVH topology as binary representations, allowing the identification of missing nodes in a complete binary tree. Consequently, implicit nodes can be mapped to compact memory locations, resulting in fast construction while remaining memory-efficient and computationally light.

The bounding volume used to partition the primitives also impacts the number of intersections tests, as a loose fit may result in unnecessary ray-intersection operations. The discussed approaches reduced this effect to some extent by assigning each primitive its own axis-aligned bounding box (AABB). An alternative approach is to employ different bounding volumes to achieve a tighter fit. One of the early alternatives was the use of oriented bounding boxes (OBB), introduced by O'Rourke [85]. Although OBBs provide a compact fit, the method initially proved slow. Gottschalk [86] improved on this by using principal component analysis (PCA) to approximate the OBB. Later, Chang et al. [87] developed an approach to compute minimal volume OBBs through a hybrid optimization algorithm that searches the space of rotation matrices. Larsson and Källberg [88] proposed the di-tetrahedron OBB (DiTO) algorithm, which balances bounding volume quality and performance, offering a more efficient alternative compared to the previous approximation methods.

Discrete orientation polytopes bounded by k hyperplanes (k-DOPs) are another compact bounding volume for geometry [89]. As k increases, the hyperplanes form a convex polyhedron that tightly encloses a primitive. Klosowski et al. [90] proposed one of the initial methods for constructing BVHs using k-DOPs, aimed at improving collision detection efficiency. Later, advanced this idea by introducing velocity-aligned DOPs (VADOPs), bounding volumes based on k-DOPs that demonstrated real-time performance in dynamic collision detection. Nonetheless, constructing BVHs with k-DOPs and the surface-area heuristic remains challenging, as calculating the surface area of a polytope with many faces is computationally expensive [91].

Recent approaches leverage k-DOPs as building blocks for constructing oriented bounding boxes (OBBs) For instance, Vitsas et al. [92] employ the di-tetrahedron-OBB algorithm to generate a representative set of points for the underlying geometry, then select a tightly fitting OBB, where the di-tetrahedron represents a polytope. Sabino et al. [93] propose a different method, utilizing orthogonal discrete orientation polytopes (ODOPs) to store the mesh's topological features, allowing straightforward conversion to an OBB. Káčerik and

Bittner [91] revisited the use of k-DOPs for BVH construction, introducing an optimized algorithm that uses the surface-area heuristic (SAH) to improve topology. Their method incorporates the parallel locally ordered clustering (PLOC) algorithm [94] to compute a fast, exact evaluation of the surface area for a 14-DOP. Although this method requires more construction time and memory than an AABB-based BVH, it achieves a 2.5x increase in ray tracing speed.

3.3 Denoising Techniques

The increased realism in complex scenes brought by modern ray tracing algorithms comes at the cost of high rendering times for noise-free images. Examples of these algorithms include Path Tracing [95], Bidirectional Path Tracing [96], and Metropolis Light Transport [97]. These algorithms are employed to transport light from one point of the scene to another, simulating the complex interactions of light with surfaces, including reflection, refraction, and scattering. Path Tracing, for instance, traces rays from the camera into the scene, following their paths as they bounce off surfaces until they reach a light source or exit the scene [11, 36]. Bidirectional Path Tracing enhances this approach by tracing rays both from the camera and from light sources, meeting in the middle to handle difficult lighting scenarios such as caustics better [11, 96]. Metropolis Light Transport, on the other hand, uses a more sophisticated approach by focusing sampling efforts on important light paths, effectively reducing noise and increasing efficiency in scenes with complex lighting [11, 97].

Incorporating these techniques in real-time rendering is not feasible because of the multiple path samples required to compute realistic lighting. Because of the limited number of samples that can be generated in a real-time renderer, denoising techniques are introduced [3, 36]. These techniques aim to mitigate the noise in low-sample renders by predicting and correcting pixel values based on surrounding pixel information, temporal data from previous frames, or by using machine learning and deep learning methods trained to reduce noise patterns. As explained by Bako et al. [98] and Firmino et al. [99], the most well-known denoising techniques belong to the class of non-linear image space filters. Non-linear filters adapt to the local distribution of noise levels in pixels [100], allowing them to effectively reduce noise in complex regions of an image while preserving important features such as edges and detail [101]. Some of these filters include bilateral filtering [102], anisotropic diffusion [103], À-Trous wavelet transform [104], and guided filtering [105].

More advanced methods involve a series of filtering stages to further reduce noise while maintaining realism. Some of these techniques include Spatiotemporal Variance-Guided Filtering (SVGF) [106], adaptive SVGF [107], and Reservoir-based Spatiotemporal Importance Resampling (ReSTIR) [108]. SVGF improves upon traditional variance-guided filters by incorporating temporal accumulation, spatiotemporal luminance variance, and a wavelet filter to reduce noise while preserving high-frequency details [106]. Adaptive SVGF enhances this method by analyzing previously generated frames to reconstruct an adaptive temporal gradient, thereby reducing lag and ghosting [107]. Finally, ReSTIR utilizes a reservoir-based approach to resample and redistribute light paths, optimizing sampling efficiency and enhancing the accuracy of light simulation across both space and time [108, 109].

In recent years, there has also been the inclusion of machine learning and deep learning methods specifically trained to reduce the noise in frames. For instance, Chaitanya et al. [110] developed a denoising autoencoder that receives noisy frames as input and reconstructs them by removing the noise present. Bako et al. [98] trained a convolutional neural network to learn the relationship between noisy and reference frames using a dataset of several frames from the film Finding Dory, which contains various effects. In turn, the approach taken by Xu et al. [111] uses a generative adversarial network (GAN) and a conditioned feature modulation technique to integrate auxiliary information into the network and produce a noise-free frame. A more complex architecture is introduced by Hofmann et al. [112], which comprises a dual autoencoder to handle the noise of direct and indirect lighting separately, a special noise function for the reconstruction of specular highlights, and a relativistic discriminator to improve sharp details.

3.4 Summary

Current innovations in ray tracing rendering span a wide range of disciplines, including computer science, mathematics, and physics. Discussing every aspect of these advancements is challenging due to the highly specialized nature of the concepts involved. For example, integrating deep learning techniques into ray tracing algorithms requires a thorough understanding of both light transport principles and neural network design. Similarly, progress in hardware acceleration demands knowledge of GPU architecture and parallel computing. Despite covering the basics of light transport, physically based rendering, and shading models in the following chapter, these only scratch the surface of modern ray tracing and computer graphics.

Chapter 4

Methodology

In this chapter, we will cover all the concepts for implementing a ray-tracing renderer. Although some programming requirements are required, relevant and useful sources will be provided for you to understand every implementation aspect.

4.1 Implementation Details

Real-time rendering is only appreciable if interaction with all the components of a scene is granted. Luckily, this is possible by working with a highly parallelizable processor like the GPU, that allows us to render multiple frames of our scene per second. Working with a GPU, however, is not straightforward, primarily because each GPU has a different architecture. Furthermore, GPU's resource access depends largely on the drivers provided by the company that designs the architecture, where NVIDIA [113] and AMD [114] are two of the largest GPU companies in the world. Since the drivers for each GPU are architecture-dependent, a standard API is required to access GPU resources.

Perhaps one of the most known APIs/specifications is OpenGL [38], which defines standardized mechanisms to access GPU resources across multiple platforms. Providing standardized mechanisms ensures that GPU resources can be used regardless of the difference in drivers/controllers in each architecture. Such mechanisms are written by the GPU manufacturers taking into consideration the OpenGL specification. Likewise, being cross-platform offers an important advantage over other platform-specific APIs such as DirectX [39] or Metal [41] since the programs created can be run on different operating systems including some modifications. In this regard, the following work will be implemented completely in OpenGL. Table 4.1 shows a complete list of the implementation details of the renderer. Moreover, a repository with the ray tracing renderer and all the code examples derived from this section are available in a GitHub repository that contains all the details about installation and usage, see Appendix A for the link.

Learning about computer graphics, and specifically about OpenGL may look complicated at first, but there are several online resources available. One of the best online resources referenced already is that of Joey de Vries [2] who provides an extensive guide for learning OpenGL. Another well-recognized resource is the OpenGL Programming Guide [61] also known as the Red Book. Regarding video tutorials, I will recommend the YouTube series of The Cherno [116], Michael Grieco [117], and GetIntoGameDev [118]. It must be emphasized that this work does not cover the OpenGL API as it will require a complete chapter or even more to cover the essentials. Nonetheless, all the references provided in this chapter will serve as an additional source to help you understand better the different concepts presented.

4.1.1 Shaders and the Rasterization Pipeline

Before introducing the rendering pipeline used in this work, the concept of shaders must be presented. As Lengyel [8] comments, the processing a GPU does during rendering is determined by a series of operations defined within the rendering pipeline. These operations can, for example, transform the primitives that make up the scene, refine the 3D model meshes, generate realistic lighting conditions, apply textures, etc [3, 61]. The instructions that detail how those operations are performed are specified within programs denominated shaders which execute in the GPU. Shader instructions are defined using C-like shading languages, for instance, OpenGL uses the OpenGL Shading Language (GLSL) whereas DirectX uses the High-Level Shading Language (HLSL) [3, 61]. Furthermore, shader instructions are highly optimized to take advantage of the parallelization offered by GPUs.

Several types of shaders are already defined within the rendering pipeline, however, our interest is focused on only three, vertex, fragment, and compute shaders. It is important to emphasize that vertex and fragment shaders belong to the rasterization pipeline, yet

Table 4.1: Implementation details of the renderer including general development aspects, platform support, and libraries.

Aspect	**Description**
Programming Language	The complete implementation is developed under the C programming language standard using the C99 version.
Platform Support	The renderer and all code examples created can be compiled to run both on Linux and Windows.
Compilers	Both the Clang and GNU gcc compilers can be employed to generate the binaries in Linux. Generating binaries on Windows requires installing MinGW, which is a development environment for creating native Windows applications using the GNU gcc compiler.
Build Automation	CMake 3.27.0 is employed to generate the corresponding build files while `make` is used for build automation. In Windows, the `mingw32-make` tool is employed to build and generate the binaries.
OpenGL version	Version 4.6 of the OpenGL specification is used.
GLFW	Cross-platform library for OpenGL and Vulkan that provides an API for creating windows, contexts, and surfaces, while offering support for inputs and events. GLFW manages the OpenGL context required for rendering and, as such, handles the version of OpenGL to be used.
GLAD	Generates a loader to dynamically load function pointers at runtime to be used within the OpenGL context created.
CGLM	Highly optimized math library written in C for working with 2D/3D math operations. It includes SIMD versions of different functions.
Tinyobjloader-c	Lightweight header-only C-library for loading triangular meshes of 3D models in the wavefront.obj format. The library is easy to use and was incorporated easily into the renderer.
Camera Model	The camera model was implemented following all the notions described in the Coordinate Systems section of Chapter 2. In addition, De Vries [2] devotes an entire chapter to the implementation of the rasterization camera model, which can be easily adapted for the ray-tracing renderer.
Random Number Generation	The PCG hash function [115] was implemented in GLSL for random number generation in the GPU.

their function in our pipeline will be discussed later. If you recall the section on coordinate systems reviewed in Chapter 2, the rasterization pipeline goes from world space into screen space. Such operations can be covered within the vertex shader and fragment shader.

Vertex shaders as their name suggests process the vertices comprising the primitives of the models that are going to be rendered [61]. A model matrix transformation is applied to the vertices of the 3D models comprising the scene if needed. Otherwise, their coordinates are considered to be already in world space. The vertex shader then transforms the vertices coordinates to their corresponding view-space representation. Color, normals, and texture attributes are also passed to vertex shaders which in turn redirect them to the following rendering stages until the fragment shading stage.

The primitive assembly stage then generates the corresponding primitives, where triangles are often the selected primitive [2]. This stage is automatically handled by the rasterization pipeline. The generated primitives are then passed to another automatic stage known as clipping. In this stage, primitives whose coordinates are outside the viewport are clipped [1]. The output of this stage is then processed by the rasterizer for fragment generation, where the clipped primitives are mapped to their corresponding pixel positions [2, 61]. In addition, during rasterization, attributes such as color and texture coordinates specified at the vertices are interpolated across the surface of the primitive [2]. This interpolation determines the attribute values in each fragment ensuring the correct blending of colors and proper mapping of textures within the primitive.

Subsequently, the generated fragments are then processed by the fragment shader to determine their final color [8, 61]. During this stage, different effects can be incorporated to enhance the realism of the rendered scene such as lights, shadows, bloom effects, etc. A commonly used shading model to account for light is the Phong model [119] which will be discussed later. Finally, the checking and blending stage involves the final processing of the individual fragments before generating the final frame [61]. This stage includes depth testing to determine which fragment is in front, discarding those that are occluded by others. It also includes color blending of the fragments at the boundaries of the primitives, which ensures proper layering of the overlapping objects [2]. See Figure 4.1 for an illustration of the rasterization pipeline.

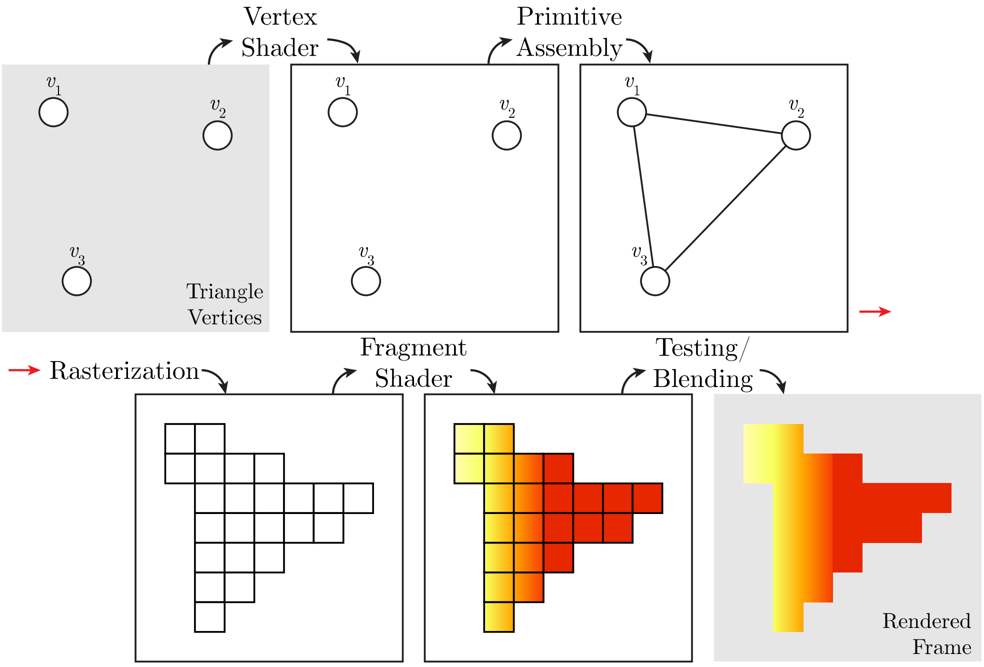


Figure 4.1: Rasterization pipeline for rendering a simple triangle. Because the coordinates are within the viewport the clipping stage is not illustrated.

4.1.2 Ray Tracing Pipeline

Earlier, it was mentioned that compute shaders were also of interest for the pipeline of this work. However, these were not mentioned at all in the rasterization pipeline, why is that? Traditionally, GPUs were used primarily for rendering, but as computational tasks became more demanding, the perspective changed. The parallelization capabilities offered by GPUs were taken into account, and new parallel algorithms were developed to solve tasks faster. Such tasks are commonly enclosed under the category of general-purpose computing on graphics processing units (GPGPU) [2].

The paradigm employed by GPUs to process data in parallel is known as single instruction multiple data (SIMD) [120], where a set of operations is applied independently to the data being processed. Furthermore, the functions describing the operations to be performed are denominated kernels, an example being the fragment shader [2, 121]. Two well-known frameworks for GPGPU computing are CUDA [42] specific for NVIDIA GPUs and OpenCL [122] which is a cross-platform API that supports different hardware and GPU architectures. An alternative solution is compute shaders which date before the launch of GPGPU-specific solutions [43, 61, 121].

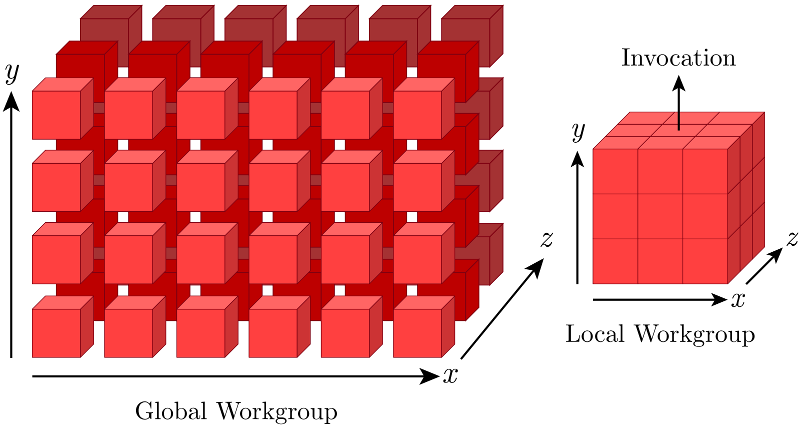


Figure 4.2: Workgroup hierarchy used to instantiate compute shaders and parallelize a task.

Compute shaders are graphic APIs' solution to enable GPGPU computing in an isolated single-stage pipeline [61, 121]. Despite their isolated execution, they can still communicate with the common rendering pipeline by employing different data storage structures. Likewise, there are several forms to map data to a compute shader, the main one used in this work is a `Image2D` variable [61], to which the ray-traced rendered frame will be saved. Whereas in the rendering pipeline task parallelization is automatically handled, in compute shaders the parallelization of a task is more explicit.

Compute shaders are parallelized using workgroups, where each workgroup represents a collection of GPU threads also referred to as invocations [2]. Figure 4.2 illustrates the hierarchy or workgroups, where the total number of threads employed for a task comprises the global workgroup. Local workgroups, in turn, comprise a subset of these threads that can communicate and share memory locally. The organization of a global and a local workgroup is three-dimensional, however, our problem has only 2 dimensions as the final output of the ray tracing renderer is an image. Defining the number of threads to be used is done by calling the **`glDispatchCompute`**`(groups_x,groups_y,groups_z)` function [2, 61]. For instance, if the rendered image size is 1920 × 1080 pixels, the number of threads per local workgroup can be set to $x = 4$, $y = 8$, and $z = 1$, as there is no need for the third dimension. This means each local workgroup will handle 4 pixels in the x-axis and 8 pixels in the y-axis. In this sense, the number of local workgroups to process all the pixels of the image across each dimension is

$$
\begin{aligned}
x &= 1920/4 = 480 \\
y &= 1080/8 = 135 \\
z &= 1.
\end{aligned}
$$

Therefore, the global workgroup has $480 \times 135 = 62400$ local workgroups with $1920 \times 1080 = 2073600$ threads in total. Specifically, on NVIDIA GPUs, the number of threads per local group is 32 to properly parallelize tasks [120]. In this regard, the previous example represents a possible combination to ensure 32 threads per local group. The ray tracing pipeline employed in this work, see Figure 4.3, is separated into two parts:

- **Compute Shader Pipeline**
    1. Initialization Stage: All the details regarding the scene to be rendered must be defined. This includes the primitives or complete 3D models, the details of the camera (position, target direction, coordinate axes), the acceleration structure to speed the rendering of each frame, and binding the Image2D variable to which the rendered frame will be stored. The concept of acceleration structure is reviewed later in this chapter.
    2. Compute Shader Stage: The ray tracing rendering of a scene is handled by a compute shader. For each pixel of the image to be rendered, a ray will be emitted for each pixel of the scene from the camera's position. Every ray emitted is handled independently by only one thread/invocation of a local group. During this stage, the shading and lighting of a scene are computed with the details discussed later in this chapter. Finally, every instance of the computer shader writes its output to its corresponding pixel in the Image2D variable.
- **Rasterization Pipeline** Because the image cannot be displayed directly by itself, the rasterization pipeline is used to render the image on the screen. To do so, a large enough triangle covering the entire viewport is defined. The generated image is then bound as a texture that will be mapped onto the rendered triangle, thus displaying the ray-traced rendering.

- Vertex Shader: The triangle vertices are passed to a vertex shader responsible for positioning the triangle such that the ray-traced rendered frame will be displayed correctly. Considering that OpenGL defines coordinates in NDC space [2, 61], the triangle vertices used to cover the entire viewport are (-1,-1,0), (3.0,-1.0,0.0) and (-1.0,3.0,0.0). The texture coordinates for mapping the generated image of the compute shader to the triangle are also passed, these are (0.0,0.0), (3.0,0.0), and (0.0,3.0)

- Fragment Shader: The output image of the compute pipeline is bound to a texture and then mapped accordingly to the triangle vertices. Since the generated frame matched the size of the viewport in NDC coordinates, a texture wrapping technique known as clamp-to-border [2, 61] is used to properly wrap the triangle with the texture of the ray-traced rendered image.

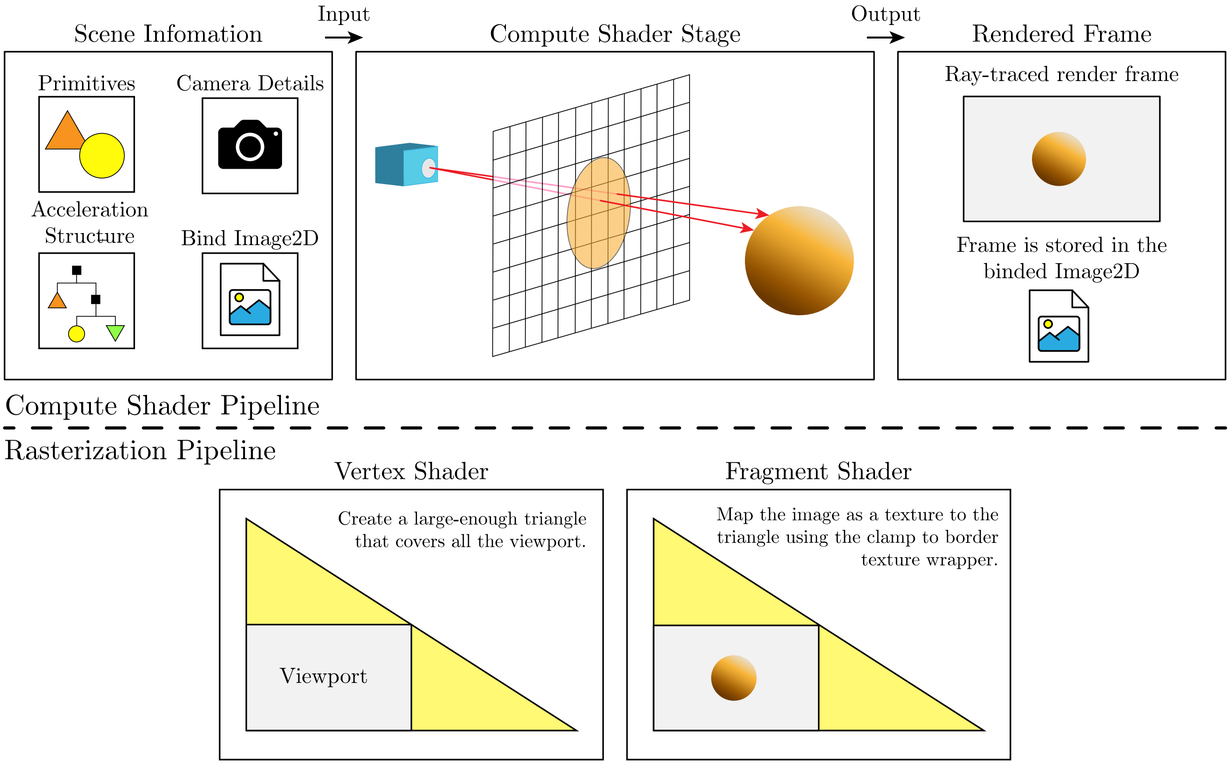


Figure 4.3: Ray tracing pipeline for rendering a scene. The compute shader pipeline is used to render a ray-traced frame. The frame is stored as an image and bound to a texture. Finally, using the rasterization pipeline, the texture containing the rendered image is mapped to a triangle that covers the entire viewport and then displayed on the screen

4.2 Primitive Rendering

4.2.1 Sphere Rendering

The first primitive that is going to be covered is the sphere. Because of its simplicity of definition and implementation, it has become the Hello World of ray tracing by many authors [30, 57, 60, 123]. The implementation by Shirley [30] is the most appealing in terms of how to render a sphere. The equation of a sphere S with radius r and centered at the origin is given by,

$$x^2 + y^2 + z^2 = r^2$$

The previous definition can also be thought of as if a point $\mathbf{p} = (x, y, z)$ lies on the sphere's surface then $x^2+y^2+z^2 = r^2$, it is inside when $x^2+y^2+z^2 < r^2$, or outside if $x^2+y^2+z^2 > r^2$. Following, the general equation of a sphere also considers its center position $\mathbf{c} = (\mathbf{c}_x, \mathbf{c}_y, \mathbf{c}_z)$ becoming,

$$(\mathbf{c}_x - x)^2 + (\mathbf{c}_y - y)^2 + (\mathbf{c}_z - z)^2 = r^2$$

Notice that, the vector from point $\mathbf{p}$ to the center $\mathbf{c}$ is given by $\mathbf{c}-\mathbf{p}$, which allows rewriting the sphere general equation in terms of the dot product of $\mathbf{c} - \mathbf{p}$.

$$(\mathbf{c}_x - x)^2 + (\mathbf{c}_y - y)^2 + (\mathbf{c}_z - z)^2 = (\mathbf{c} - \mathbf{p}) \cdot (\mathbf{c} - \mathbf{p}) = r^2$$

Rendering a sphere requires determining the surface points at which incoming rays intersect it. That is, given a ray $\mathbf{p}(t) = \mathbf{o} + t\boldsymbol{d}$, we want to know whether or not it intersects the sphere in the scene. If it does, then there exists some t for which $\mathbf{p}(t)$ satisfies the sphere equation, and it belongs to the scene. Based on these considerations it follows that:

$$\begin{aligned}
&(\mathbf{c} - \mathbf{p}(t)) \cdot (\mathbf{c} - \mathbf{p}(t)) = r^2 \\
&\rightarrow (\mathbf{c} - (\mathbf{o} + t\boldsymbol{d})) \cdot (\mathbf{c} - (\mathbf{o} + t\boldsymbol{d})) = r^2 \\
&\rightarrow (-t\boldsymbol{d} + (\mathbf{c} - \mathbf{o})) \cdot (-t\boldsymbol{d} + (\mathbf{c} - \mathbf{o})) = r^2 \\
&\rightarrow t^2\boldsymbol{d} \cdot \boldsymbol{d} - 2t\boldsymbol{d} \cdot (\mathbf{c} - \mathbf{o}) + (\mathbf{c} - \mathbf{o}) \cdot (\mathbf{c} - \mathbf{o}) - r^2 = 0
\end{aligned}$$

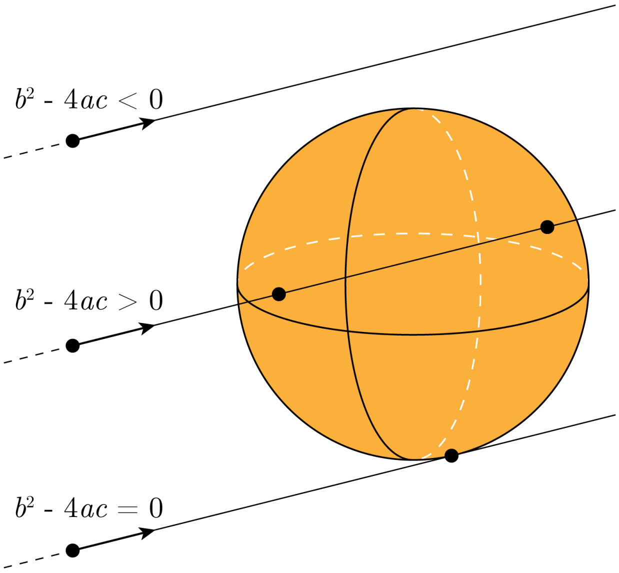


Figure 4.4: Ray-sphere intersection illustrating no solution, one solution, or two solutions.

From the previous result, all the dot products are reduced to scalars and the only unknown term is t. The previous result is a quadratic equation, $ax^2 + bx + c = 0$ that can be solved using the quadratic formula

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where $a = \boldsymbol{d} \cdot \boldsymbol{d}$, $b = -2\boldsymbol{d} \cdot (\mathbf{c} - \mathbf{o})$, and $c = (\mathbf{c} - \mathbf{o}) \cdot (\mathbf{c} - \mathbf{o}) - r^2$. Based on the discriminant $\Delta = b^2 - 4ac$ there could be two real solutions ($\Delta > 0$), one real solution ($\Delta = 0$), or no real solution ($\Delta < 0$) as illustrated in Figure 4.4. An additional step is to simplify the quadratic formula by assuming that $b = -2h$ such that,

$$\begin{aligned}
\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} &= \frac{-(-2h) \pm \sqrt{(-2h)^2 - 4ac}}{2a} \\
&= \frac{2h \pm \sqrt{4h^2 - 4ac}}{2a} \\
&= \frac{2h \pm 2\sqrt{h^2 - ac}}{2a} \\
&= \frac{h \pm \sqrt{h^2 - ac}}{a}.
\end{aligned}$$

From the definition of $b = -2\boldsymbol{d} \cdot (\mathbf{c} - \mathbf{o})$, it follows that $h = -b/2 = \boldsymbol{d} \cdot (\mathbf{c} - \mathbf{o})$. Appendix B presents the code implementation for rendering our first ray-traced sphere.

4.2.2 Triangle Rendering

In modern computer graphics, an object's model is commonly represented by a 3D triangular mesh [2, 57]. The current GPU rendering pipeline (rasterization) is highly optimized to handle triangular meshes efficiently [3, 43, 61]. The reason for using primarily triangles is because of their ease of representation, being the simplest polygon with three vertices and always planar (all three vertices lie on the same plane) [60]. To render our first triangle, an intersection algorithm must be implemented to determine if a ray hits the triangle in the scene.

The algorithm that is going to be implemented was developed by Möller and Trumbore [10] being one of the fastest methods to determine a ray-triangle intersection. Based on [60], given a triangle with vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, a point $\mathbf{q} = (1-t)\mathbf{a} + t\mathbf{b}$ with $0 \leq t \leq 1$ is on the line segment formed by $\mathbf{a}$ and $\mathbf{b}$. Likewise, a point $\mathbf{r} = (1-s)\mathbf{q} + s\mathbf{c} = (1-s)(1-t)\mathbf{a} + (1-s)t\mathbf{b} + s\mathbf{c}$ with $0 \leq s, t \leq 1$ is on the line segment between $\mathbf{q}$ and $\mathbf{c}$. The previous can be defined in terms of a function $\mathbf{T} : [0,1] \times [0,1] \rightarrow R^2$ such that,

$$\mathbf{T}(s,t) = (1-s)(1-t)\mathbf{a} + (1-s)t\mathbf{b} + s\mathbf{c}, \quad 0 \leq s, t \leq 1,$$

and whose image is exactly the triangle formed by the vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. This represents a parameterization of the triangle in terms of s and t. Furthermore, the coefficients of the parameterized triangle have a special property.

$$\begin{aligned}(1-s)(1-t) + (1-s)t + s &= (1-s)((1-t)+t) + s \\ &= (1-s) + s \\ &= 1.\end{aligned}$$

From the above, it follows that combinations of multiple points are defined only when the coefficients sum to 1. Thus, any point lying inside the triangle can be written as

$$\mathbf{p} = u\mathbf{a} + v\mathbf{b} + w\mathbf{c},$$

where the terms u, v, and w are known as the barycentric coordinates of $\mathbf{p}$ regarding the triangle with vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. Moreover, because $u + v + w = 1$ and $u, v, w \geq 0$, then $w = 1 - u - v$, which allows one to rewrite the parameterized triangle function as

$$\begin{aligned}\mathbf{T}(u, v) &= (1 - u - v)\mathbf{a} + u\mathbf{b} + v\mathbf{c} \\ &= \mathbf{a} + u(\mathbf{b} - \mathbf{a}) + v(\mathbf{c} - \mathbf{a})\end{aligned}$$

Rendering a triangle is similar to the sphere case, where the rays that intersect it must be found. Computing the intersection point is done by establishing the equality $\mathbf{p}(t) = \mathbf{T}(u, v)$ such that,

$$\begin{aligned}\mathbf{o} + t\boldsymbol{d} &= \mathbf{a} + u(\mathbf{b} - \mathbf{a}) + v(\mathbf{c} - \mathbf{a}) \\ \mathbf{o} - \mathbf{a} &= -t\boldsymbol{d} + u(\mathbf{b} - \mathbf{a}) + v(\mathbf{c} - \mathbf{a}).\end{aligned}$$

The latter can be expressed in terms of a matrix multiplication where the three known vectors $(-\boldsymbol{d}, \mathbf{b} - \mathbf{a}, \mathbf{c} - \mathbf{a})$ are multiplied by the three unknown scalars (t, u, v) as follows,

$$[-\boldsymbol{d} \quad (\mathbf{b} - \mathbf{a}) \quad (\mathbf{c} - \mathbf{a})] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{o} - \mathbf{a}$$

Möller and Trumbore [10] explain that the term $\mathbf{o} - \mathbf{a}$ translates the triangle to the origin (the first vertex is at the origin). The inverse of the matrix $M = [-\boldsymbol{d}, \mathbf{c} - \mathbf{a}, \mathbf{c} - \mathbf{a}]$, in turn, transforms the triangle into a unit triangle lying in uv-space (See Figure 4.5). Let $\boldsymbol{e}_1 = \mathbf{b} - \mathbf{a}$, $\boldsymbol{e}_2 = \mathbf{c} - \mathbf{a}$, and $\boldsymbol{f} = \mathbf{o} - \mathbf{a}$, then the solution to determine the values of (t, u, v) is obtained by using Cramer's rule [124],

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\boldsymbol{d} \quad \boldsymbol{e}_1 \quad \boldsymbol{e}_2|} \begin{bmatrix} |\ \boldsymbol{f} & \boldsymbol{e}_1 & \boldsymbol{e}_2| \\ |-\boldsymbol{d} & \boldsymbol{f} & \boldsymbol{e}_2| \\ |-\boldsymbol{d} & \boldsymbol{e}_1 & \boldsymbol{f}\ | \end{bmatrix}.$$

Moreover, the determinant of a 3×3 matrix comprised of the column vectors $\boldsymbol{j} = (\boldsymbol{j}_1, \boldsymbol{j}_2, \boldsymbol{j}_3)$, $\boldsymbol{k} = (\boldsymbol{k}_1, \boldsymbol{k}_2, \boldsymbol{k}_3)$, and $\boldsymbol{l} = (\boldsymbol{l}_1, \boldsymbol{l}_2, \boldsymbol{l}_3)$, is given by the scalar triple product $-(\boldsymbol{j} \times$

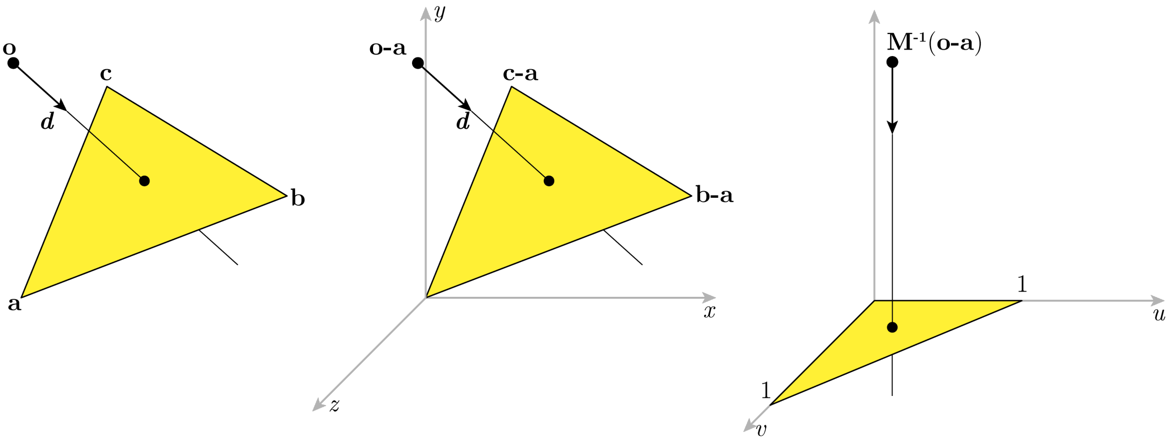


Figure 4.5: Visual representation of Möller and Trumbore algorithm for ray-triangle intersection. Adapted from [10].

$\boldsymbol{l}) \cdot \boldsymbol{k} = -(\boldsymbol{l} \times \boldsymbol{k}) \cdot \boldsymbol{j}$ [125]. Then, the equation to determine (t, u, v) can be rewritten as,

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\boldsymbol{d} \times \boldsymbol{e}_2) \cdot \boldsymbol{e}_1} \begin{bmatrix} (\boldsymbol{f} \times \boldsymbol{e}_1) \cdot \boldsymbol{e}_2 \\ (\boldsymbol{d} \times \boldsymbol{e}_2) \cdot \boldsymbol{f} \\ (\boldsymbol{f} \times \boldsymbol{e}_1) \cdot \boldsymbol{d} \end{bmatrix} = \frac{1}{\boldsymbol{g} \cdot \boldsymbol{e}_1} \begin{bmatrix} \boldsymbol{h} \cdot \boldsymbol{e}_2 \\ \boldsymbol{g} \cdot \boldsymbol{f} \\ \boldsymbol{h} \cdot \boldsymbol{d} \end{bmatrix},$$

where $\boldsymbol{g} = \boldsymbol{d} \times \boldsymbol{e}_2$ and $\boldsymbol{h} = \boldsymbol{f} \times \boldsymbol{e}_1$. These terms are reused to speed up the computation of the triangle intersection. The complete implementation for rendering a ray-traced triangle can be found in the examples directory of the main repository or follow the direct link presented in Appendix C.

4.2.3 Axis-Aligned Bounding Box/Ray Intersection

Boxes are an interesting primitive in ray tracing, defining the surface of a hollow rectangular parallelepiped often referred to as voxels [126]. Moreover, a box could be either a primitive itself having an orientation (OBox) or being axis-aligned (AABox) to the axes of a Cartesian coordinate system. Its second form is contention or bounding, that is, enclosing some other primitive referred to as an oriented bounding box (OBB) or an axis-aligned bounding box (AABB) [127, 128]. The purpose of defining a bounding box will be revealed later, for the moment the ray intersection algorithm will be presented.

The method to determine the intersection of a ray with an AABB is that of the slab introduced by Kay and Kajiya [129]. Following the description presented in [11, 130], an

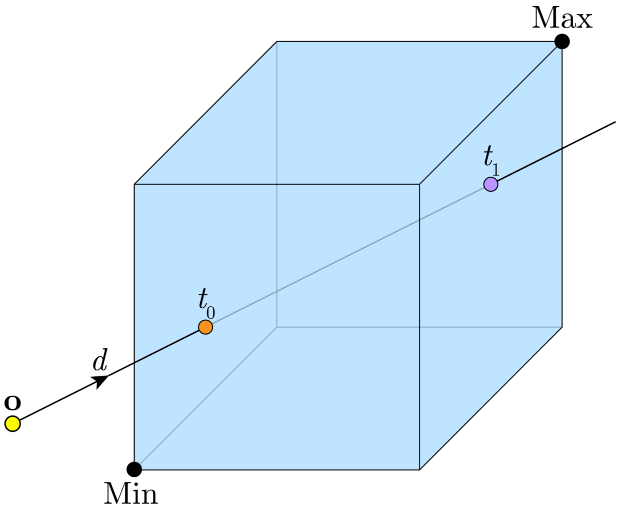


Figure 4.6: Ray-box intersection presenting the two intersections with the AABB. The AABB is represented by its min_B and max_B (B stands for bound) coordinates following a left-hand orientation of the coordinate axes.

$n-$dimensional AABB is just the intersection of n axis-aligned intervals denominated slabs. For instance, the axis-aligned planes x_0 and x_1 enclose the interval $[x_0, x_1]$ with the space between them called a slab. An AABB is usually represented by its bound coordinates $\text{min}_\text{B} = (x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $\text{max}_\text{B}(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$. Figure 4.6 illustrates the intersection of a ray with an AABB.

Given a ray $\mathbf{p}(t) = \mathbf{o} + t\boldsymbol{d}$, the values t_0 and t_1 for which the ray intersects the AABB must be computed. This implies finding the t_0 and t_1 values for each axis-aligned slab. Additionally, the parametric t value at which a ray intersects the plane $ax + by + cz + d = 0$ can be determined by substituting the ray's equation into the plane's equation.

$$\begin{gathered} a(\mathbf{o} + t\boldsymbol{d}) + b(\mathbf{o} + t\boldsymbol{d}) + c(\mathbf{o} + t\boldsymbol{d}) + d = 0 \\ \rightarrow t = \frac{-d - (a, b, c) \cdot \mathbf{o}}{(a, b, c) \cdot \boldsymbol{d}}. \end{gathered}$$

Depending on which axis the plane is aligned on, the other components are canceled. That is, for the plane x_0, $a = 1$ while b and c are just 0, such that:

$$t = \frac{x_0 - \mathbf{o}_x}{\boldsymbol{d}_x},$$

with $d = -x_0$. Then, the vectors t_{Near} and t_{Far} contain the corresponding t_0 and t_1 values for all the axis-aligned slabs.

$$\begin{aligned} t_{\text{Near}} &= \left(\frac{x_{\min} - \mathbf{o}_x}{d_x}, \frac{y_{\min} - \mathbf{o}_y}{d_y}, \frac{z_{\min} - \mathbf{o}_z}{d_z} \right), \\ t_{\text{Far}} &= \left(\frac{x_{\max} - \mathbf{o}_x}{d_x}, \frac{y_{\max} - \mathbf{o}_y}{d_y}, \frac{z_{\max} - \mathbf{o}_z}{d_z} \right). \end{aligned}$$

Subsequently, the $t_{\min}$ and $t_{\max}$ vectors are computed as follows:

$$\begin{aligned} t_{\min} &= \min(t_{\text{Near}}, t_{\text{Far}}) \\ t_{\max} &= \max(t_{\text{Near}}, t_{\text{Far}}). \end{aligned}$$

To handle situations where the ray direction is negative along an axis, we calculate the $t_{\min}$ and $t_{\max}$ vectors. This is because, for a negative direction, the first t value calculated will be larger than the second t value. Finally, the maximum component of $t_{\min}$ and the minimum component of $t_{\max}$ are found, these values being t_0 and t_1 respectively.

$$\begin{aligned} t_0 &= \max(t_{\min}) \\ t_1 &= \min(t_{\max}) \end{aligned}$$

All these computations represent the slab method and allow us to determine if the ray intersects the AABB. Figure 4.7 shows a visual representation of the slab method for a 2D bounding box (BB) for three of the possible cases. In the first case, $t_0 = t_1$, the ray starts by intersecting the lines y_0 and y_1, to then intersect the lines x_0 and x_1. The parametric t values for which the ray intersects the slabs formed by these lines are represented in the intervals $[t_{y0}, t_{y1}]$ and $[t_{x0}, t_{x1}]$ with $t_{y1} = t_{x0}$. Because the direction of the ray is non-negative in neither axis, $t_{\min} = (t_{x0}, t_{y0})$ and $t_{\max} = (t_{x1}, t_{y1})$. Lastly, $t_0 = \max(t_{x0}, t_{y0}) = t_{x0}$ and $t_1 = \min(t_{x1}, t_{y1}) = t_{y1}$, which means that $t_0 = t_1$ and the ray has hit a corner of the bounding box.

In the second case, $t_0 < t_1$, the order in which the ray intersects the lines forming the BB are x_0, y_0, x_1, and y_1. Looking closely, the intervals formed by the parametric t-values of the x and y slabs intersect (see Figure 4.7). The vectors $t_{\min}$ and $t_{\max}$ are (t_{x0}, t_{y0}) and (t_{x1}, t_{y1}) respectively, with $t_0 = \max(t_{\min}) = t_{y0}$ and $t_1 = \min(t_{\max}) = t_{x1}$.

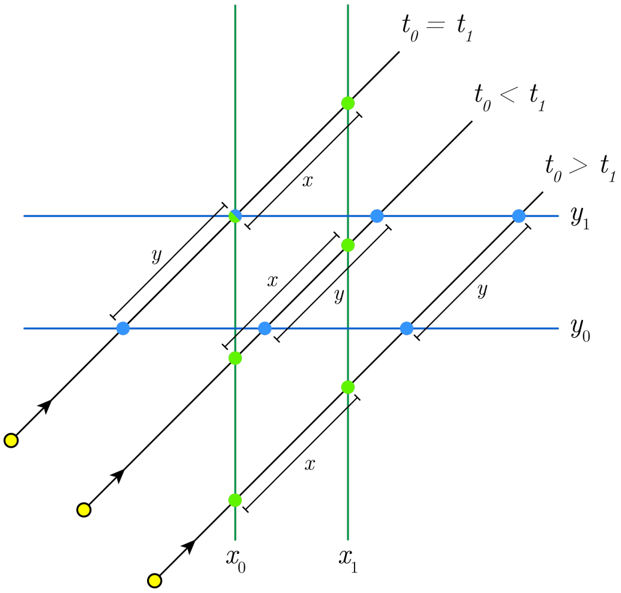


Figure 4.7: Visual representation of the slab method to determine the intersection of a ray with a 2D bounding box (BB). Three cases are presented, a) the ray hits a corner of the BB b) a ray intersects the BB at two points, and c) the ray does not intersect the BB.

As a result, $t_0 < t_1$ and the ray has intersected the BB at two points. Lastly, there is the case in which the ray does not intersect the BB. The ray intersects the x slab within the parametric interval $[t_{x0}, t_{x1}]$ and the y slab in the interval $[t_{y0}, t_{y1}]$. Then, $t_{\min} = (t_x0, t_{y0})$ and $t_{\max} = t_{x1}, t_{y1}$ and consequently $t_0 = t_{y0}$ and $t_1 = t_{x1}$. From Figure 4.7 it is noticed that the t intervals of both slabs do not intersect and more importantly, $t_0 > t_1$.

In summary, the slab method specifies that a ray-box intersection occurs when all slab intervals intersect, that is when $t_0 \leq t_1$. In simpler words, an intersection happens when the maximum entry point (the latest entry into any slab) is less than or equal to the minimum exit point (the earliest exit from any slab).

4.3 Intersection Acceleration

Ray tracing can generate realistic 3D renderings with plenty of details when combined with other techniques. One of these methods, as mentioned by Fujimoto et al. [131], is a global illumination setting rather than local illumination which accounts for multiple light behaviors (will be discussed later). However, they also mention that in a cost/performance comparison ray tracing falls behind traditional rendering methods. Whitted [132] acknowledges this by demonstrating that the most expensive ray tracing operation is ray intersection calculations, which account for 75 percent of total rendering time and reach 95 percent as the complexity of the scene increases. Later on, Rubin and Whitted [133] further explored this aspect by demonstrating that the rendering time of the ray tracing algorithm increased linearly as the number of primitives in the scene increased. At this point, the main drawback of ray tracing is clear, testing if a ray intersects a primitive and which one is closer is computationally too expensive.

Since 1980, considerable research began to be developed to accelerate ray intersection operations and reduce the number of intersection tests. One of the initial proposals was by Rubin and Whitted [133], who introduced an acceleration structure to subdivide the primitives of a scene into a hierarchy of subspaces using rectangular parallelepiped. The top level of this hierarchy is the largest subspace containing all subsequent smaller parallelepipeds. This hierarchical subdivision reduces the number of intersection tests since the ray is only tested with the primitives of the smallest subspaces it intersects. Kay and Kajiya [129] improved on this idea by introducing bounding volume hierarchies (BVH), in which the scene primitives are enclosed in a volume consisting of 7 slabs. These are then grouped into a larger bounding volume based on proximity, forming a hierarchical tree structure. Consequently, the number of intersection tests is reduced, since the ray is only tested with the primitives of the leaf nodes and will stop if it fails to intersect the bounding volume of the interior nodes.

Currently, the landscape of acceleration structures for ray tracing is divided into two main approaches: spatial subdivision and primitive subdivision [11]. Spatial subdivision methods divide the space surrounding an object into smaller regions using planes and assign its primitives to the subspaces they overlap with. Testing for a ray-primitive intersection is only done against the primitives of those subspaces the ray passes through. Some methods under this category include uniform grids [131], octrees [134], kd-trees [74], binary space partition (BSP) trees [135], and their different variations. Primitive subdivision, on the other hand, comprises methods that subdivide the primitives of a scene into bounding volumes (e.g., spheres, AABB, OBB, etc.), with the BVH algorithm being a clear representative of this category. Meister et al. [76] provide a comprehensive survey on BVHs for ray tracing including their different variants.

Over the years, the two most used acceleration structures have been ADAPTIVE kd-trees and BVHs. As a result, an ongoing discussion over which method is better has developed with different works discussing the matter [34, 75, 136]. Meister et al. [76], however, highlight four important characteristics of BVHs. These include a predictable memory footprint, robust and efficient querying of branch intersections, scalable construction algorithms that enable fast, highly optimized, or hybrid BVHs, and support for dynamic geometry in animated scenes, leveraging their fast construction algorithms. Furthermore, NVIDIA RTX GPUs are highly optimized for BVH traversal by making use of its RT (Ray Tracing) cores [137]. Likewise, Vulkan and DirectX12 use BVHs as their acceleration structures in their ray-tracing pipelines [67, 68].

4.4 Bounding Volume Hierarchies

After the previous introduction, the acceleration structure selected for this work's ray-tracing renderer was the bounding volume hierarchy (BVH). The implementation follows mainly the description provided by Pharr et al. [11] and Meister et al. [76] while also recurring to other authors for some specific explanations and details.

As discussed earlier, a BVH is a primitive subdivision algorithm that reduces the number of intersection tests by dividing primitives into a tree hierarchy of disjoint sets. Figure 4.8 shows both the scene's primitive subdivision into bounding volumes and its resultant tree hierarchy. Primitives are located at the leaf nodes of the tree. The root node represents the largest bounding volume that encompasses the entire scene, while the internal nodes represent intermediate bounding volumes that enclose their respective child nodes. Testing for a ray-primitive intersection requires checking against the bounding volumes at the internal nodes. Consequently, if a ray does not intersect an internal node, the subtree below it is omitted reducing the number of intersection tests.

An important characteristic of BVHs is that primitives are only instantiated once, that is, there are no duplicates as in other acceleration structure methods [11]. In consequence, its memory space requirements are bounded, needing a maximum of $2n-1$ nodes, n is the number of primitives, to represent a binary tree hierarchy. The number of leaf and internal nodes is n and $n-1$ respectively. Furthermore, the number of total nodes can be reduced

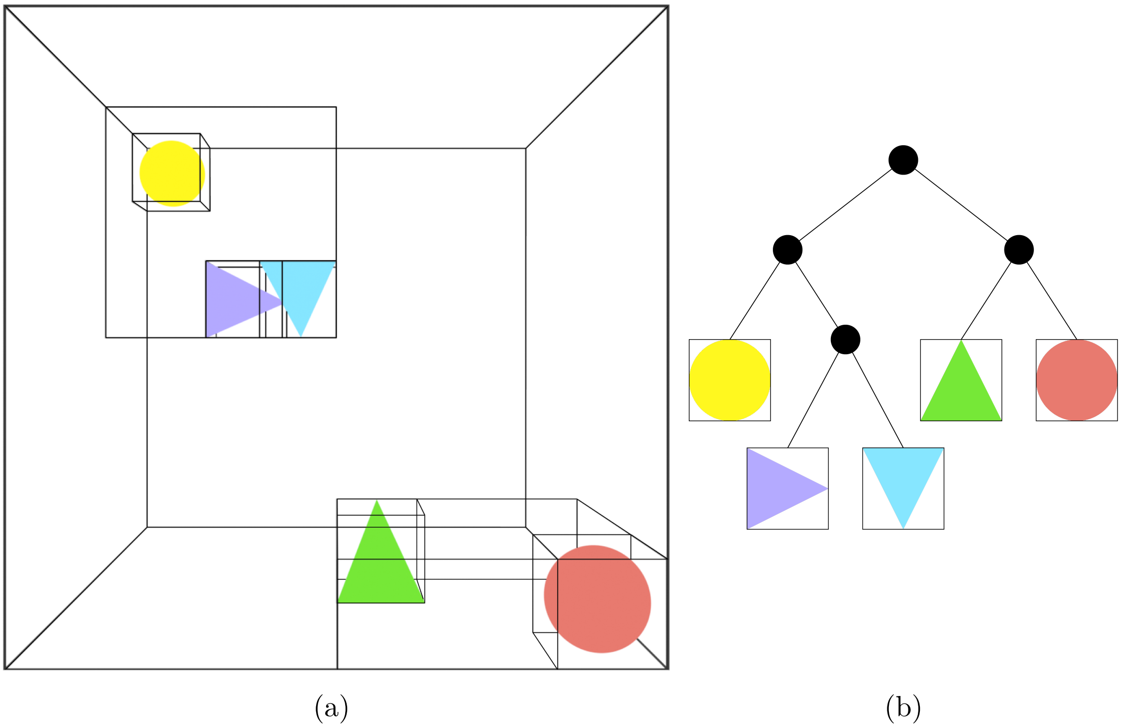

(a) (b)

Figure 4.8: Bounding Volume Hierarchy for a scene comprised of two spheres and three triangles (a) For each primitive, a bounding box is computed and then these are added together based on proximity forming a larger bounding volume. The largest bounding volume encloses all other bounding volumes of the scene. The example also shows the intersection of two bounding volumes enclosing the purple and cyan triangles. (b) The bounding volume tree hierarchy with the root node representing the largest bounding volume, followed by the internal nodes, and finally the leaf nodes containing the primitives.

if multiple primitives are stored at the leaf nodes.

4.4.1 BVH Construction

Meister et al. [76] mention two approaches for BVH construction, top-down and bottom-up. In the top-down approach, the root node containing all the primitives is divided into two subsets, which are then assigned to the node's two children respectively. This process is repeated recursively until the termination conditions are met, these usually are a maximum tree depth or a maximum number of primitives per node. The bottom-up method is rather considered a clustering algorithm. Initially, each primitive's bounding volume is treated as an individual cluster. The clusters are then merged by proximity into a larger bounding volume whose surface area is the minimum to enclose them.

The implementation given by Pharr et al. [11] follows a top-down construction inspired in turn by Wald [81] and Gúnther et al. [34]. In summary, a top-down construction has three steps: 1) computing the AABB for each primitive, 2) building the tree hierarchy using a split function, and 3) converting the tree into an array-based representation that is easier to traverse and does not require recursion. However, steps two and three can be merged into one, directly computing an array representation of the three as done in [138].

4.4.2 Primitives Bounding Boxes

Computing a bounding box depends on the complexity of the primitive, for a sphere or a triangle it is straightforward. Recalling from earlier, an AABB can be represented by its min_B and max_B bound coordinates. As such, the AABB that tightly encloses a sphere with center $\mathbf{c} = (\mathbf{c}_x, \mathbf{c}_y, \mathbf{c}_z)$ and radius r is given by:

$$\text{Sphere}_\text{AABB} : \begin{array}{l} \text{min}_\text{B} = (\mathbf{c}_x - r, \mathbf{c}_y - r, \mathbf{c}_z - r) \\ \text{max}_\text{B} = (\mathbf{c}_x + r, \mathbf{c}_y + r, \mathbf{c}_z + r). \end{array}$$

In turn, for a triangle with vertices $\mathbf{a} = (\mathbf{a}_x, \mathbf{a}_y, \mathbf{a}_z)$, $\mathbf{b} = (\mathbf{b}_x, \mathbf{b}_y, \mathbf{b}_z)$, $\mathbf{c} = (\mathbf{c}_x, \mathbf{c}_y, \mathbf{c}_z)$ the AABB is computed using the component-wise min and max operations as follows,

$$\text{Triangle}_\text{AABB} : \begin{array}{l} \text{min}_\text{B} = \min(\min(\mathbf{a}, \mathbf{b}), \mathbf{c}) \\ \text{max}_\text{B} = \max(\max(\mathbf{a}, \mathbf{b}), \mathbf{c}). \end{array}$$

4.4.3 Split Methods to Construct a BVH

As discussed in [11, 76], there are three methods on how to partition a BVH tree node, spatial median split, object median split, or a cost-based function split. The spatial median split partitions the bounding box along a specific axis assigning the primitives of each side to the corresponding left or right child node. In contrast, the object median split sorts the primitives along an axis and then partitions them into two children nodes with approximately the same number of primitives. This method is useful when all primitives lie on the same side of the spatial splitting plane or all its centroids are located at the same point. Pharr et al. [11] explain both of these methods more in-depth providing their corresponding implementations.

The two methods covered depend largely on the scene configuration and often generate partitions that are not optimal [11]. As a result, more nodes are visited having a direct impact on time traversal as more intersection tests are performed. Goldsmith and Salomon [139] proposed a heuristic approach where the probability of a ray intersecting a bounding volume depends on its surface area. MacDonald and Booth [140] later formalized the method known as the surface area heuristic (SAH). The SAH estimates the performance of an acceleration structure based on the cost traversal of internal nodes, leaf nodes, and primitive nodes.

Meister et al. [76] mention that the cost of traversing a BVH is estimated based on the number of operations required to find the first ray intersection. Given the BVH tree with root N, its cost function is defined by the recurrence equation:

$$c(N) = \begin{cases} c_T + \sum\limits_{N_c} \mathrm{P}(N_c|N)c(N_c) & \text{if } N \text{ is an interior node} \\ c_I|N| & \text{otherwise} \end{cases}$$

where $c(N)$ is the cost of traversing the subtree with root node N, c_T is the cost of traversing an internal node, N_c is a child of node N, $\mathrm{P}(N_c/N)$ is the conditional probability of intersecting node N_c after having intersected node N, and $|N|$ is the number of primitives contained in the subtree with root node N. The values of c_T and c_I are implementation-specific constants and express the average cost of ray-primitive intersection for a traversal step [76, 81].

Furthermore, based on [11], the probability of traversing a child node is computed using geometric probability notions [141]. Specifically, given the convex volume N of the parent node, which encloses the convex volume N_L of its left child node, the conditional probability [142] of the ray intersecting the N_L node after having intersected the N node is given by the ratio of their surface areas (SA).

$$\mathrm{P}(N_L|N) = \frac{\mathrm{SA}(N_L)}{\mathrm{SA}(N)}$$

where $\mathrm{SA} = 2 \times (w \times h + w \times d + h \times d)$ of the bounding volume. Replacing this result into the recurrence equation and after unrolling it (finding its closed form), the following is obtained,

$$c(N) = c_T \sum_{N_i} \frac{\mathrm{SA}(N_i)}{\mathrm{SA}(N)} + c_I \sum_{N_l} \frac{\mathrm{SA}(N_l)|N|}{\mathrm{SA}(N)}$$

where N_i and N_l represent the interior and leaf nodes of a subtree with root node N. Constructing an efficient BVH involves minimizing its cost of traversal. Unfortunately, this translates into a global optimization problem to find an optimal node topology with most of the algorithms being exponential regarding the number of primitives [143]. Wald et al. [144], however, proposed a local greedy approximation for a recursive top-down construction that yields excellent results. An explanation follows based on the description provided by Pharr et al. [11].

The SAH cost function assumes that at any time instance, a leaf node could be created for the current space region and its primitives. Therefore, if a ray traverses this region, an intersection test will be performed for all its primitives having a cost of

$$\sum_{i=1}^{n} t_{\mathrm{isect}}(i)$$

with n being the total number of primitives and $t_{\mathrm{isect}}(i)$ the time required to test for ray intersection with the i-th primitive. An alternative is to partition the region in two, where testing for a ray intersection has a cost of

$$c(L, R) = t_{\mathrm{trav}} + \mathrm{P}(L|N) \sum_{l=1}^{|L|} t_{\mathrm{isect}}(l) + \mathrm{P}(R|N) \sum_{r=1}^{|R|} t_{\mathrm{isect}}(r),$$

where t_{trav} is the time of traversing an internal node and determining which child node the ray intersects, $\mathrm{P}(L|N)$ and $\mathrm{P}(R|N)$ are the conditional probabilities oh intersecting a child node after intersection the parent node N, r_i and l_i are the primitive indices in each child respectively, and $|L|$ and $|R|$ the total number of primitives of each child. For a given subtree, a collection of n primitives can be split between the two child nodes in $2^n - 2$ possible ways excluding the empty cases. Therefore, the procedure is to partition each node of the BVH tree recursively, starting with the largest bounding volume, into two child nodes while seeking the partition at the minimum cost.

An observation in BVH construction is that primitives are usually partitioned by an axis-aligned plane across one of the three axes (x, y, or z). Furthermore, evaluating the minimum cost of the $|N| - 1$ partition planes separating each primitive along an axis may still be costly, specifically at the largest bounding volume nodes that contain a larger amount of primitives [76]. As a result, Wald et al. [144] proposed a method known as binning, in which the axis is divided into b equally-extended bins to which primitive centroids are projected. If a centroid lies in the partition plane of one bin, the primitive is assigned to the next bin. The cost function is then evaluated only at the splitting planes of each bin. Finally, the possible stop conditions are when $|N| < 2$ or when the minimum cost partition $c_{min}(L, R)$ is higher than creating a leaf with cost $\sum_{i=1}^{|N|} t_{\text{isect}}(n_i)$.

The BVH tree implementation of this work follows the top-down approach using the SAH heuristic and the binning partitioning scheme. The code for constructing the BVH acceleration structure is located within the `scene.c` file in the main repository. Follow the link provided in Appendix D for more details. It should be noted that the costs of t_{trav} and t_{isect} were set based on the results of Pharr et al. [11]. In this way, t_{trav} was set to 0.5, and $t_{\text{isect}(i)}$ was set to 1.0 for all primitives based on the notion that the cost of traversing a node, testing for AABB intersection, is less costly than testing for a primitive intersection.

4.5 The Global Illumination Problem

All the previous methods and code implementations presented earlier are the basics of ray tracing and are often encompassed under ray casting. Ray casting focuses mainly on determining object visibility across a pixel rather than global illumination [132]. However, what is global illumination and how is it related to 3D rendering? In 3D rendering, a scene is comprised of different objects, each having a size, location, and material, as well as the position and characteristics of light sources. Creating a photorealistic render of a scene should account for the behavior of light and its effect on the materials that comprise an object's surface [14].

Light propagates into a scene, reflecting, refracting, casting shadows, and scattering. Global illumination accounts for all the light bouncing off objects and indirectly illuminates other objects [14, 145]. For example, the sun's rays can refract through a window onto a kitchen countertop and then reflect off the cabinets or wall. These interactions may also cast a shadow behind a flower pot on the countertop, or the light may refract through the flower pot if it is made of glass, illuminating other objects as well. Therefore, all these objects are not illuminated directly by a light source, but indirectly by the multiple paths that light rays can follow [146]. In summary, global illumination considers both direct and indirect illumination across the scene.

Computing global illumination, however, is a complicated task that is solved by combining physics and statistics. The following sections are devoted to presenting the theory and notions used to solve this problem.

4.6 Radiometry

Global illumination algorithms aim to determine the steady-state distribution of light propagating in a scene, that is, the balance between direct and indirect light interactions [14]. The physical quantities needed for these calculations come from radiometry. Radiometry is the area of study concerned with the measurement of electromagnetic radiation, including light [2]. Photometry, in turn, is responsible for quantifying the perception of light energy (based on the human's visual system) as summary values obtained from weighted sums of radiometric measurements [14, 60]. Some of the mathematical and radiometric concepts required are discussed below.

- **Solid Angle**

  Whereas in 2D the angular extent is simply the angle between two directions with the same origin, the solid angle is the angular extent representation in 3D. As such, the solid angle is measured as the area A of a patch projected onto a sphere, divided by the squared radius of the sphere [147].

  $$\omega = \frac{A}{r^2}$$

  Similar to its 2D counterpart, the solid angle is unitless, but the term steradians (sr) is employed to differentiate it from other unitless metrics. The differential solid angle can be derived from Figure 4.9, in which by fixing the angle θ and varying φ, a circle of radius $r \sin\theta$ is traced. Noticed that for an infinitesimal variation of $rd\varphi$, the circular arc length is $r \sin\theta d\varphi$. Then, by considering an infinitesimal small variation

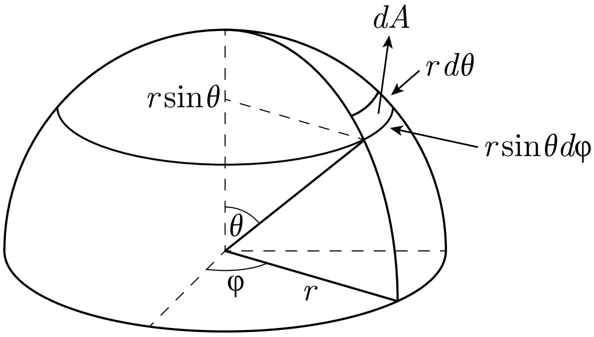


Figure 4.9: Representation of the differential solid angle for a sphere of radius r.

$rd\varphi$, the area formed by these variations of the angles can be expressed as,

$$dA = (r\sin\theta d\varphi)(rd\theta) = r^2 \sin\theta d\theta d\varphi$$

where dA denotes the differential area of an infinitesimal small patch projected into the sphere. Considering the definition of solid angle, the differential solid angle can be expressed as

$$d\omega = \frac{dA}{r^2} = \sin\theta d\theta d\varphi.$$

Computing the solid angle involves integrating the differential solid angle over a region of the unit sphere. Therefore, by integrating the differential solid angle over the complete hemisphere, that is $0 \leq \theta \leq \pi$ and $0 \leq \varphi \leq 2\pi$ it follows that,

$$\begin{aligned}
\omega &= \int_0^{2\pi}\int_0^{\pi} \sin\theta d\theta d\varphi \\
&= -\int_0^{2\pi} [\cos\theta]\Big|_0^{\pi} d\varphi = -\int_0^{2\pi}(-1-1)d\varphi \\
&= 2\int_0^{2\pi} d\varphi = 2[\varphi]\Big|_0^{2\pi} = 2(2\pi - 0) \\
&= 4\pi sr
\end{aligned}$$

where $4\pi sr$ represents the total solid angle of a unit sphere.

- **Radiant Energy**

Radiant energy can be understood as the amount of energy carried by photons emitted from a light source t [11, 147]. It is denoted by $Q(t)$ and is measured in Joules

(J). In this regard, a photon of wavelength λ carries energy,

$$Q = \frac{hc}{\lambda},$$

where c represents the speed of light $299,472,458\ m/s$, and h corresponds to Planck's constant, $h \approx 6.626 \times 10^{-34}\ m^2 \cdot kg \cdot s^{-1}$ [45].

- **Radiant Flux**

  Radiant flux or radiant power, denoted as $\Phi(t)$ represents the total amount of energy reflected/absorbed by unit time and is measured in watts (W) ($1W = 1J/s$) [11, 147]. Radiant flux is represented as

  $$\Phi(t) = \frac{dQ(t)}{dt}.$$

  Moreover, integrating radiant flux over a range of time gives the total radiant energy.

  $$Q = \int_{t_0}^{t_1} \Phi(t)dt.$$

- **Radiant Intensity**

  For a point light source centered at the unit sphere (r=1), the amount of radiant flux reflected per solid angle is referred to as radiant intensity (I) [11, 14, 147], Figure 4.10 presents an illustration of this concept. As such, radiant intensity is represented as

  $$I = \frac{d\Phi}{d\omega}$$

  with its units being (W/sr). Note that for the hemisphere of directions, the radiant intensity is just,

  $$I = \frac{\Phi}{4\pi}.$$

- **Irradiance**

  Irradiance is the incident radiant flux as a function of surface position $\mathbf{p}$ and is denoted as $E(\mathbf{p})$. In other words, irradiance is understood as the amount of light

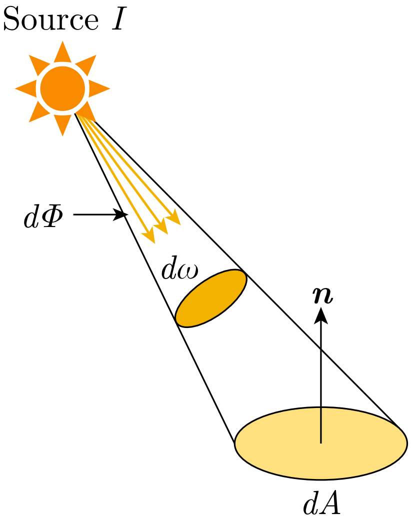


Figure 4.10: Radiant intensity (I) of a light source.

received per unit surface area [2, 14, 147]. As such, it is calculated as the derivative of radiant flux regarding a differential surface area, such that,

$$E(\mathbf{p}) = \frac{d\Phi(\mathbf{p})}{dA},$$

with its units being (W/m^2). Moreover, when integrating irradiance over a differential surface area centered at $\mathbf{p}$, the radiant power is obtained.

$$\Phi = \int_A E(\mathbf{p})dA.$$

Consider the example presented in Figure 4.11 in which a light source is located at a point $\mathbf{l}$ emitting I watts per steradian in all directions. The goal is to compute the irradiance at a surface patch located at $\mathbf{p}$. The angle θ formed between the normal $\boldsymbol{n}$ of the surface patch and the direction vector of the light source is $\boldsymbol{l}' = \mathbf{l} - \mathbf{p}$, with its length being $r = ||\boldsymbol{l}'||$. However, if the surface normal does not point at $\mathbf{q}$, the notion of foreshortening must be considered. Foreshortening represents the reduction of the projected area as seen from a particular point. Notice that as the differential surface normal $\boldsymbol{n}$ rotates away from the normalized direction vector $\boldsymbol{l}'$, the surface patch as observed from point q becomes smaller. That is, the solid angle is projected into a smaller area of a sphere centered at $\mathbf{q}$ and radius $r = ||\boldsymbol{l}'||$, with the projected area

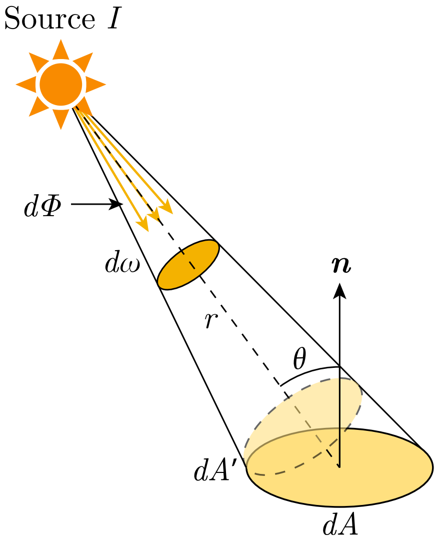


Figure 4.11: Irradiance incident to an infinitesimal patch S with area dA.

said to be foreshortened. Therefore, for differential surface area dA with a normal $\boldsymbol{n}$ rotated an angle θ away from $\boldsymbol{l}'$, the solid angle becomes

$$d\omega = \frac{dA\cos\theta}{r^2}$$

where $dA' = dA\cos\theta$ represents the foreshortened differential area. In this regard, from the definition of radiant intensity and the notion of foreshortened area, the surface patch located at $\mathbf{p}$ receives an irradiance of

$$E(\mathbf{p}) = \frac{d\Phi(\mathbf{p})}{dA} = \frac{I d\omega}{dA} = \frac{I}{dA}\frac{dA\cos\theta}{r^2} = \frac{I\cos\theta}{r^2}.$$

From this result, it is observed that the illumination received by a surface, its irradiance, is inversely proportional to the square of the distance from the light source. In other words, as the distance between the light source and the surface patch increases, the irradiance decreases. Likewise, as the normal of the surface patch rotates away from the direction vector bml', the radiant intensity at the surface will be smaller. Therefore, when the surface patch is oriented perpendicular to the direction of the light source, the maximum irradiance is obtained. As the surface patch inclines towards the grazing angle, the irradiance decreases until it reaches zero.

- **Radiant exitance**

Radiant exitance or radiosity (B), is the exiting radiant power per unit surface area [14]. In this regard, light received at the surface of an object is associated with irradiance, and light reflected from a surface, as a function of surface position $\mathbf{p}$, is linked to radiant exitance. As such,

$$B(\mathbf{p}) = \frac{d\Phi(\mathbf{p})}{dA}.$$

- **Radiance**

Irradiance and radiant exitance determine the differential radiant power per differential surface area at a point $\mathbf{p}$. Nonetheless, these quantities do not consider the directional distribution of power. Consequently, the radiometric quantity that measures irradiance or radiant exitance for a differential solid angle is known as radiance (L). As such, radiance is a measure of incident/reflective radiant power by a surface in a particular direction. In this regard, it is a function of position $\mathbf{p}$ and a normalized direction vector $\boldsymbol{d}$, denoted as $L(\mathbf{p}, \boldsymbol{d})$. An important consideration is that radiance is defined for a surface perpendicular to the normalized direction of incidence/reflection $\boldsymbol{d}_i/\boldsymbol{d}_o$ and can be derived from irradiance or radiant exitance as follows,

$$\begin{aligned} L(\mathbf{p}, \boldsymbol{d}_i) &= \frac{dE(\mathbf{p})}{\cos\theta_i d\omega}, \quad \cos\theta_i = \boldsymbol{n} \cdot \boldsymbol{d}i \\ L(\mathbf{p}, \boldsymbol{d}_o) &= \frac{dB(\mathbf{p})}{\cos\theta_o d\omega}, \quad \cos\theta_o = \boldsymbol{n} \cdot \boldsymbol{d}_o. \end{aligned}$$

Another notation will be $L_i(\mathbf{p}, \boldsymbol{d}_i)$ and $L_o(\mathbf{p}, \boldsymbol{d}_o)$ to denote radiance due to incident or reflective radiant power respectively. In terms of flux, radiance is defined as radiant power per unit foreshortened surface area per unit solid angle ($W \cdot m^{-2} \cdot sr^{-1}$) [14, 147], such that,

$$L(\mathbf{p}, \boldsymbol{d}_o) = \frac{d^2\Phi}{dA \cos\theta d\omega}, \quad \cos\theta = \boldsymbol{n} \cdot \boldsymbol{d}_o.$$

Likewise, radiant exitance represents the integral of surface radiance across the hemi-

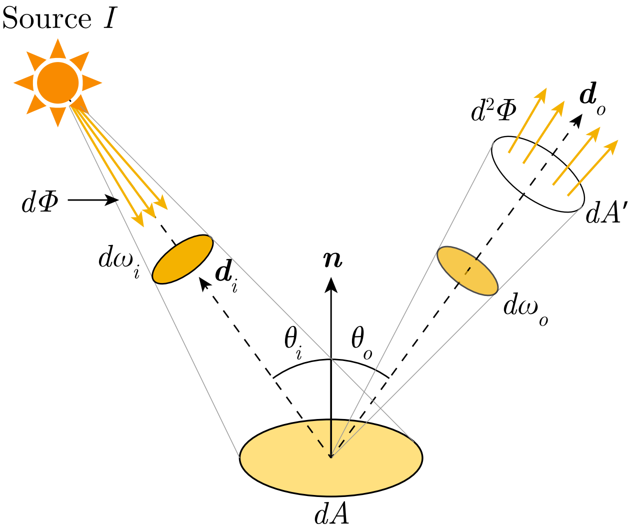


Figure 4.12: Radiance representation in terms of radiant flux defined as the radiant power per unit foreshortened surface area per unit solid angle along the direction of reflection $\boldsymbol{d}_o$

sphere of reflective directions. The same follows for irradiance, representing the integral of surface radiance across all the incident directions. Therefore, it follows that,

$$
\begin{aligned}
B(\mathbf{p}) &= \int_{\boldsymbol{d}_o \in \Omega} L(\mathbf{p}, \boldsymbol{d}_o) \cos\theta_o d\omega, \\
E(\mathbf{p}) &= \int_{\boldsymbol{d}_i \in \Omega} L(\mathbf{p}, \boldsymbol{d}_i) \cos\theta_i d\omega,
\end{aligned}
$$

where Ω represents the hemispheres of either reflective or incident directions around point $\mathbf{p}$. Finally, the radiant power can also be integrated from radiance as follows,

$$
\Phi = \int_A \int_{\boldsymbol{d}_o \in \Omega} L(\mathbf{p}, \boldsymbol{d}_o) \cos\theta_o d\omega dA.
$$

Among all radiometric quantities, radiance becomes essential in rendering. The color visible through a pixel is directly proportional to the radiance at the surface point visible through that pixel.

4.6.1 Light Source Representation

Representing light sources in a scene requires defining their attributes, primarily the shape. In this sense, there are several models of light sources, such as point, distant (or directional), spotlight, area, and environmental, among others [11]. Perhaps the most known model is the point light source, represented by a 3D point in space that illuminates in all directions [2, 60]. A distant light could be understood as a point light located at infinity and is characterized by all its rays being parallel, meaning, it has no position but rather a direction [2, 11]. A clear example is the Sun, which although it is not located at infinity, its light rays reach the Earth as if they were parallel due to their great distance, appearing to have the same direction.

Spotlights are a variation of point lights, emitting light in a cone of directions originating at a point [2, 11]. These are represented by an origin (position of the light source) and a direction called the spot direction [2, 12]. Area lights, in turn, could be represented by polygons, polygon meshes, or the surface area of volumes such as spheres or cylinders [11, 12, 60]. Finally, infinite area lights encompass light sources that surround the entire scene [11]. These are especially useful for simulating the lighting of an environment, such as the sky, casting light into the scene from every direction. A derivation of infinite area lights is known as image infinite lights, or more broadly, image-based lighting [11, 148]. In this scenario, the objects in a scene are illuminated by the incoming light of images mapped to a volume, such as a sphere or cube, covering the scene.

4.6.2 Light-Surface Interactions

Surface materials interact with light in different ways, some are mirror-like while others are diffuse (scatter light in different directions) [14]. The material's color depends largely on which wavelengths it absorbs and which wavelengths it reflects [147]. Generally speaking, light can be absorbed by a surface with an angle ψ at a point $\mathbf{p}$ and can exit at a position $q \neq p$ with an angle θ [14]. A function describing this behavior of light is known as bidirectional surface scattering reflectance distribution function or BSSRDF [11]. Materials can also emit light at different wavelengths and after a time period, accounting for two phenomena known as fluorescence and phosphorescence respectively [14].

Assuming that light is reflected, exits at the same point of incidence, with the same wavelength, and at the same time, allows for the definition of a simpler function known as a bidirectional reflectance distribution function (BRDF) [11, 14, 147]. If the light is instead refracted by the material the function is known as bidirectional transmission distribution function (BTDF) [11, 60].

4.6.3 Reflectance and BRDF

Ganovelli et al. [12] provide an important connotation, explaining that surface reflection can be described with two properties, reflectance or a BRDF. Hemispherical surface reflectance or simply reflectance, denoted as ρ, determines the ratio of reflected radiant power to incident radiant power, Φ_o/Φ_i, or also as the ratio of radiant exitance to incident irradiance, B/E. However, reflectance is direction-independent, that is, it does not consider the incident direction of irradiance nor the reflected direction of radiant exitance. In consequence, this property is only useful for diffuse materials.

Directional hemispherical reflectance, on the other hand, takes into account materials that reflect light as a function of incident direction. In this regard, to differentiate it from reflectance, it is denoted with the same symbol ρ but as a function of an incident direction $\boldsymbol{d}_i$ such that,

$$\rho(\boldsymbol{d}_i) = \frac{B}{E_{\boldsymbol{d}_i}}.$$

Finally, there are materials whose reflected radiance is not uniform across the hemisphere of directions Ω_o and is dependent on the incident direction of irradiance. As such the function that describes this surface is bidirectional regarding the incident and reflected directions. This function is formally known as the bidirectional reflectance distribution function (BRDF). Formally, a BRDF at a point $\mathbf{p}$ is defined as the ratio of differential radiance reflected towards an exciting direction $\boldsymbol{d}_o$ regarding the differential irradiance incident to the differential surface area at $\mathbf{p}$ through a differential solid angle with direction $\boldsymbol{d}_i$.

$$\text{BRDF: } f_r(\mathbf{p}, \boldsymbol{d}_i, \boldsymbol{d}_o) = \frac{dL(\mathbf{p}, \boldsymbol{d}_o)}{dE_{\boldsymbol{d}_i}(\mathbf{p})}$$

Based on the relation between differential irradiance and reflected radiance, that is,

$$dE_{\boldsymbol{d}_i}(\mathbf{p}) = L(\mathbf{p}, \boldsymbol{d}_i) \cos\theta_i d\omega,$$

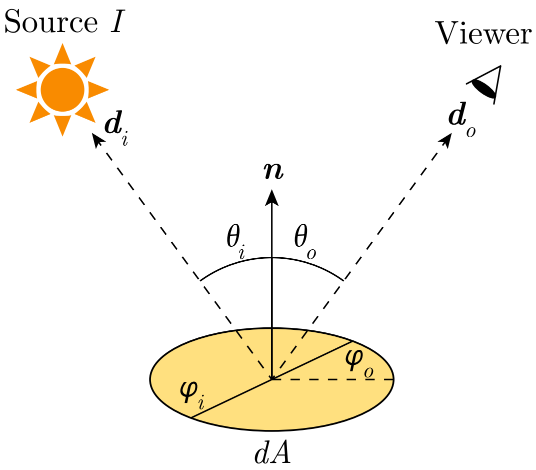


Figure 4.13: Bidirectional Reflectance Distribution Function (BRDF) formed by the zenith angles, θ_i and θ_o, and the azimuth angles, θ_i and θ_o. A BRDF f_r is said to be 4-dimensional, but can also be represented in terms of the direction vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_o$, such that $fr(\theta_i, \theta_o, \theta_i, \theta_0) = fr(\boldsymbol{d}_o, \boldsymbol{d}_i)$[11, 12].

the formal definition of a BRDF can be rewritten in terms of radiance only,

$$f_r(\mathbf{p}, \boldsymbol{d}_i, \boldsymbol{d}_o) = \frac{dL(\mathbf{p}, \boldsymbol{d}_o)}{L(\mathbf{p}, \boldsymbol{d}_i)\cos\theta_i d\omega}.$$

A relation between directional reflectance and a BRDF can be derived based on its formal definitions and previous radiometric quantities; consider the differential surface area at point $\mathbf{p}$ implicit.

$$\begin{aligned}
\rho(\boldsymbol{d}_i) &= \frac{B}{E_{\boldsymbol{d}_i}} \\
&= \frac{\displaystyle\int_{\boldsymbol{d}_o \in \Omega} L(\boldsymbol{d}_o)\cos\theta_o d\omega}{E_{\boldsymbol{d}_i}} \\
&= \frac{E_{\boldsymbol{d}_i}\displaystyle\int_{\boldsymbol{d}_o \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o)\cos\theta_o d\omega}{E_{\boldsymbol{d}_i}} \\
&= \int_{\boldsymbol{d}_o \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o)\cos\theta_o d\omega.
\end{aligned}$$

Continuing, BRDFs that take into consideration the physical properties of energy are known as physically-based. In this regard, physically-based BRDFs comply with two properties, reciprocity, and energy conservation [11]. Reciprocity imposes that the BRDF value is independent of the incident and emitting directions, meaning that if the direction vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_o$ are switched the BRDF value remains the same. This property is formally known as Helmholtz reciprocity and is denoted as

$$f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) = f_r(\boldsymbol{d}_o, \boldsymbol{d}_i).$$

Second, the law of conservation of energy states that energy can neither be created nor destroyed, but only transformed from one form to another [45]. Accordingly, a physically-based BRDF ensures that the radiant power reflected over all directions is less than or equal to the radiant power incident to the surface [11]. The formal concept as explained in [14] can be expressed as

$$\int_{\boldsymbol{d}_o \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) \cos\theta_o dw \leq 1.$$

4.6.4 Diffuse and Specular Reflections

For a given irradiance distribution, a diffuse material's reflected radiance is independent of the reflected direction [14, 147]. That is, the reflected radiance of a diffuse material is constant, $f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) = f_r =$ constant, in all directions having a matte aspect [147]. Purely diffuse materials follow Lambert's cosine law, which states that the reflected radiance of a diffuse surface is directly proportional to the cosine of the angle between the surface normal and the direction of the incident light [12]. Recall from earlier, that materials whose reflections are direction-independent are properly explained by the reflectance ρ property. Taking this into account and building on the directional reflectance BRDF, it

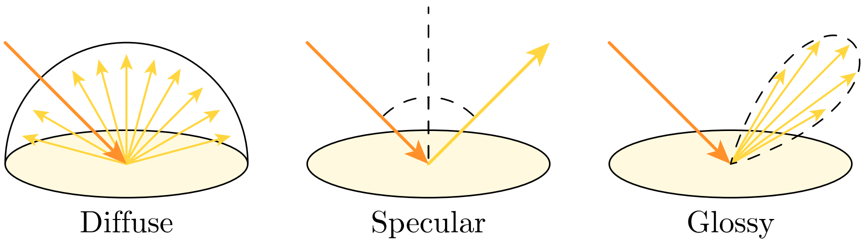


Figure 4.14: Diffuse, specular, and glossy reflections.

follows that,

$$
\begin{aligned}
\rho = \rho(\boldsymbol{d}_i) &= \int_{\boldsymbol{d}_o \in \Omega_o} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) \cos_o \theta d\omega = \int_{\Omega} f_r \cos \theta d\omega \\
&= f_r \int_0^{2\pi} \int_0^{\pi/2} \cos \theta \sin \theta d\theta d\varphi = f_r \int_0^{2\pi} d\varphi \cdot \int_0^{\pi/2} \cos \theta \sin \theta d\theta \\
&= f_r \int_0^{2\pi} d\varphi \cdot \int_0^{1} u du, \quad u = \sin \theta, du = \cos \theta d\theta \\
&= f_r \cdot 2\pi \cdot \frac{u^2}{2}\Big|_0^1 \\
&= f_r \pi
\end{aligned}
$$

meaning that the BRDF for a diffuse material is ρ/π.

Specular reflections, in turn, account for the direction-dependent component of light [14]. Surfaces that reflect light with the same angle of incidence are known as ideal specular surfaces and act like mirrors. Taking into account the optics of light [45], the reflection direction $\boldsymbol{r}$ (often referred to as mirror reflection direction) of an ideal specular surface is computed based on the incident direction $\boldsymbol{d}_i$ and the normal $\boldsymbol{n}$ at the surface as

$$\boldsymbol{r} = 2(\boldsymbol{n} \cdot \boldsymbol{d}_i)\boldsymbol{n} - \boldsymbol{d}_i.$$

For non-ideal specular surfaces, also known as glossy surfaces, the specular direction in which light is reflected is partially diffused forming a small lobe of possible directions centered around the reflective direction $\boldsymbol{r}$ [12]. Formulating a BRDF that correctly shapes the physical behavior of specular materials is complicated and will be discussed below. Lastly, Figure 4.14 presents an illustration of how diffuse, fully specular, and glossy surfaces reflect light.

4.7 The Rendering Equation

Global illumination, as previously mentioned, is intended to compute the steady state of the light distribution in a scene. Additionally, the reflection properties of a surface can be approximated by a BRDF. In this sense, and building on the definitions of radiance and the BRDF, the rendering equation determines the radiance of the reflected surface along a specific viewing direction. The rendering equation was formally introduced by Kajiya [95] and is fundamental for solving the global illumination problem.

The hemispherical formulation of the rendering equation provides an intuitive introduction following the concepts reviewed earlier [14]. The total outgoing radiance leaving a surface at a point **p** towards a direction $\boldsymbol{d}_o$ can be expressed as the sum of the emitted radiance of the surface denoted as $L_e(\boldsymbol{d}_o)$ and the reflected radiance $L_o(\boldsymbol{d}_o)$, such that,

$$L(\boldsymbol{d}_o) = L_e(\boldsymbol{d}_o) + L_r(\boldsymbol{d}_o).$$

From the definition of a BRDF, it follows that,

$$\begin{aligned} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) &= \frac{dL_r(\boldsymbol{d}_o)}{L(\boldsymbol{d}_i)\cos\theta_i d\omega} \\ \rightarrow dL_r(\boldsymbol{d}_o) &= L_i(\boldsymbol{d}_i) f_r(\boldsymbol{d}_i, \boldsymbol{d}_o)\cos\theta_i d\omega \\ \rightarrow L_r(\boldsymbol{d}_o) &= \int_{\boldsymbol{d}_i \in \Omega} L(\boldsymbol{d}_i) f_r(\boldsymbol{d}_i, \boldsymbol{d}_o)\cos\theta_i d\omega. \end{aligned}$$

Then, the rendering equation can be expressed as

$$L(\boldsymbol{d}_o) = L_e(\boldsymbol{d}_o) + \int_{\boldsymbol{d}_i \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L(\boldsymbol{d}_i)\cos\theta_i d\omega.$$

Nonetheless, if the surface material does not emit light (i.e. it is not a light source) then the surface emitted radiance term can be ignored. In addition, if the only light source in the scene is a point light source, the integral through the hemisphere of incoming directions is omitted, since there is only one direction of incoming light, further simplifying the model. Under such assumptions, the rendering equation can be rewritten as,

$$L(\boldsymbol{d}_o) = L(\boldsymbol{d}_i) f_r(\boldsymbol{d}_i, \boldsymbol{d}_o)\cos\theta_i d\omega.$$

4.8 Shading Models

The route followed in this work introduces different concepts important to ray tracing, such as the radiometric quantities of light and their relation with the global illumination problem. Understanding each topic is essential and will allow us to finally implement shading models incorporating the reviewed concepts to light up the objects presented in the scene.

4.8.1 Phong Model

The most popular shading model was introduced by Phong [119], which was empirically designed and provides a good balance between simplicity and realism in terms of rendering. The Phong model is comprised of three components, ambient, diffuse, and specular,

$$L = k_a L_a + k_d L_d + k_s L_s,$$

where the constants k_a, k_d, and k_s define the color and reflection properties of the material. For instance, setting $k_a = 0, k_d = 1, k_s = 0$ means that the surface is entirely diffuse, while $k_a = 0, k_d = 0.7, k_s = 0.3$ means that the surface is partially diffuse and partially specular which can simulate a surface like varnished wood. However, the model is not physically based, as the sum of all components may be greater than 1, failing to follow the law of conservation of energy. This is observed when accounting for the ambient component, used to replace the effect of indirect illumination by a constant radiance term $L_{ambient}$ that looks to simulate the inter-reflections of light between the objects in the scene [12, 60].

The diffuse component considers that the reflected radiance is dependable only on the incident direction. As such, the maximum reflected radiance occurs when the incident radiance direction $\boldsymbol{d}_i$ is perpendicular to the surface, and declines as the angle between the surface normal increases, which can be formulated as $\cos(\theta) = \boldsymbol{n} \cdot \boldsymbol{d}_i$. Then the diffuse radiance term can be expressed as:

$$L_{\text{diffuse}} = L_i \cos(\theta).$$

Specular reflections, as mentioned previously, depend both on the incident and reflection directions of radiance. Ideal specular surfaces reflect light only along the mirror direction $\boldsymbol{r}$ whereas glossy surfaces reflect light along a lobe of possible directions centered at r. A clear example of an ideal specular surface is polished metal as it reflects light only around the mirror direction. In rendering, the interest lies in those reflections that are visible to the camera and have direction $\boldsymbol{v} = \dfrac{\mathbf{c} - \mathbf{p}}{||\mathbf{c} - \mathbf{p}||}$, where $\mathbf{c}$ represents the camera location in world coordinates and $\mathbf{p}$ is the surface point from which radiance is reflected. In this regard, Phong formulated the specular radiance along the direction vector $\boldsymbol{v}$ as,

$$L_{\text{specular}} = L_i cos(\theta)^{n_s},$$

where θ is the angle between the viewing direction $\boldsymbol{v}$ and the mirror reflection direction $\boldsymbol{r}$. The term $n_s > 1$ is known as shininess, and models the specular highlight falloff, meaning the larger its value the closer the specular reflection is to the mirror reflection [12, 119]. In simpler words, the shininess term n_s determines how wide the specular lobe is, with higher values narrowing the lobe of reflections to that of the only mirror reflection $\boldsymbol{r}$. Then the Phong model can be written as,

$$L = k_a L_a + L_i(k_d(\boldsymbol{n} \cdot \boldsymbol{d}_i) + k_s(\boldsymbol{r} \cdot \boldsymbol{v})^{n_s}).$$

There is a missing piece in this puzzle, where is the color component? Recall that visible light propagates in the form of waves with different wavelengths. When considering the electromagnetic spectrum of light, a wavelength is associated with a specific color perceived by the three cone photoreceptors (S-blue, M-green, and L-red) in our eyes. In computer graphics, the RGB model represents color as perceived by our eyes using a triplet of normalized values between 0 and 1, each representing the intensity of the red, green, and blue wavelengths.

In this way, in rendering radiance being wavelength dependent is treated as a triplet of RGB colors, $\boldsymbol{L} = (\boldsymbol{L}_R, \boldsymbol{L}_G, \boldsymbol{L}_B)$ [12, 60]. Additionally, for the Phong model, each component (ambient, diffuse, and specular) is given a specific color expressed as an RGB triplet ($\boldsymbol{c}_a$, $\boldsymbol{c}_d$, and $\boldsymbol{c}_s$) such that,

$$\boldsymbol{L} = k_a \boldsymbol{c}_a \boldsymbol{L}_a + \boldsymbol{L}_i(k_d(\boldsymbol{n} \cdot \boldsymbol{d}_i)\boldsymbol{c}_d + k_s(\boldsymbol{r} \cdot \boldsymbol{v})^{n_s}\boldsymbol{c}_s).$$

For example, a grey ambient color (0.6,0.6,0.6) ◼, an orange diffuse color (1.0,0.6,0.0) 🟧, and a white specular color (1.0,1.0,1.0). Moreover, by setting the coefficients of each component to be $k_a = 1, k_d = 0.8, k_s = 0.2$ and $\boldsymbol{L}_a = \boldsymbol{L}_i = (1.0, 1.0, 1.0)$, the Phong model

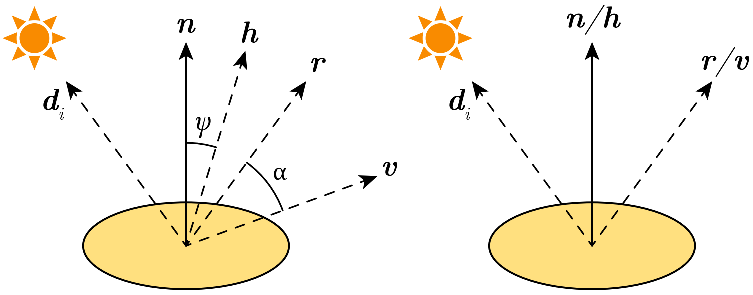


Figure 4.15: Blinn-Phong model where the half vector $\boldsymbol{h}$ prevents from calculating the reflection vector $\boldsymbol{r}$.

output radiance can be expressed as,

$$\begin{bmatrix} \boldsymbol{L}_R \\ \boldsymbol{L}_G \\ \boldsymbol{L}_B \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.6 \\ 0.6 \end{bmatrix} + (\boldsymbol{n} \cdot \boldsymbol{d}_i) \begin{bmatrix} 0.8 \\ 0.48 \\ 0.0 \end{bmatrix} + (\boldsymbol{r} \cdot \boldsymbol{v})^{n_s} \begin{bmatrix} 0.2 \\ 0.2 \\ 0.2 \end{bmatrix}.$$

4.8.2 Blinn-Phong Model

Blinn [149] improves upon the work of Phong by introducing an important optimization to the model. From Figure 4.15 it is noticed the direction vector $\boldsymbol{h}$ represents the normalized half vector between $\boldsymbol{d}_i$ and $\boldsymbol{v}$ and is computed as,

$$\boldsymbol{h} = \frac{\boldsymbol{d}_i + \boldsymbol{v}}{||\boldsymbol{d}_i + \boldsymbol{v}||}.$$

Observe that, when the view direction $\boldsymbol{v}$ is equal to the mirror direction $\boldsymbol{r}$, the half-vector $\boldsymbol{h}$ is equal to the normal vector $\boldsymbol{n}$. In other words, the angle ψ formed between the normal vector $\boldsymbol{n}$ and the half-vector $\boldsymbol{h}$ is a measure of how distant the viewing direction $\boldsymbol{v}$ from the mirror direction $\boldsymbol{r}$. Therefore, the mirror reflection vector $\boldsymbol{r}$ can be omitted, and instead the dot product $\boldsymbol{n} \cdot \boldsymbol{h}$ is employed to compute the specular term of the Phong shading model.

$$\boldsymbol{L} = k_a \boldsymbol{c}_a \boldsymbol{L}_a + \boldsymbol{L}_i (k_d (\boldsymbol{n} \cdot \boldsymbol{d}_i) \boldsymbol{c}_d + k_s (\boldsymbol{n} \cdot \boldsymbol{h})^{n_s} \boldsymbol{c}_s).$$

4.9 Physically Based Models

From the Phong model, it is known that a surface illuminated from an incident direction will reflect light across all directions with a max peak around the mirror direction. Nonetheless, several observations showed that for rough surfaces the mirror direction was not the peak direction [60]. Because the Phong model is empirical, the reflection properties of the rendered surfaces may differ from their real counterparts [12]. The solution is to turn to the field of geometrical optics in physics and adapt the physically based models already developed. One of these models was developed by Torrance and Sparrow [150] and describes off-mirror peaks for both metallic and non-metallic surfaces. However, before describing this model, an introduction to the Fresnel equations is given that will become handy later.

4.9.1 The Fresnel Equations

The Fresnel equations describe the reflection and refraction of light as it travels from one medium to another, from air to water for instance [11, 60]. When considering light's waveform propagation, it can be differentiated into polarized and unpolarized light [45]. Polarized light can be associated with waves whose oscillation occurs along a specific axis-plane perpendicular to the direction of travel, whereas in unpolarized light, waves oscillate at random angles but still perpendicular to the direction of travel [45]. The Fresnel equations, as such, relate the amplitude of the reflected light wave to that of the incident light wave. This proportion is determined based on the indices of refraction of the participating media, η_1 and η_2, and the angle of incident light θ_i and refracted light θ_t also referred to as transmitted light [11]. The Fresnel equations are then divided into,

$$R_{\parallel} = \left| \frac{\eta_1 \cos\theta_i - \eta_2 \cos\theta_t}{\eta_1 \cos\theta_i + \eta_2 \cos\theta_t} \right|^2$$
$$R_{\perp} = \left| \frac{\eta_1 \cos\theta_t - \eta_2 \cos\theta_i}{\eta_1 \cos\theta_t + \eta_2 \cos\theta_i} \right|^2$$

where $R_{\parallel}$ and $R_{\perp}$ account for the reflectance of parallel and perpendicular polarized light regarding the point of incidence (boundary of the two media). To differentiate them from the reflectance term ρ reviewed earlier, these are referred to as Fresnel reflectance terms.

The angle of refracted light is computed based on Snell's law [45] which states that the ratio between the incident sine angle and refraction sine angle is equal to the ratio of the index of refraction of the second medium with regard to the first medium:

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_2}{\eta_1}$$

which is also expressed as,

$$\eta_1 \sin\theta_i = \eta_2 \sin\theta_t.$$

Nonetheless, polarized light is imperceptible for humans or cameras if not using a specialized polarized filter [11]. In this way, because unpolarized light is the superposition of multiple polarized waves, the reflectance of unpolarized light is expressed as the average of $R_\parallel$ and $R_\perp$,

$$R = \frac{R_\parallel + R_\perp}{2}.$$

An alternative to computing the Fresnel equation for unpolarized light is to use the approximation developed by Schlick [151] which is formulated as,

$$F = R_0 + (1 - R_0)(1 - \cos\theta)^5.$$

The term R_0, often referred to as base reflectance, represents the surface material's specular reflectance at normal incidence, that is when the incident light is perpendicular to the surface. For dielectrics or non-metals (e.g., wood, glass, plastic), the base reflectance term is computed as:

$$R_0 = \left(\frac{\eta_2 - \eta_1}{\eta_2 + \eta_1}\right)^2.$$

where η_1 is often set to be the index of refraction in vacuum (which is 1), and η_2 is the index of refraction of the medium with which light interacts. The base reflectance term for metals is computed differently, as it involves complex values. In this sense, when considering these materials in rendering, a table of pre-calculated R_0 values for different materials is used. Since specular reflectance is wavelength dependent, it can be encoded with an RGB triplet representing the proportion of red, green, and blue light perceived by our eyes [3]. Table 4.2 shows some common base reflectance values for different materials.

Table 4.2: Base reflectance F_0 values for different materials. Each value is expressed as a normalized RGB triplet in the ’Linear’ column and its respective integer 8-bit representation in the ’Texture’ column. Values with just one term indicate a triplet with the same value in all components. Moreover, only one value was picked for those materials whose reflectances are represented in ranges. All values were obtained from [2, 3].

Material	**Linear**	**Texture**	**Color**
Water	0.02	39	⬛
Skin	0.028	47	⬛
Fabric	0.04-0.056	56-67	⬛
Plastic/Glass	0.04-0.05	56-63	⬛
Gems	0.05–0.08	63–80	⬛
Diamond-like	0.13-0.20	101–124	⬛
Iron	(0.56, 0.56, 0.57)	(198, 198, 200)	⬜
Copper	(0.95, 0.64, 0.54)	(250, 209, 194)	🟧
Gold	(1.00, 0.78, 0.34)	(255, 229, 158)	🟨
Aluminum	(0.91, 0.92, 0.92)	(245, 246, 246)	⬜
Silver	(0.97, 0.96, 0.91)	(252, 250, 245)	⬜

4.9.2 Torrance-Sparrow Model

The Torrance-Sparrow model is based on the theory of microfacets [3], which represents a surface as a collection of tiny mirrors randomly oriented. If a small differential area on a surface is considered, the normal of that differential patch is said to be n. However, when considering the microfacet aspect, the normal n represents the average alignment of the microfacet normals and is referred to as the macroscopic normal [152].

The Torrance-Sparrow model considers that the microfacets are in pairs forming a “$\vee$” shape and can have different slopes [3, 60]. Furthermore, the diffuse component of the reflected light is the result of multiple inter-reflections between the microfacets, while the specular component is associated with those microfacets whose normals are aligned with the h vector [149]. In this way, the specular component of the BRDF can be expressed as the combination of four factors,

$$L_s = \frac{DGF}{4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})}.$$

The D term represents the distribution of microfacet normals in the surface, G accounts for the effect by which microfacets shadow or mask each other, and F is the Fresnel equation for unpolarized light [12]. The division by $4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})$ accounts for both the incident and reflected projected areas of radiance. The representation of the D used in the Torrance-Sparrow model is the Gaussian distribution

$$D = e^{-(\gamma c)^2}.$$

As such, the D component represents the statistical distribution of microfacet normals oriented at an angle γ of the average normal $\boldsymbol{n}$. Remember that the specular reflection of interest is that for which the microfacet normals are oriented with the half vector, then $\gamma = cos^{-1}(\boldsymbol{n}\cdot\boldsymbol{h})$. The c factor represents the standard deviation of the distribution and is a surface material property [149].

The term G is known as the geometrical attenuation factor. It describes how microfacets can block some of the reflected light reducing the outgoing radiance causing a masking effect, or can block some of the incoming radiance at a point causing a shadow effect. Figure 4.16 presents a surface both at the macroscopic level as a smooth line and at the microscopic level comprised of different microfacets, the effects of masking and shadowing are also illustrated. Continuing, the G term is then a value between 0 and 1, representing the fraction of light that remains after masking and shadowing effects, which is 1 when all the incident radiance is reflected [12]. The G term is then computed as,

$$\begin{aligned} G1 &= \frac{2(\boldsymbol{n}\cdot\boldsymbol{h})(\boldsymbol{n}\cdot\boldsymbol{v})}{\boldsymbol{v}\cdot\boldsymbol{h}}, \\ G2 &= \frac{2(\boldsymbol{n}\cdot\boldsymbol{h})(\boldsymbol{n}\cdot\boldsymbol{l})}{\boldsymbol{v}\cdot\boldsymbol{h}}, \quad \boldsymbol{l} = \boldsymbol{d}_i \\ G &= \min(1, G1, G2) \end{aligned}$$

where G_1 accounts for the masking effect and G_2 for the shadowing effect. Blinn [149] provides a good explanation of how both components are calculated. Finally, the Fresnel equation F term used in the Torrance-Sparrow model differs from the one reviewed earlier, having the following form,

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right),$$

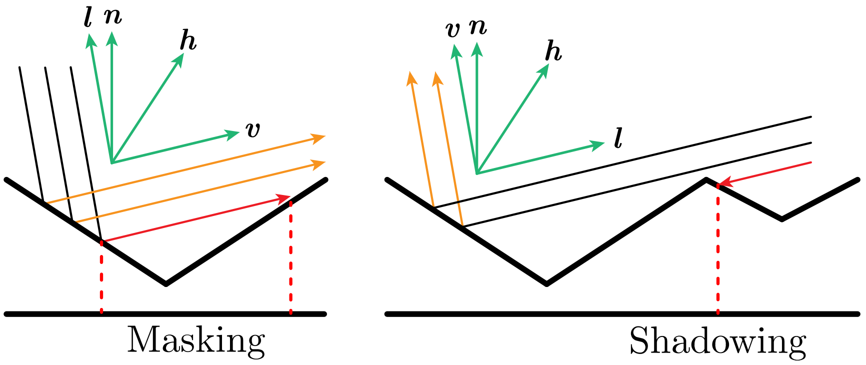


Figure 4.16: Microfacet representation of a surface based on the Torrance-Sparrow model. The incident radiance direction $\boldsymbol{d}_i$ is represented as $\boldsymbol{l}$. The masking effect is shown on the left side while the shadowing effect is on the right side, notice that $\boldsymbol{l}$ and $\boldsymbol{v}$ are interchanged but $\boldsymbol{h}$ remain the same.

where

$$c = \boldsymbol{h} \cdot \boldsymbol{v}$$

$$g = \sqrt{n^2 + c^2 - 1}.$$

If the light incidence is normal, $c = 1$ and $g = n$, where n represents the second medium's index of refraction. As such, it can be computed from the formula of base reflectance by considering that the first medium corresponds to a vacuum.

$$R_0 = \frac{(n-1)^2}{(n+1)^2}$$

$$\rightarrow n = \frac{1 + \sqrt{R_0}}{1 - \sqrt{R_0}}$$

4.9.3 Cook-Torrance Model

In the following years, Cook and Torrance [153] adapted the physically based Torrance-Sparrow model into computer graphics. This model considers that surfaces are not ideal Lambertian nor ideal specular, but rather a linear combination of both. In this respect, it also considers glossy reflections, following the idea that a surface is a set of distributed microfacets. This approach in the difference of specular reflections can be employed to simulate different materials. Plastics, for example, specularly reflect light with the same spectral distribution as the light source, while metals tint the reflected light with their color. The model is also comprised of three components (ambient, diffuse, and specular), and is formally expressed as

$$L = L_a\rho_a + L_i(k_d f_{r,d} + k_s f_{r,s}(\boldsymbol{d}_i, v))(\boldsymbol{n} \cdot \boldsymbol{d_i})d\omega.$$

The values of k_d and k_s represent the proportion of the diffuse and specular radiance reflected, where $k_d + ks = 1$. The ρ_a term accounts for the indirectly received light of the ambient as a reflectance value. The terms $f_{r,d}$ and $f_{r,s}$ represent the diffuse and specular BRDFs of the model. Recall from earlier, that the BRDF of a diffuse material is direction-independent and reflects light uniformly across the hemisphere of directions. It is defined as $f_{r,d} = \rho_d/\pi$, where ρ_d denotes diffuse reflectance of the material. The specular BRDF in turn, is the one developed in the Torrance-Sparrow model, this being,

$$f_{r,s} = \frac{DGF}{4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})}.$$

Thus, the Cook-Torrance BRDF is defined as $f_r(\boldsymbol{d}_i, \boldsymbol{v}) = k_d f_{r,d} + k_s f_{r,s}(\boldsymbol{d}_i, v)$. Recall the Fresnel equation term F, calculated as,

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right),$$

where $c = \boldsymbol{n} \cdot \boldsymbol{v}$, $g = \sqrt{n^2 + c^2 - 1}$, and $n = \dfrac{1+\sqrt{\rho_s}}{1-\sqrt{\rho_s}}$. Note that the specular reflectance at normal incidence is represented in the literature as R_0 or F_0, however, to maintain consistency across the notation used, it will be more congruent to define it as ρ_s. The same change in notation can be applied to the Fresnel-Schlick approximation.

Different physical BRDFs derive from the Cook-Torrance BRDF, using alternative functions to compute the specular component. Table 4.3 presents different functions employed to calculate each component of the specular BDRF. Moreover, these variations are often based on the metalness-roughness workflow [2, 3], which specifies the characteristics of a material in terms of its metalness and roughness. The metalness property m is either 0 or 1, where $m = 0$ denotes a dielectric material and $m = 1$ is a metallic material. On the other hand, the roughness parameter r determines the degree of distribution of microfacets on the surface, which affects the sharpness of the specular highlight [3]. It is defined as a value between 0.0 and 1.0, where $r = 0.0$ describes a perfectly smooth surface, and $r = 1.0$ is a rough surface.

Table 4.3: Alternative functions for computing the statistical distribution of microfacet normals D, and the geometric attenuation factor G. For the computation of D, the vector $\boldsymbol{m}$ represents the microfacets normals. When computing G, the vector $\boldsymbol{s}$ represents either $\boldsymbol{v}$ or $\boldsymbol{l}$ to account for the effect of masking or shadowing, respectively. The value of α is equal to the squared roughness.

	Name	**Expression**
Distribution Term	Beckman [154]	$D(\boldsymbol{m}) = \frac{1}{\pi\alpha^2(\boldsymbol{n}\cdot\boldsymbol{m})^2}\exp\left(\frac{(\boldsymbol{n}\cdot\boldsymbol{m})-1}{\alpha^2(\boldsymbol{n}\cdot\boldsymbol{m})^2}\right)$
	Trowbridge-Reitz/GGX [155]	$D(\boldsymbol{m}) = \frac{\alpha^2}{\pi((\boldsymbol{n}\cdot\boldsymbol{m})^2(\alpha^2-1)+1)^2}$
	GGX Anisotropic [152]	$D(\boldsymbol{m}) = \frac{1}{\pi\alpha_x\alpha_y}\frac{1}{\left(\frac{(\boldsymbol{x}\cdot\boldsymbol{m})^2}{\alpha_x^2}+\frac{(\boldsymbol{y}\cdot\boldsymbol{m})^2}{\alpha_y^2}+(\boldsymbol{n}\cdot\boldsymbol{m})^2\right)^2}$
Geometry Term	Neumann [156]	$G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h}) = \frac{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}{\max(\boldsymbol{n}\cdot\boldsymbol{l},\boldsymbol{n}\cdot\boldsymbol{v})}.$
	Kelemen [157]	$G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h}) = \frac{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}{(\boldsymbol{v}\cdot\boldsymbol{h})^2}.$
	Smith-GGX [152]	$G(\boldsymbol{s}) = \frac{2(\boldsymbol{n}\cdot\boldsymbol{s})}{(\boldsymbol{n}\cdot\boldsymbol{s})+\sqrt{\alpha^2+(1-\alpha^2)(\boldsymbol{n}\cdot\boldsymbol{s})^2}}$, such that, $G = G(\boldsymbol{l})G(\boldsymbol{v})$
	Schlick-GGX [158]	$G(\boldsymbol{s}) = \frac{\boldsymbol{n}\cdot\boldsymbol{s}}{(\boldsymbol{n}\cdot\boldsymbol{s})(1-k)+k}, k = \frac{\alpha}{2}$, such that, $G = G(\boldsymbol{l})G(\boldsymbol{v})$

4.10 Path Tracing

Radiometry serves as the backbone of global illumination by explaining how light reflects off surfaces, which is fundamental to understanding the rendering equation and its various components. With radiometry established, we are now one step closer to solving the global illumination problem. The final aspect to address involves the conveyance of light from one point to another. To explore this, let us revisit the rendering equation while ignoring the emission term, such that,

$$L(\boldsymbol{d}_o) = \int_{\boldsymbol{d}_i \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L(\boldsymbol{d}_i) \cos\theta_i d\omega.$$

The integral over the hemisphere of incident directions Ω implicitly accounts for all direct and indirect incoming radiance, where

$$L(\boldsymbol{d}_i) = L_{direct}(\boldsymbol{d}_i) + L_{indirect}(\boldsymbol{d}_i).$$

As such, direct radiance accounts for light that is coming directly from a light source, whereas the indirect component considers all incident radiance resulting from the different inter-reflections of light from the other surfaces in the scene. Then, the rendering equation can be split to account for direct and indirect radiance independently, such that,

$$L(\boldsymbol{d}_o) = \int_{\boldsymbol{d}_i \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L_{direct}(\boldsymbol{d}_i) \cos\theta_i d\omega + \int_{\boldsymbol{d}_i \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L_{indirect}(\boldsymbol{d}_i) \cos\theta_i d\omega.$$

This latter representation of the rendering equation can be simplified even more by assuming that there is a finite number of point light sources in the scene, which is expressed as,

$$L(\boldsymbol{d}_o) = \sum_{i=1}^{N} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L_{direct}(\boldsymbol{d}_i) \cos\theta_i + \int_{\boldsymbol{d}_i \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L_{indirect}(\boldsymbol{d}_i) \cos\theta_i d\omega.$$

Despite the different reformulations and simplifications, the rendering equation remains difficult to solve due to the indirect component.

Path Tracing is one of the proposed solutions, introduced by Kajiya [95] along the rendering equation. Path tracing is a probabilistic method that sends rays from the camera and traces paths to a light source or until a determined number of bounces/inter-reflections is reached. The method is based on Monte Carlo integration and is employed to approximate the integral regarding the indirect radiance term [14, 28]. Monte Carlo integration consists of selecting N random samples x_i over the domain of the integral, where the probability density function (PDF) [142] of selecting one sample is $p(x_i)$ such that,

$$\int f(x)dx \approx (b-a)\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}, a \leq x_i \leq b.$$

This notion can be applied to approximate the indirect term of the rendering equation, resulting in the following,

$$\int_{\boldsymbol{d}_i \in \Omega} f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L_{indirect}(\boldsymbol{d}_i) \cos\theta_i d\omega \approx \frac{1}{N} \sum_{i=1}^{N} \left(\frac{f_r(\boldsymbol{d}_i, \boldsymbol{d}_o) L_{indirect}(\boldsymbol{d}_i) \cos\theta_i d\omega}{p(\boldsymbol{d}_i)} \right)$$

where $p(\boldsymbol{d}_i)$ describes the probability density of selecting that direction in specific. In this way, the problem is reduced to a finite number of sampling directions. However, since $L_{indirect}(\boldsymbol{d}_i)$ is unknown, a ray is traced according to the surface'es BRDF[28]. Thus, computing the rendering equation can be divided into two parts: determining the direct incident radiance from a light source and tracing rays based on the surface's BRDF to account for indirect incident radiance. Later, we will see that when using an infinite area light source, sampling directions for direct illumination becomes unnecessary.

4.10.1 Indirect Illumination

As previously discussed, the Monte Carlo integration method samples a finite number of incident directions to approximate the contribution of all incoming indirect radiance. However, sampling for incident directions requires considering the BRDF of the surface material. Given the Cook-Torrance BRDF, a material can have diffuse and specular reflections, where the contribution of each is given by their respective coefficients k_d and k_s. By using a physically-based BRDF, the indirect term of the rendering equation can be rewritten as,

$$\begin{aligned} L &= \int_{\boldsymbol{d}_i \in \Omega} \left(k_d \frac{\rho_d}{\pi} + k_s \frac{NDF}{4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})} \right) L(\boldsymbol{d}_i) \cos\theta_i d\omega \\ &= \int_{\boldsymbol{d}_i \in \Omega} k_d \frac{\rho_d}{\pi} L(\boldsymbol{d}_i) \cos\theta_i d\omega + \int_{\boldsymbol{d}_i \in \Omega} k_s \frac{NDF}{4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})} L(\boldsymbol{d}_i) \cos\theta_i d\omega. \end{aligned}$$

Dutre et al. [14] explain that sampling the previous form of the indirect term can be done as follows:

1. Constructing a discrete PDF $p(\boldsymbol{d}_i)$ for three events, whose probabilities are q_1, q_2, and $q3$, such that, $q1 + q2 + q3 = 1$. The three events determine which component of the BRDF to be sampled, where q_1 accounts for the diffuse component. q_2 accounts

for the specular component, and q_3 can be considered the event of absorption.

2. The sampling incident direction is then computed either with $p_1(\boldsymbol{d}_i)$ or $p_2(\boldsymbol{d}_i)$, which represent the PDF of the diffuse and specular BRDF components, respectively.
3. Select the event the sampled direction $\boldsymbol{d}_i$ represents, where the estimator is

$$\begin{cases} \dfrac{k_d \frac{\rho_d}{\pi} L(\boldsymbol{d}_i) \cos\theta_i}{q_1 p_1(\boldsymbol{d}_i)}, & \text{if event 1} \\ \dfrac{k_s \frac{NDF}{4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})} L(\boldsymbol{d}_i) \cos\theta_i}{q_2 p_2(d_i)}, & \text{if event 2} \\ 0, & \text{if event 3} \end{cases}$$

Dutre et al. [14] also comment that an alternative is to consider the sampled direction $\boldsymbol{d}_i$ to be generated by a single distribution and approximate the indirect radiance integral. Then, the Monte Carlo estimator has the following form,

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{\left(k_d\frac{\rho_d}{\pi} + k_s\frac{NDF}{4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}\right) L(\boldsymbol{d}_i)\cos\theta_i}{q_1 p_1(\boldsymbol{d}_i) + q_2 p_2(\boldsymbol{d}_i)}\right),$$

where $q_1 p_1(\boldsymbol{d}_i) + q_2 p_2(\boldsymbol{d}_i)$ accounts for the PDF of the entire estimator. In this regard, selecting whether to sample a direction for the diffuse or specular component can be solved by generating a random number ξ and sample diffusely if $\xi < q_1$ or specularly otherwise.

Sampling a direction $\boldsymbol{d}_i$ for a diffuse material must follow Lambert's law to contribute to the reflected radiance [14, 28]. Remember that, Lambert's law states that the radiance reflected by a diffuse material is proportional to the cosine of the angle between the surface normal and the direction of the incident light [12]. Then, the PDF $p(\boldsymbol{d}_i)$ for a diffuse material should be cosine should be proportional to $\cos\theta_i$, reducing the number of sampled directions that are close to the horizon of the hemisphere where $\cos\theta_i = 0$ [14]. Considering the properties of a continuous PDF [142], the integral over the hemisphere of directions should be equal to 1. In this sense, for a direction $\boldsymbol{d}_i$ sampled uniformly, $p(\boldsymbol{d}_i)$ will be constant as any direction has the same probability of being sampled. However, the PDF should be proportional to the cosine of the angle, $p(\boldsymbol{d}_i) = C \cos\theta_i$. Then, $p(\boldsymbol{d}_i)$ is integrated over the hemisphere of directions to find the constant C, such that,

$$\begin{aligned}
&\int_0^{\pi/2} p(\boldsymbol{d}_i) \cos\theta d\omega = 1 \\
&\rightarrow \int_0^{\pi/2} C \cos\theta d\omega = 1 \\
&\rightarrow C \int_0^{2\pi} \int_0^{\pi/2} \cos\theta \sin d\theta d\varphi = 1 \\
&\rightarrow C\pi = 1 \\
&\rightarrow C = \frac{1}{\pi}.
\end{aligned}$$

Thus, the normalized PDF for sampling directions on a diffuse surface is $p(\boldsymbol{d}_i) = \cos\theta_i/\pi = (\boldsymbol{n} \cdot \boldsymbol{d}_i)/\pi$. Lastly, to generate a random direction $\boldsymbol{d}_i$ with a cosine distribution [28, 159] the following steps are followed,

1. Generate two random number ξ_1 and ξ_2 between [0,1].
2. Generate φ between $[0, 2\pi]$, $\varphi = 2\pi\xi_1$.
3. Generate z between $[0, 1]$, $z = \sqrt{\xi_2}$.
4. Let $\theta = \cos^{-1} z$.
5. Generate the random vector $(\cos\varphi \sin\theta, \sin\varphi \sin\theta, \cos\theta)$.
6. Create a coordinate system centered around the normal vector $\boldsymbol{n}$. For the tangent vector t select any vectorμ non-parallel to $\boldsymbol{n}$, such that,

   $$\boldsymbol{t} = \frac{\mu \times \boldsymbol{n}}{||\mu \times \boldsymbol{n}||}$$

   The bitangent vector is then,

   $$\boldsymbol{t} = \boldsymbol{n} \times \boldsymbol{t}.$$

7. Multiply the generated random vector with the coordinate system matrix to align it around $\boldsymbol{n}$.

Continuing, sampling a $\boldsymbol{d}_i$ direction for a specular surface depends on the BRDF used. In industry, Unreal Engine 4 [158] uses a combination of the Trowbridge-Reitz/GGX function, the Schlick-GGX function, and a modified version of the Fresnel-Schlick approximation to compute the D, G, and F terms, respectively. Disney [160], in turn, used to employ a combination of a modified Trowbridge/Reizt function, the Smith-GGX function, and the Fresnel-Schlick approximation. For this work, the selected functions are Trowbridge-Reitz/GGX for the D term, Schlick-GGX for the G term, and the Fresnel-Schlick approximation for the F term.

Walter et al. [152] mention that sampling an incident direction accounting for the BRDF microfacet model, first requires sampling a microfacet normal $\boldsymbol{m}$ and then using it to generate an incident direction. The steps to sample a microfacet normal based on [152, 159] are:

1. Generate two random numbers ξ_1 and ξ_2 between [0,1].
2. Generate φ between $[0{,}2\pi]$, $\phi = 2\xi_1$
3. Generate $\theta = \arctan\left(\frac{\alpha\sqrt{\xi_2}}{\sqrt{1-\xi_2}}\right)$
4. Let the microfacet normal $\boldsymbol{m} = (\cos\varphi\sin\theta, \sin\varphi\sin\theta, \cos\theta)$.
5. Transform the generated random microfacet normal $\boldsymbol{m}$ from tangent space to world space. This is done similarly to aligning the direction vector of a diffuse material to the normal $\boldsymbol{n}$.

Once the sampled microfacet normal $\boldsymbol{m}$ is computed, the direction of incident radiance is computed following the direction of reflection, such that,

$$\boldsymbol{d}_i = 2(\boldsymbol{v} \cdot \boldsymbol{m})\boldsymbol{m} - \boldsymbol{v}.$$

Notice that $D(\boldsymbol{m})$ determines the distribution of microfacet normals in the surface. As such, when sampling bmm,

$$p(\boldsymbol{m}) = D(\boldsymbol{m}).$$

However, when generating $\boldsymbol{d}_i$ from $\boldsymbol{m}$, Walter et al. [152] mention that the probability of the resulting direction should include the Jacobian of the half-vector transform given by $\dfrac{1}{4(\boldsymbol{v}\cdot\boldsymbol{h})}$. Then, it follows that,

$$p(\boldsymbol{d}_i)=\frac{D(\boldsymbol{m})(\boldsymbol{n}\cdot\boldsymbol{h})}{4(\boldsymbol{v}\cdot\boldsymbol{h})}.$$

Finally, the Monte Carlo estimator for sampling the BRDF following all aspects mentioned can be written as,

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{\left(k_d\frac{\rho_d}{\pi}+k_s\frac{NDF}{4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}\right)L(\boldsymbol{d}_i)\cos\theta_i}{q_1\frac{(\boldsymbol{n}\cdot\boldsymbol{d}_i}{\pi}+q_2\frac{D(\boldsymbol{m})(\boldsymbol{n}\cdot\boldsymbol{h})}{4(\boldsymbol{v}\cdot\boldsymbol{h})}}\right).$$

An important feature of this approach, particularly when combined with path tracing, is its ability to incorporate an infinite area light source surrounding the scene [30]. With finite light sources, rays from the camera must reach the light source, which often requires multiple inter-reflections or bounces. To manage this, incident directions can be sampled based on the light source's position and shape [28, 161]. Nonetheless, if a ray is occluded by an object in that sampled direction a new direction must be sampled according to BRDF for the object's material [3, 14]. In this regard, a maximum number of bounces is specified when implementing path tracing. An infinite area light source, in turn, provides consistent illumination in all directions, considering a point directly illuminated if the ray bounces only once. Points are indirectly illuminated if a ray undergoes multiple bounces, up to the specified bounce limit. Therefore, this work employs only an infinite area light source for consistent illumination while maintaining simplicity.

Moreover, at the beginning of this chapter, it was mentioned that a Image2D variable is used to store the rendered frame. The Monte Carlo integration method better approximates the solution to the indirect illumination integral as the number of samples increases. In this sense, for every ray sent into the scene, the direction of the following bounce is sampled randomly according to the BRDF. As a result, every rendered frame may differ from the previous one depending on the sampled directions generated. Nonetheless, when averaging the computed radiance, that is, averaging the values of each pixel among all rendered frames, the Monte Carlo integral is computed with not just one sample per pixel but according to the number of frames generated.

To achieve this, a second Image2D variable is used to accumulate the summation results of the Monte Carlo integral, while the first Image2D variable displays the accumulated results divided by the number of frames generated. Consequently, as the number of frames increases, the approximation of the indirect term of the rendering equation improves. However, if the camera position or the direction of the target changes, the content of the accumulated Image2D variable is reset, since a different view of the scene is being approximated. Therefore, as the camera moves, each rendered frame on the screen represents the Monte Carlo approximation of the indirect term integral with a single sample per pixel.

4.11 Summary

During this chapter, the theory for the implementation of different concepts such as rendering primitives, acceleration of ray primitive intersection operations, and one of the solutions for the global illumination problem have been discussed. Although several concepts have been omitted due to lack of time, the references provided will help to clarify and deepen your knowledge. In addition, the theory behind microfacet BRDFs and sampling is so complex that it will take more than one chapter to explain all in detail. Despite these difficulties, this chapter remains a good introduction to delve into the world of ray tracing and physically-based rendering.

Chapter 5

Results and Discussion

Throughout this chapter, the results of the different features of the ray-tracing renderer will be presented. The main metrics are the frame rendering time in milliseconds and the average frame rate per second (AvgFPS). The rendering time of a frame is calculated as the delta time between the previous and current frame. The average frame rate per second is computed as the total number of frames rendered divided by the total elapsed time. The analysis will cover different rendering scenarios and will contrast the differences in the use of an acceleration structure to speed up ray primitive intersection operations. Also, the results will cover the effect of indirect lighting on the realism of a scene. The system specifications on which the tests are performed are presented in Table 5.1.

Table 5.1: Specifications of the system on which the tests were performed

CPU	**RAM**	**GPU**	**VRAM**	**Operating System**
i7-9750H	16 GB	RTX 2070 Max-Q	8 GB	Arch Linux

5.1 Traversal of Bounding Volume Hierarchies

Traversing a bounding volume hierarchy (BVH) can be done in several ways. In this work, two stack-based methods were implemented based on the description provided by Meister et al. [76] and Pharr et al. [11]. Traversing a BVH using a stack is done using a while loop until no nodes are left in the stack. The first node added to the stack is the root of the BVH. For each node, the first step is to check if the ray intersects it, otherwise, the loop proceeds to remove the top node from the stack if it is not empty. If the node is a leaf, the ray is checked against all its primitives. If the stack is not empty, the ray is checked with the remaining nodes for a closer intersection.

The difference between the implemented algorithms lies in how to traverse an interior node. In case the jumped node in the stack is an interior node, and it is intersected, then its children must also be checked for intersection. As Pharr et al. [11] emphasize, it is desirable to visit the node that the ray intersects first, in case a primitive intersection occurs that reduces the $tMax$-value parameter of the ray. In this sense, the first algorithm computes the t_1 and t_2 values for the intersection of the left and right child nodes, respectively. It then pushes each node on the stack with the nearest one on top.

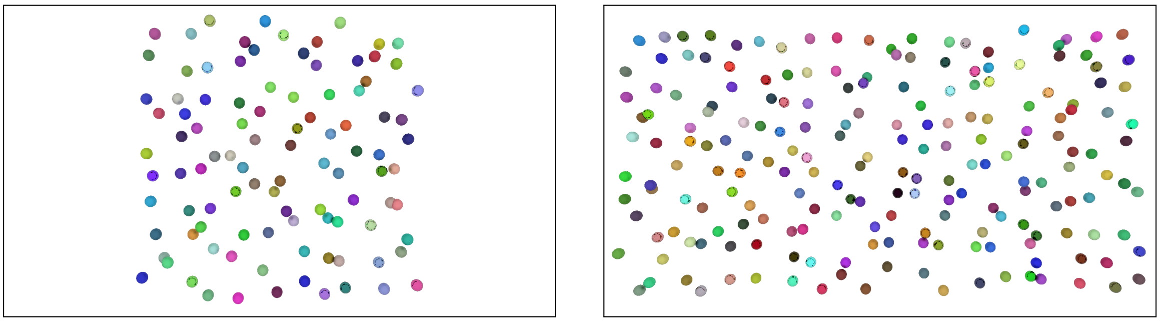
The second method avoids computing t_1 and t_2 by using the sign of the ray's direction vector along the axis for which primitives were partitioned for that interior node. Assuming the ray's sign along the partitioning axis is positive, then the first node to be traversed is the left child, as its primitives belong to the right/bottom/back partition in three dimensions. Conversely, if the ray's sign is negative, then the right child is visited first since its primitives belong to the right/top/front of the partition axis in three dimensions. This explanation considers a right-hand orientation of the coordinate axes, if a left-hand orientation is used, then the order of visiting the nodes along the z-axis is switched as it grows positively towards the screen. These traversal algorithms will be referred to as DB for distance-based and RDSB for ray-direction sign-based, respectively, and their pseudocode is presented in Appendix E and Appendix F.

5.2 Spheres Rendering Comparison

The first analysis consists of rendering different scenarios with a given number of spheres. In this sense, the objective is to analyze the rendering performance of multiple objects at the same time rather than to analyze the realism of the scene. The test analyzes the frame rendering times of each scenario without acceleration structure, and using a BVH for which the DB and RDSB traversal algorithms will be tested.

Table 5.2: Spheres Rendering Comparison - Testing Parameters

Resolution	**Rendering Time**	**Runs**	**FOV$_y$**	**Camera Movement**	**Bounces**	**BVH Bins**
1920×1080	10 s	5	45	No	2	11


(a) 100 spheres - A=10x10 (b) 200 spheres - A=20x10

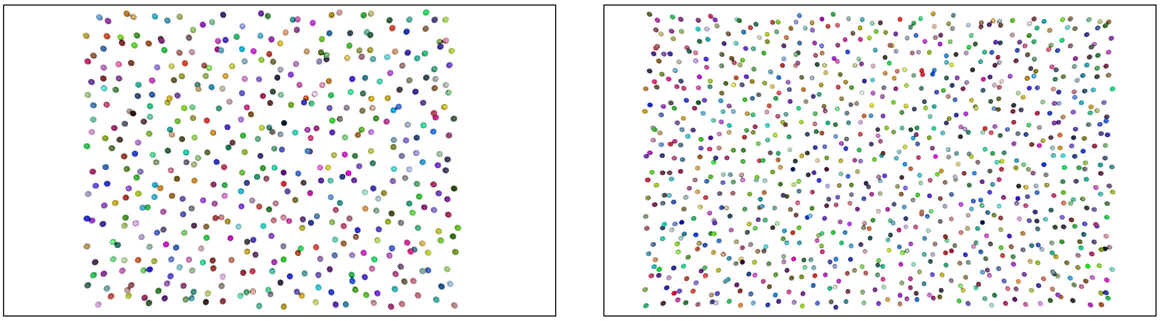

(c) 500 spheres - A=25x20 (d) 1000 spheres - A=40x25

Figure 5.1: Representation of the four scenarios rendered for Test 1

The global parameters for the test are presented in Table 5.2. The rendered frames will have a resolution of 1920×1080 pixels, the total rendering time will be 10 s and each rendering will be executed five times. The FOV$_y$ is set to 45° and is a common value when rendering [2]. The camera is configured to be positioned in a specific point and viewing direction, and no movement will be applied. The spheres across all scenarios are randomly located within a delimited area. The materials of each sphere are also randomly generated, with a 0.2 probability of being metal, and the remaining 0.8 being diffuse. The number of bins into which primitives are partitioned to form the BVH is set to 11. As such, the number of inter-reflections/bounces was set to two. Finally, the number of runs/executions for each scenario was five to generate the results.

Test 1 compares four scenarios consisting of 100, 200, 500, and 1000 spheres where

Figure 5.1 presents a random rendered scene of each scenario. Figure 5.2 presents the results for frame rendering time whereas Table 5.3 shows the average frame rate per second (FPS) for all cases across the 10 seconds of render time. As expected, not using an acceleration structure directly affects each frame's render times. In contrast, the difference between the DB and RDSB algorithms is noticeable. Focusing on the results without acceleration, it is observed that for 100 spheres, the average frame rendering time is 8.24 ms. Over 10 seconds of rendering time in five runs, 7749 frames were generated, meaning an average of 154.98 FPS. However, as the number of spheres increases, the frame rendering times grow drastically, averaging 87.53 ms per frame in the 1000-sphere scenario. As a result, the frame rendering time for 1000 spheres is approximately 10.62 times longer than for 100 spheres. This translates to a reduction in average FPS from 154.98 to 11.42.

Regarding the comparison of BVH traversal algorithms, it is noticed that at rendering 100 spheres the RDSB method renders a frame 0.44 ms faster than the DB method, with the difference in average FPS being 136.88. The gap is maintained when rendering 200 spheres, with the RDSB method rendering a frame in 0.93 ms on average compared to the DB method. This difference in frame rendering times corresponds to a gap of 162.96

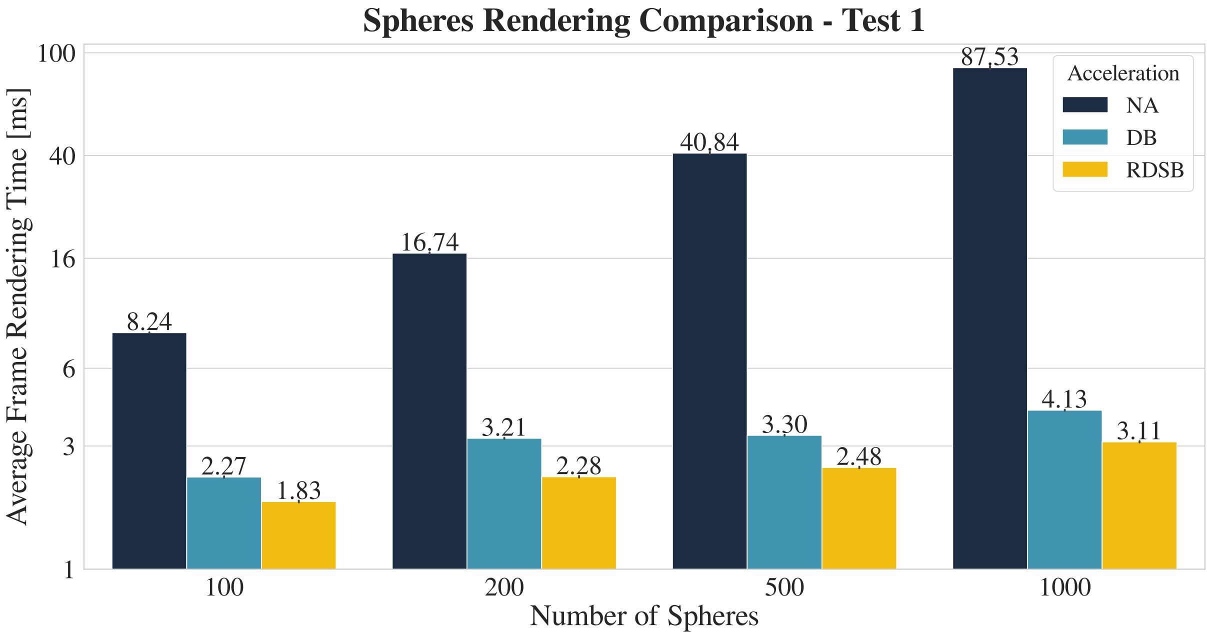


Figure 5.2: Average frame rendering time comparison for each scenario in the absence of an acceleration structure (NA), using a BVH with the distance-based method (DB), and employing the ray-direction sing-based traversal (RDSB). A logarithmic scale is used to represent the results of the five runs.

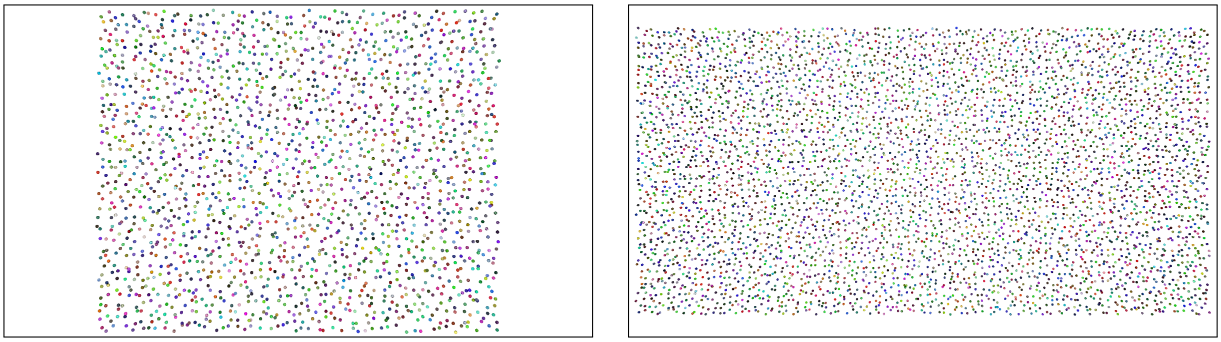
FPS between the two methods. In the 500 spheres scenario, the rendering times increase slightly, with the DB method at 3.30 ms and the RDSB method at 2.48 ms. The reason of this difference is that the area extent where the spheres are distributed covers less viewport space in contrast to the 200 spheres scenario. This reduces the number of ray intersection tests for those pixels in which no sphere is present.

Despite this difference in the distribution of spheres, the RDSB traversal algorithm is still faster by 0.82 ms in rendering a frame as opposed to the DB method. Finally, for the 1000 sphere scenario, the difference is more noticeable, with the average render time of the RDSB method being 1.02 ms faster. In terms of average FPS, the RDSB method achieves 410.76 FPS whereas the DB method generates 310.08 FPS, meaning that the increase in FPS with the RDSB method is of approximately 32.47%.

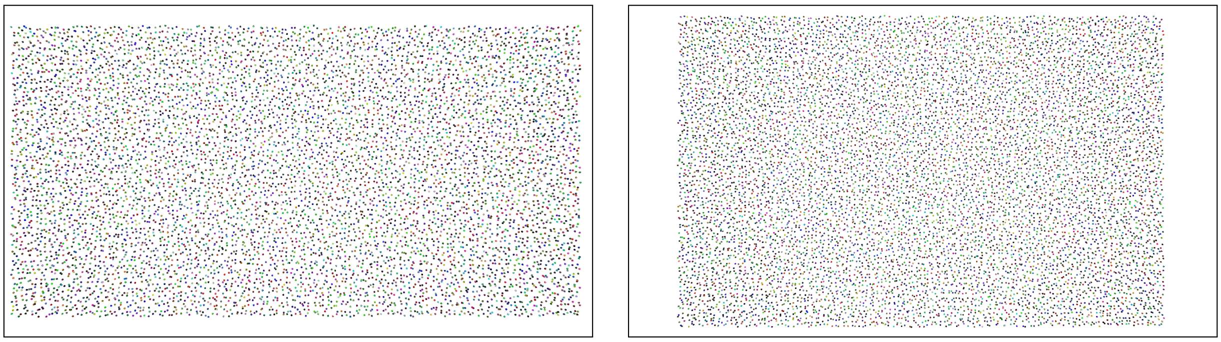
From the results of Test 1, the cost of not using an acceleration structure resulted in longer rendering times. However, the difference between the traversal algorithms can be highlighted even more. To further emphasize their differences, four additional scenarios, consisting of 2000, 5000, 8000, and 10000 spheres, were considered, as illustrated in Figure 5.3. The parameters of test II remained the same as those presented in Table 5.2. The frame rendering time results are presented in Figure 5.4 and the average FPS of each

Table 5.3: Average FPS comparison across five runs for the sphere scenarios rendered in Test 1.

Spheres	**Acceleration**	**Total Frames**	**Avg FPS**
100	NA	7749	154.98
	DB	28149	562.98
	RDSB	34993	699.86
200	NA	3815	76.30
	DB	19913	398.26
	RDSB	28061	561.22
500	NA	1558	31.16
	DB	19338	386.76
	RDSB	25764	515.28
1000	NA	721	14.42
	DB	15504	310.08
	RDSB	20538	410.76



(a) 2000 spheres - A=50x40 (b) 5000 spheres - A=100x50



(c) 8000 spheres - A=125x64 (d) 10000 spheres - A=125x80

Figure 5.3: Representation of rendered scenarios for Test 2

scenario is presented in Table 5.4. Compared to the 1000 spheres scenario from earlier, rendering 2000 spheres with the DB algorithm shows a difference in frame generation time of 3.08 ms. This represents approximately a 1.74 times increase in frame rendering times for twice as many spheres. The RDSB method, in turn, has a difference of only 0.26 ms, representing an increase of 1.08 times to render twice as many spheres. At this point, the difference in the average FPS between both methods has become more pronounced, being 201.96 FPS.

As the number of spheres increases, the tendency continues although not exponentially. In the 5000 sphere scenario, the difference in frame rendering times between the DB and RDSB traversal algorithms is 5.86 ms, with the gap in average FPS being 161.68. Observing the 8000 spheres scenario shows that the frame rendering time difference is 6.80 ms between the two methods, resulting in a gap of 151.36 FPS. Finally, in the 10000 spheres scenario, the frame rendering times decrease slightly due to the smaller area extent covered by the spheres in the viewport contrary to the 8000 spheres scenario. Therefore, the rays sent into these pixels will find no intersection, reducing the frame rendering times. As such,

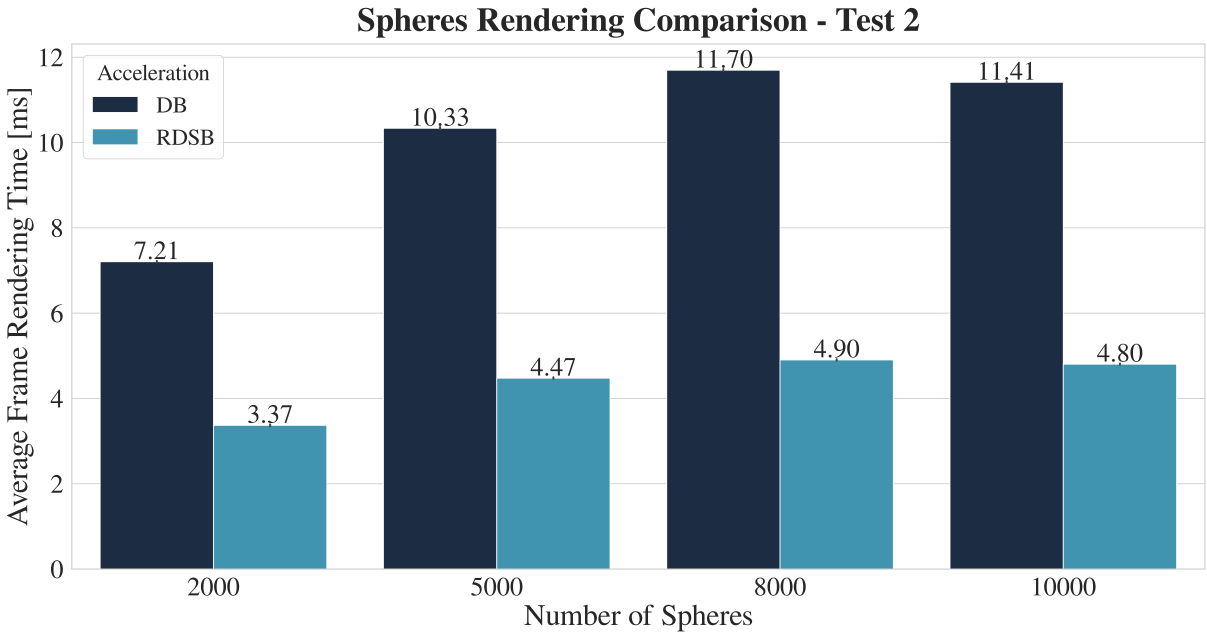


Figure 5.4: Average frame rendering time comparison for Test 2 using the DB and RDSB traversal algorithms. The results represent the average across the five runs.

with the RDSB method, a frame is rendered in 4.80 ms as opposed to 11.41 ms with the DB method.

In summary, Test 2 shows that the RDSB method is more beneficial than the DB method where the difference becomes more noticeable as the number of objects in the scene increases. Note, however, that the camera remains stationary and is positioned so that all objects are visible, which requires it to be far away. As a result, frame rendering

Table 5.4: Average FPS comparison across five runs for the sphere scenarios rendered in Test 2.

Spheres	**Traversal**	**Total Frames**	**Avg FPS**
2000	DB	8822	176.44
	RDSB	18920	378.40
5000	DB	6137	122.74
	RDSB	14227	284.54
8000	DB	5404	108.08
	RDSB	12972	259.44
10000	DB	5534	110.68
	RDSB	13234	264.68

times could be greatly affected if each ray intersects multiple BVH nodes or multiple primitives at a leaf node. This occurs when the camera is positioned in front of several objects covering the entire viewport and accounting for the effects of the BRDF becomes more expensive.

5.3 Analyzing Global Illumination Effects

During this section, an analysis of global illumination and the features of the renderer will be presented. As stated in Section 4.10, the implemented BRDF consists of a diffuse component based on Lambert's law and a combination of the functions Trowbridge-Reitz/GGx, Schlick-GGX, and the Fresnel-Schlick approximation to compute the specular term. Because the implemented BRDF is based on microfacet theory, diffuse and specular reflections are determined based on the distribution of microfacets along the surface. The parameter that determines this distribution is the material's roughness. As the roughness increases, the material's specular reflections decrease and become none when roughness equals one.

Likewise, at the end of Section 4.9, it was mentioned that for implementing a BRDF the metal-roughness workflow is followed. It was also discussed that a material is either dielectric or metallic, denoted by $m = 0$ and $m = 1$, respectively. However, for artistic purposes, it is common to vary m in the range $[0, 1]$ as done in Unreal Engine 4 [158] and by Disney [160]. Figure 5.5 presents the different variations for a material rendered with the implemented BRDF. In the first row (from bottom to top), it is noticed that as the roughness increases, the reflections of light become diffuse as more light is scattered in all directions. In the second row, the effect is the contrary, as the roughness decreases, light is reflected specularly having a glossy finish. Finally, in the third row, as the value of m increases, the material converts from a dielectric to a metal. However, representing a metallic material requires setting roughness to 0.0 so that light can be reflected specularly.

The difference between a highly specular dielectric and a metal can be observed in the third row, specifically when $m = 0$ and $m = 1$. At $m = 0$ the sphere represents a grey specular dielectric, whereas at $m = 1$, the sphere has a grey metallic material. The reflections of the below spheres on the specular dielectric have the color of the material itself, grey. In contrast, the reflections in the metallic material are tinted by the metal color. From a programming perspective, this can be understood as multiplying the RGB color representation of the below spheres by the RGB color representation of the metal. As a result, the blue color 🟦 of the second row and the purple color 🟪 of the first row are multiplied by the grey color ⬜ of the metallic sphere resulting into a darker blue 🟦 and purple 🟪 colors.

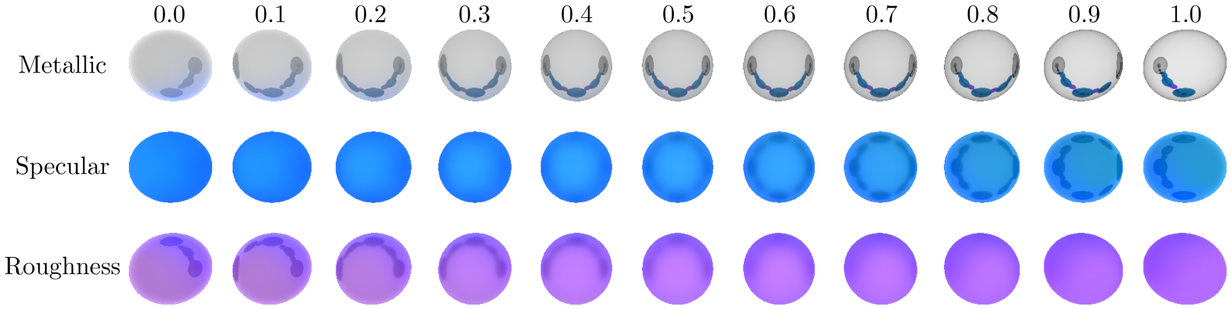


Figure 5.5: Differences between metallic, specular, and roughness properties in surfaces.

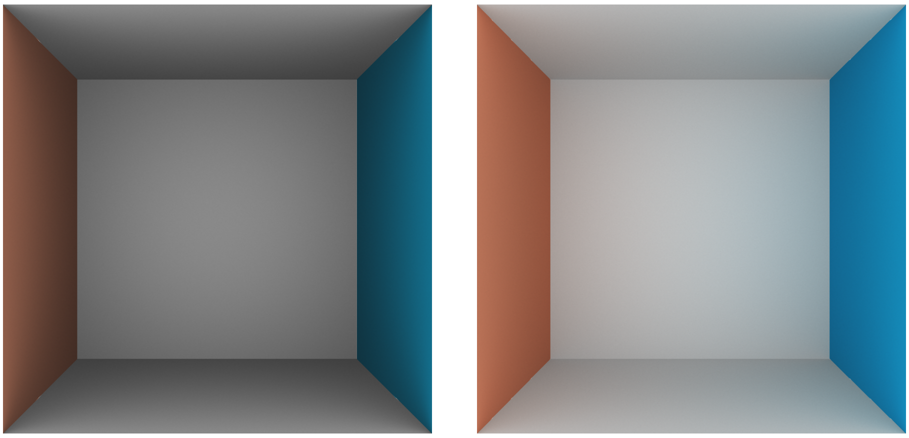
A common model for testing the features of a ray tracing renderer is the Cornell Box model [162]. The model consists of a box in which walls can have different colors or materials and objects are placed inside. This simple scene demonstrates the effects of global illumination, such as inter-reflections/bounces, color bleeding, and soft shadows. In this regard, Test 3 consists of a series of scenes to showcase the effects of global illumination possible with the implemented BRDF. Table 5.5 presents the global parameters for the following renders. All scenes will be rendered with a 1920 × 1080 resolution, with the camera stationary and the FOV_y value at 45°. The rendering time of each scene will be 25 seconds and for the results, each scene will be rendered five times. Moreover, all scenes will be rendered with one to four bounces to appreciate the difference in the effects.

Table 5.5: Global Illumination Comparison - Testing Parameters

Resolution	**Rendering Time**	**Runs**	**FOV_y**	**Camera Movement**	**Bounces**	**BVH Bins**
1920 × 1080	25 s	5	45	No	1-4	11

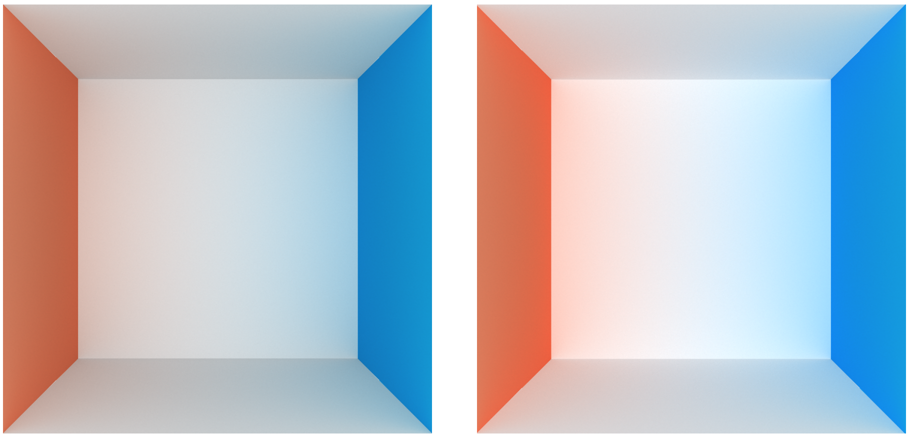
5.3.1 Color Bleeding

The first scenario presents the effect of color bleeding using diffuse surfaces. Figure 5.6 hows a Cornell box with a red left wall, a blue right wall, and white remaining walls. In the one-bounce case shown in Figure 5.6a, colors appear opaque and progressively darker toward the back of the box, and no color bleeding occurs. As shown in Figure 5.6b, increasing the number of bounces to two enhances the colors more and the depth of the box is no longer as dark as in the previous case. Looking carefully the effect of color bleeding is already present, yet is quite dim. However, when adding a third bounce (see Figure 5.6c), the effect visibility is higher, where the color of the red and blue walls is now reflected on the bottom, back, and the top box walls. Finally, as presented in Figure 5.6d, four bounces cause the scene to become brighter overall. The red and blue walls have become more shiny, and the color bleeding into the white walls is more pronounced and bright.



(a) 1 Bounce

(b) 2 Bounces



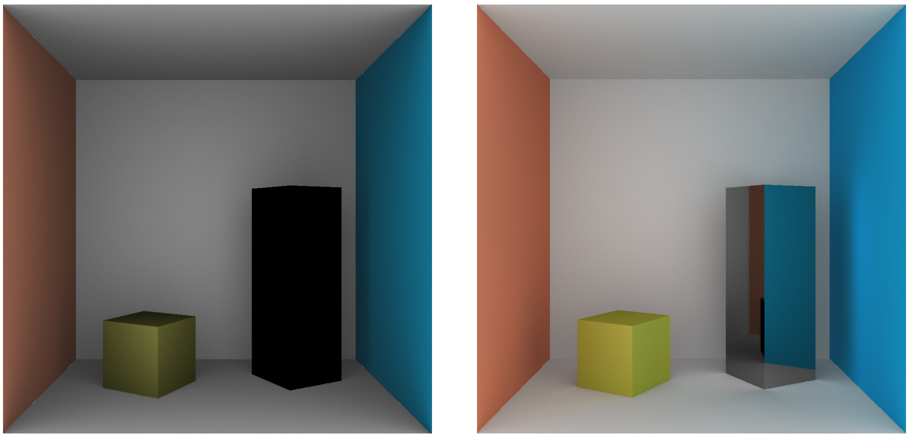
(c) 3 Bounces

(d) 4 Bounces

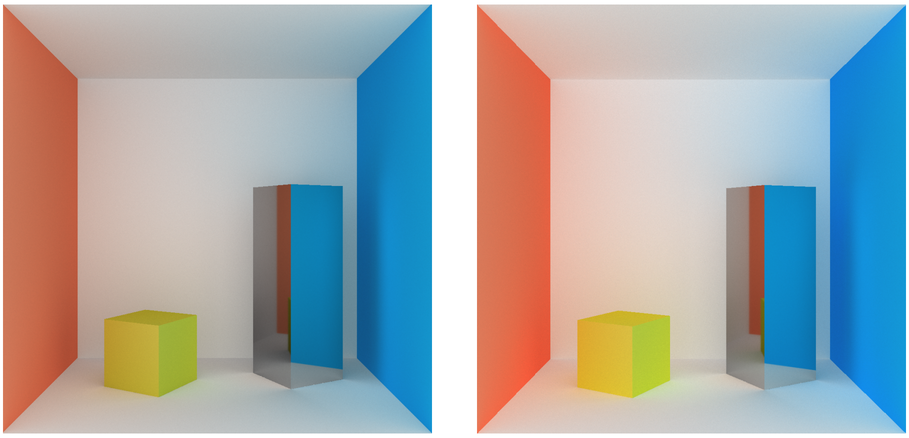
Figure 5.6: Color bleeding effect inside the Cornell box.

5.3.2 Soft Shadows and Specular Reflections

The second scenario adds two objects inside the Cornell box as illustrated in Figure 5.7. The first object is a diffuse yellow box on the left, while the second is a gray metal box on the right. Starting with only one bounce (see Figure 5.7a), the overall colors are opaque similar to the previous scenario. Despite this, the effect of soft shadows is already present behind the diffuse and metal boxes. However, it seems that the metal box is not rendered properly and is just a black box. This occurs because rays are only allowed to bounce once after being sent from the camera. As a result, rays will reflect from the metal box into one of the walls but will not get outside of the Cornell box. Consequently, it gives the effect that light is not reaching the metal box indirectly. Nonetheless, when adding a second bounce, Figure 5.7b, the rays reflected into the walls will be diffusely reflected outside the



(a) 1 Bounce (b) 2 Bounces



(c) 3 Bounces (d) 4 Bounces

Figure 5.7: Soft shadows and reflections caused by a yellow diffuse box and a gray metallic box placed on the left and right, respectively, inside the Cornell box.

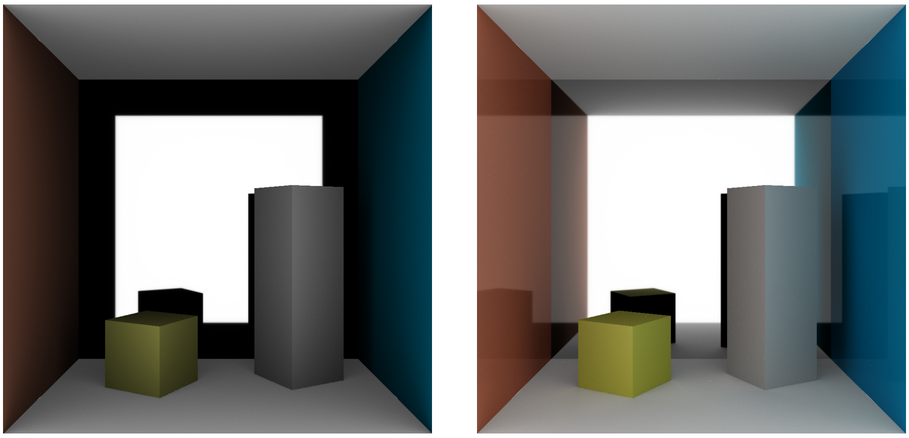
box reaching the light coming from the overall scene. At this point, the soft shadows are visible completely, yet the color reflected in the metal box is dim.

By increasing the number of bounces to three, Figure 5.7c, the overall brightness of all objects improves, the colors reflected in the metal box are no longer dull and the soft shadows are not as dark as in the previous rendering. Some color bleeding is being reflected into the diffuse yellow box having a dim orange and green color due to the red and blue walls, respectively. In addition, the soft shadows have acquired a color tint regarding the objects they cover and are not gray anymore. Lastly, setting the bounce parameter to four combines the color brightness reached in the diffuse-only scenario with all the objects introduced as presented in Figure 5.7d. The yellow box has a more perceptible color bleeding from the red and blue walls, having a brighter orange tone on the left face and a greenish tone on the right face. Likewise, the orange and green color bleeding reflecting on the floor has become more clear near the yellow box. And finally, the color reflections of the surrounding walls in the metal box have turned brighter, whereas the soft shadow tint has become more solid.

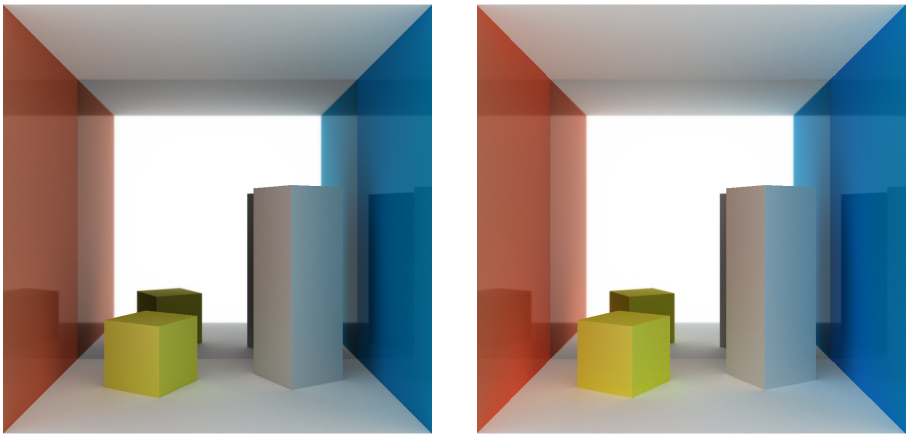
5.3.3 Combination of Effects

A third scene was created to combine all possible effects as illustrated in Figure 5.8. The red and blue walls now have glossy reflections; the back wall features a grey metallic material that acts like a mirror, and the metal box on the right has been replaced by a diffuse white box. The initial render with only one bounce, Figure 5.8a, shows an overall opaque scene, where the red and blue walls now have a darker hue compared to the previous scenarios analyzed. This occurs because, although the red and blue walls have a glossy finish, they are still dielectrics, meaning rays are still diffusely reflected. This allows some rays to escape the box and reach the light coming from the entire scene. Additionally, observe that a black silhouette appears in the metal wall in the back, due to the rays being specularly reflected into the two boxes inside or the walls around.

Setting the number of bounces to two (see Figure 5.8b) introduces more color into the scene; the glossy effect on the red and blue walls becomes evident, and some blue color bleeding appears in the white diffuse box on the left. Turning our attention to the metal



(a) 1 Bounce (b) 2 Bounces



(c) 3 Bounces (d) 4 Bounces

Figure 5.8: Combination of color bleeding, soft shadows, specular reflections, and glossiness effects.

wall at the back, the black silhouette is no longer present as adding a second bounce allows the rays to be diffusely reflected outside the box. As a result, the color of the red and blue walls is now reflected in the metal wall. Nevertheless, the color reflections of the yellow and white boxes are still black because once the rays specularly reflect into these objects these are diffusely reflected again into the back of the Cornell box. On the other hand, soft shadows have become evident behind the two diffuse boxes, being cast towards the red and blue walls. However, because of the glossiness effect of the side walls, soft shadows merge with the specular reflections of the diffuse boxes that appear in the red and blue walls.

Adding a third bounce, Figure 5.8c, shows the yellow color of the left diffuse box reflected in the metal back wall although still dim. Likewise, the color bleeding of the red and blue walls has become clear on the sides of the yellow box, and the blue color bleeding into the white box is more intense. Moreover, the glossiness effect of the red and blue walls has become more solid, and the reflections of the diffuse boxes appear brighter. Increasing the bounce number to four, as shown in Figure 5.8d, raises the brightness of all object colors in the scene. The glossy effect of the red and blue walls is more vibrant, intensifying the color bleeding into the yellow and white boxes. The orange and green hue is more distinguishable on the yellow box, and the white box now has some blending of red and yellow on its left, whereas the blueish tone on its right is stronger. Soft shadows have also become more unified with the color bleeding and glossy aspect of the red and blue walls. The colors of the side walls and boxes reflected in the metal back wall are also brighter. As a result, the composition of all objects and the interaction of all effects combine for a more realistic scene.

5.3.4 Performance Analysis

An analysis of the renderer's performance will conclude this section. Figure 5.9 presents the average frame rendering time of the different scenes in increasing order of bounces, whereas Table 5.6 shows the average FPS for the corresponding 25 s of rendering time across the five runs. In the first scene, it is noticed that the average frame rendering times do not surpass the 4 ms mark. With a single bounce, the average frame rendering time is 1.82 ms, but adding a second bounce increases it to 2.30 ms on average. This represents a relative increase of approximately 26.37% in average frame rendering time from one bounce up to two. Contrasting these values with the average FPS results shows a loss of 114.76 frames when adding a second bounce, going from 549.82 FPS to 435.06 FPS.

The difference in frame rendering times between three and two bounces has increased slightly, by 0.51 ms. The reason this increase is not more dramatic is that many rays are diffusely reflected out of the box after the second bounce, leaving fewer rays to reflect off another surface and requiring a third bounce to exit the box. Nonetheless, the drop in average FPS remains significant, at 78.60 FPS. The same tendency is observed by rendering the first scene with four bounces. The difference in frame rendering time compared to three bounces is 0.53 ms, corresponding to a decrease of 57.51 FPS. As such, the overall increase in brightness of the scene is caused by only a few rays. In summary, the cost of rendering

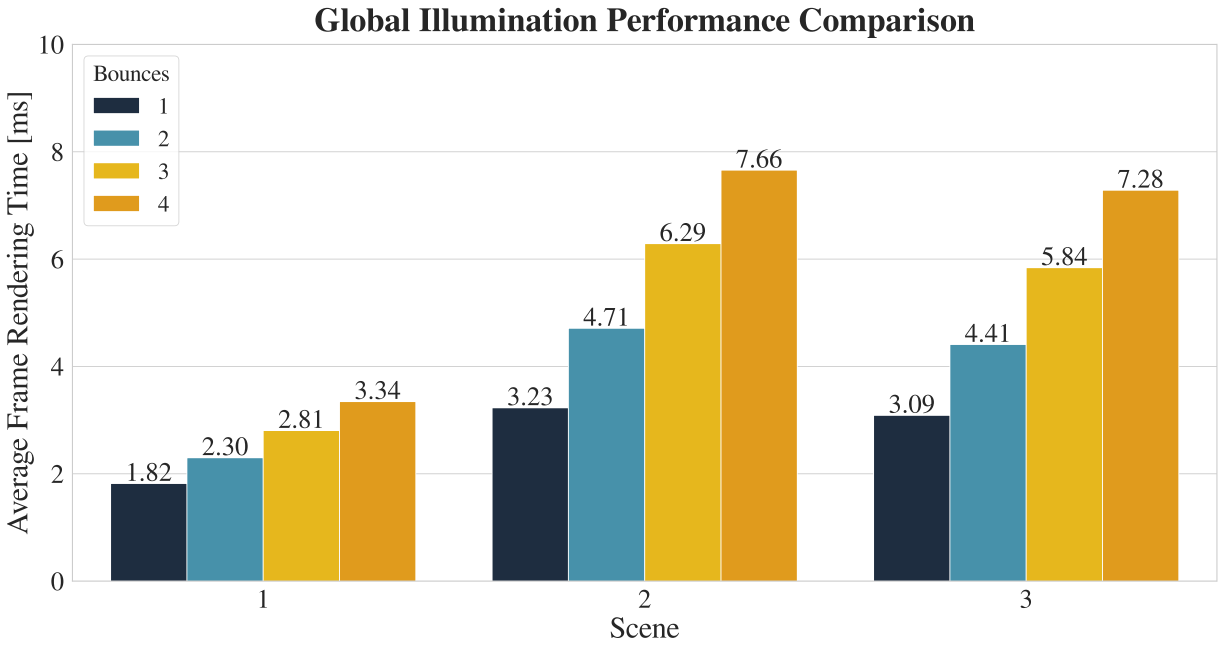


Figure 5.9: Average frame rendering time comparison of the different scenarios of global illumination. The results represent the average of the five executions for the three Cornell box scenes presented earlier.

Table 5.6: Average FPS comparison of the global illumination scenes for the five runs.

Scene	Bounces	Total Frames	Avg FPS
1	1	68727	549.82
	2	54382	435.06
	3	44557	356.46
	4	37369	298.95
2	1	38709	309.67
	2	26533	212.26
	3	19878	159.02
	4	16323	130.58
3	1	40475	323.80
	2	28353	226.82
	3	21405	171.24
	4	17163	137.30

with four bounces compared to just one is substantial, with a difference in average frame rendering time of 1.52 ms, resulting in a reduction of 250.87 FPS.

Adding two more objects into the scene increases frame rendering times higher, where the metal box at the right complicates calculations further. At only one bounce, the cost of rendering the second scene is 3.23 ms, representing a relative increase of approximately 77.47% in the average frame rendering time against the first scene with only one bounce. The tendency prevails when increasing the number of bounces. The difference between two bounces and one is 1.48 ms, mainly because various incident rays into the metal box are specularly reflected into one of the walls requiring an additional bounce to get out of the Cornell box. A third bounce allows a ray to be reflected by the metal box again into a diffuse wall or from a diffuse box back into the metal box. As a result, computing the radiance for a pixel becomes more expensive, since rays can enter a cycle of inter-reflections.

Lastly, adding a fourth bounce increases the average frame rendering time to 7.66 ms. This represents an increase of 4.43 ms in contrast to the render time with one bounce, which is 3.23 ms. Moreover, when comparing the average FPS of rendering scene two with one bounce to that with four, there is a reduction of 179.09 FPS, which is approximately a 2.37 times decrease in FPS. As a result, the average FPS of rendering the current scene has decreased considerably when compared to the first scene. One aspect observed in scene two is the problem of the inter-reflection cycle. The rays incident on the metal box can be specularly reflected onto the walls or the yellow diffuse box. This cycle can continue if the rays are then diffusely reflected again onto the metal box.

The inter-reflection problem was slightly reduced in the third scene, where the right metal box was replaced with a white diffuse material, and the back wall was changed to a grey metallic material. Although setting the red and blue walls to have a glossy finish might seem to contribute to the inter-reflection problem, the metallic back wall helps counteract this effect to some extent. With one bounce, the average frame rendering time is 3.09 ms, which is 0.14 ms longer than in scene two. This increase is partly because rays sent into the right white box may diffusely reflect outside with the first bounce, requiring no additional sampling direction as no further intersections occur. Another reason is that most rays sent into the back metal wall specularly reflect outside the box on the first bounce. In contrast, in the second scene, rays could diffusely reflect into the walls or one of the boxes, with the metal box being particularly problematic. As a result, the difference in average FPS between scenes two and three, rendered with one bounce, is 14.13 FPS.

Rendering scene three with two bounces extends the frame rendering time to 4.41 ms, approximately 1.43 times longer than with one bounce. Consequently, the average FPS drops significantly, with a decline of 96.98 FPS. When the number of bounces increases to three, the average frame rendering time climbs to 5.84 ms, leading to a drop in FPS from 226.82 with two bounces to 171.24 FPS. Lastly, adjusting the bounce parameter to four raises the average frame rendering time to 7.28 ms. The decrease in average FPS when comparing the scene with four bounces to one bounce is 186.50 FPS. As a result, the FPS with four bounces is reduced to roughly 2.36 times less than that with one bounce.

In conclusion, this section presents the global illumination effects possible with the implemented BRDF. The first scene benefits from the fact that there are no more objects inside the Cornell box so the number of illumination effects is reduced while keeping the frame rendering time low as opposed to the other scenes. The second scene on the other hand shows the highest frame rendering times due to the multiple inter-reflections caused by the metal box on the right. This problem is controlled slightly in the third scene by changing the right box material to be diffuse and the back wall to a metallic material. In this way, many of the rays reflected from the metal wall have the opportunity to leave the Cornell box with the next bounce. Therefore, it can be said that the position, orientation, and material of the objects present in a scene can contribute to or affect the frame rendering times of a scene and consequently the FPS generated.

5.4 Rendering a Complex Model

The Cornell box scenes presented above were simple, but the frame render times increased dramatically with the number of bounces. To put it in perspective, the first scene consisted of 10 triangles in total, 2 for each wall. The second and third scenes had 34 triangles in total, 12 triangles for each box added. Therefore, the current Test will analyze the frame rendering times of a more complex model formed by 7050 triangles. The model belongs to Robin Butler [163] and is part of a collection of science fiction building models, being the simplest of the three. Rendering a more complex model requires a much broader sampling of incident directions according to all the different materials comprising the model. Moreover, all the finer details comprising a specific part of a model are made of multiple triangles, and, thus, the amount of ray-intersection operations increases drastically in some portions of the scene. Computations become more complicated depending on the model's perspective and how much of the viewport space it covers.

Table 5.7: Complex Model Rendering Parameters

Resolution	**Rendering Time**	**Runs**	**FOV$_y$**	**Camera Movement**	**Bounces**	**BVH Bins**
1600×904	1 min	5	90	No	1-4	5

In this way, the rendering parameters for this model were changed as illustrated in Table 5.7. The resolution was reduced to 1600×904 to increase the number of FPS. This resolution ensures that the number of local x and y groups of the compute shader remains integer by dividing the image pixels by 4 and 8 in each dimension, as specified during Section 4.1. The rendering time selected was 1 minute. The FOV$_y$ value was increased to have a larger perspective of the model without being too far from it, while the camera was kept stationary. The number of bounces tested was 1 to 4, while the BVH number of bins to partition the primitives into new nodes was set to 5. This number of bins generated the best BVH for this model compared to other values. The scene was first rendered with a normal map, and then with the respective number of bounces.

Figure 5.12 presents the renders of the model taken after a minute of rendering. The rendering of the normal map can be found in Appendix G. The render with one bounce, Figure 5.12a, shows an overall opaque scene, where the objects on the porch of the building are barely visible. However, the stairs and porch access handrails are recognizable. Adding a second bounce, Figure 5.12b, introduces some yellow color bleeding reflecting from the ground onto the border walls and ceiling of the porch. The second bounce also allows more rays to reach the objects on the porch, making visible the yellow cylinders and white boxes on the left and right, respectively.

The third bounce, Figure 5.12c, brings more light into the porch allowing the objects to become brighter. The yellow cylinders on the right have a metallic material, the door has some white details, and, behind the white boxes, there is a tube connected to a black box above them. Two pads are visible left to the doorframe that can be used to open the door. The ground, the stairs, and the porch window frames have also gotten brighter. Finally, as shown in Figure 5.12d, the fourth causes the porch depth to become clearer and reduces the dark spots in the corners and window frames. When comparing all the renderings, the differences are significant, and the one with four bounces shows the most detail.

The realism achieved with the above renders, however, pushes the performance of the implemented renderer to its limits. Figure 5.10 shows the results of the average frame rendering times whereas Table 5.8 shows the results of the average FPS after 60 seconds of rendering. The normal map of the model has an average frame rendering time of 28.21 ms, resulting in an average of 38.22 FPS for the five runs. These results already demonstrate the computational expense of ray tracing a model with more than a thousand triangles. Compared to the second scene of the Cornell box model discussed earlier, where at four bounces the average frame rendering time was approximately 7.66 ms, rendering the current model with no bounces takes 3.68 times more. When considering the BRDF effects with one bounce, the average frame rendering time rises to 203.70 ms, which is approximately 7.22 times longer than with 0 bounces. This further highlights the significant increase in computational cost when rendering the 7050 triangles of the model after including path tracing and the BRDF sampling computations. Moreover, the impact on average FPS is drastic, dropping to 5.27 FPS, making the renderer no longer interactive.

From this point onwards the performance drops even more, where at two bounces

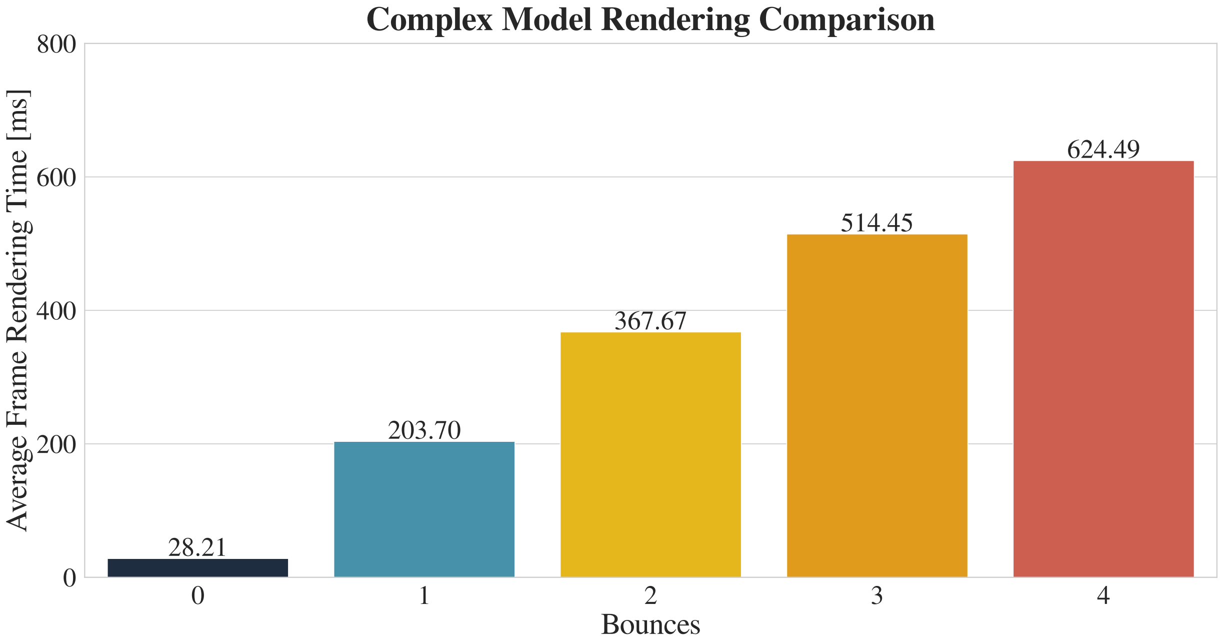


Figure 5.10: Average frame rendering time comparison of the science fiction building model across the five runs.

Table 5.8: Average FPS comparison of the science fiction building model renders across five runs.

Bounces	**Total Frames**	**Avg FPS**
0	11465	38.22
1	1581	5.27
2	872	2.91
3	622	2.07
4	512	1.71

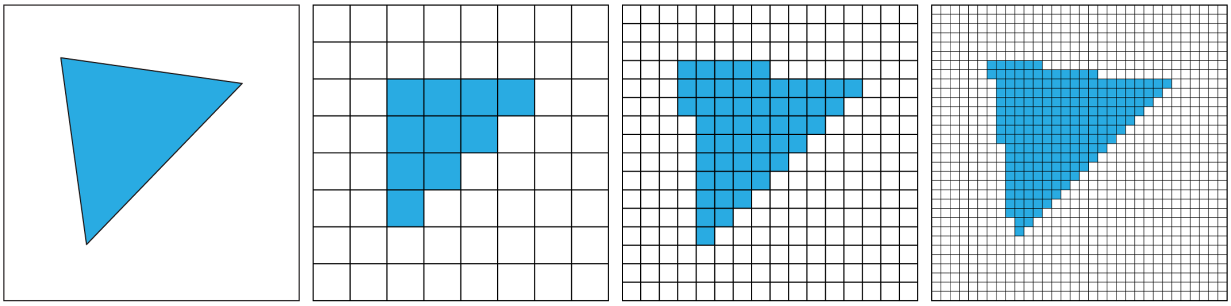
the scene already takes 367.67 ms on average to render a frame. When a new direction is sampled, the BVH traversal must be computed again for the new ray's direction to determine if a primitive intersection occurs. Although traversing a BVH is less costly than having no acceleration structure, it remains a significant operation, especially when multiple rays have previously intersected a surface. With two bounces, the average FPS reduces to 2.91, approximately 1.81 times less than the FPS generated with one bounce. Increasing the number of bounces to three and four raises the frame rendering times to 514.45 ms and 624.49 ms, respectively. Because most rays are reflected into the scene's infinite area light source after the second or third bounces, the average frame rendering times do not double. Consequently, as discussed in the previous section, the additional details added to the scene with the subsequent bounces are only handled by a smaller portion of rays. Rendering the scene with four bounces results in an average of 1.71 FPS, with only 512 frames generated after five minutes of rendering.

5.5 Limitations

The first limitation of the renderer is regarding light representations. As outlined in section 4.6.1, implementing diverse light source representations is complex because, depending on a light's shape, a point may be illuminated from multiple incident directions. Consequently, sampling for incident directions must account for the light source's shape, requiring more advanced sampling techniques [164–167]. A second aspect involves handling refractive or transmissive materials, which requires implementing a Bidirectional Transmittance Distribution Function (BTDF). A BTDF models how light passes through or refracts within transparent or semi-transparent materials, such as glass or water. Combining both functions enables the renderer to simulate complex interactions where light partially reflects and refracts at different angles, achieving a more realistic representation of materials that are both reflective and transmissive [11, 60].

The third limitation concerns noise during camera movement, which appears as pixels with null or unrelated information compared to neighboring pixels. Since incident directions are randomly sampled according to the BRDF, each frame generates ray paths distinct from those of previous frames. Additionally, because path tracing relies on Monte Carlo integration, pixel values across multiple frames are averaged to approximate the scene's steady-state illumination. In this regard, the renderer could benefit from implementing the denoising techniques reviewed in the state-of-the-art chapter. A fourth area worth exploring is aliasing, which refers to the jagged edges that can appear due to insufficient pixel sampling. Figure 5.11 illustrates the aliasing problem: while increasing the pixel resolution can make jagged edges less noticeable, the geometry will still have a discrete representation. In the current setup, one ray is cast from the camera into the center of each pixel at a specific image resolution. If an object's geometry is small or located far from the camera, it's possible that no ray will intersect with the object's surface, leading to missing details in the rendered image.

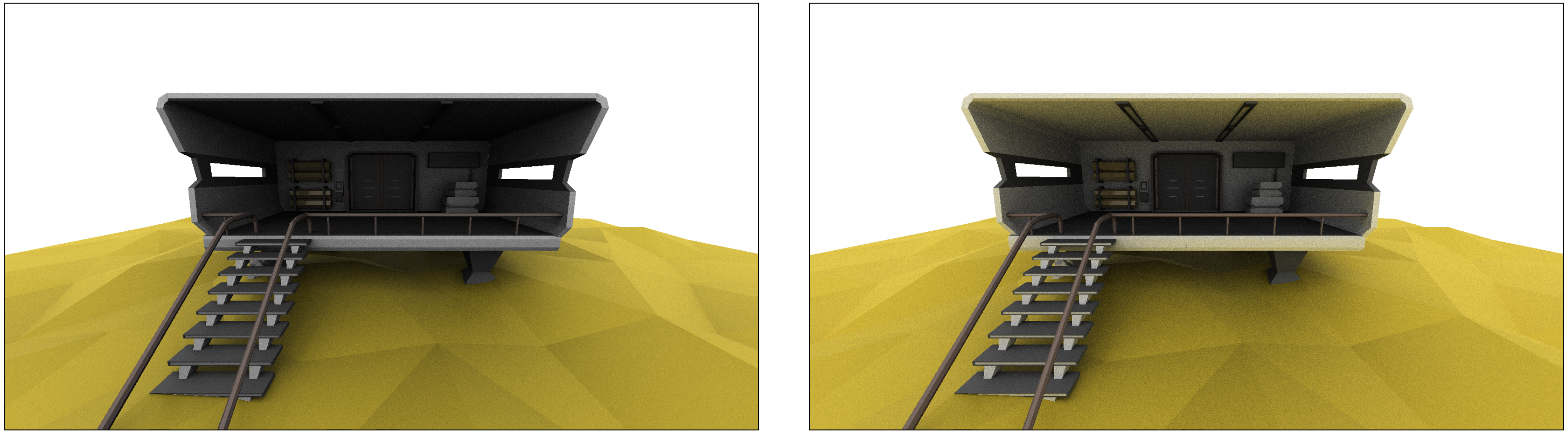
There are several anti-aliasing techniques designed to smooth or hide the appearance of jagged edges, three of them will be discussed below. Super-sampling anti-aliasing (SSAA), for example, involves casting multiple rays per pixel and averaging the resulting color



Original Triangle 8x8 16x16 32x32

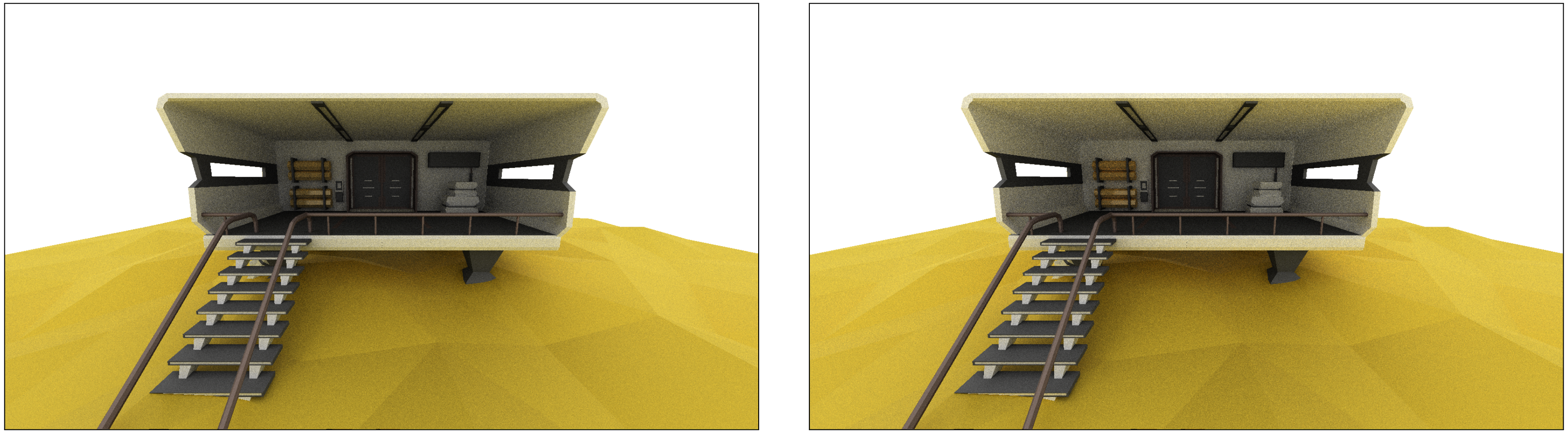
Figure 5.11: Triangle shown on grids of three different resolutions. On the discrete grid, the smooth geometry turns into stair steps denominated jaggies. No matter how high the resolution, the jaggies will not disappear, but will only become smaller.

values to determine the final pixel color, thereby reducing aliasing [7, 30]. Adaptive supersampling optimizes this process by casting additional rays only in regions with high contrast, reducing the computational load in areas of lower detail [134]. Finally, temporal anti-aliasing takes a different approach by blending samples from both the current and previous frames, with a slight camera jitter between them to ensure varied sample coverage [168]. However, any anti-aliasing technique must be implemented with real-time rendering performance in mind. In conclusion, the discussed limitations can be further addressed in a future work as to increase the capabilities of the renderer.



(a) 1 Bounce

(b) 2 Bounces



(c) 3 Bounces

(d) 4 Bounces

Figure 5.12: Renders of the science fiction building model.

Chapter 6

Conclusions

Ray tracing is a challenging problem requiring a lot of optimization to accomplish real-time rendering ensuring interactivity. Solving it mainly involves extensive knowledge of mathematics as well as programming. However, the limited number of cores in a CPU offers poor performance for generating multiple frames per second. In consequence, it is necessary to utilize the high parallelism offered by a GPU, which requires learning an API to handle it. This work uses OpenGL and compute shaders to generate images with ray tracing. Achieving photorealism further demands an understanding of the physical properties of light and its interaction with surfaces. In this sense, physics becomes a crucial aspect of realistic scene rendering with ray tracing. Therefore, ray tracing combines different fields, notions, and tools to create a fascinating solution for photorealistic rendering.

To familiarize the reader with the field of computer graphics, this work begins by introducing the key concepts required to implement a ray-tracing renderer. The first concept covered was the camera and how to create a model suitable for ray tracing. Next, the topic of light was addressed, including the notion of color and its computational representation. After these, the relationship between a ray and the forward and backward ray tracing models used to render an image was explained. This was followed by a discussion of the essential mathematical tools needed to implement a camera within a scene. Finally, the importance of coordinate systems was examined, highlighting their role in generating an image from the camera's perspective based on its position within the scene. With these fundamentals established, a summary of the state of the art in ray tracing was provided, leading to a discussion of the specific problems this work addresses.

The first aspect covered in this work was a solution for rendering graphic primitives, specifically spheres and triangles. Rendering these primitives with ray tracing requires analyzing their mathematical representations along with that of the ray. An equation is defined to determine the intersection points between the ray and the primitive, which constitutes the solution to the problem. Various algorithms exist to find these solutions, with some being more optimal than others. In this work, the optimal algorithms recommended in the literature were implemented. While many other primitives can be added, spheres are commonly used due to their simplicity in ray tracing, and triangles are essential for constructing 3D models.

While primitive rendering is an essential feature, adding multiple objects to the scene increases frame rendering times. The reason is that all rays sent from the camera into the scene must be checked for a primitive intersection. As such, intersection operations increase with multiple objects, and computational resources are wasted if no intersection is found. The second problem addressed in this work was to provide an acceleration data structure to reduce the number of intersection tests and increase the renderer's performance. The solution implemented is known as a bounding volume hierarchy (BVH), which encloses groups of primitives into axis-aligned bounding volumes (AABB), forming a hierarchy represented by a binary tree. The root node represents the bounding volume that encloses all primitives, the internal nodes contain a subset of the primitives contained in a much smaller bounding volume, and the leaf nodes contain either one primitive or a smaller subset of primitives. This approach significantly reduces the number of ray intersection operations, as primitives within a non-intersected AABB are skipped.

The third aspect implemented in the ray tracing renderer was a solution for the global illumination problem to bring realistic lighting into the scene. Global illumination aims at computing the steady state between direct and indirect illumination. Radiometry provides the tools to measure light radiation, the most important being the bidirectional reflectance distribution function (BRDF). When coupled with microfacet theory, a physically-based BRDF can be created to model light reflection of surfaces, specifically dielectrics, and metals. The last step was to convey light from one point to another using rays. Path tracing was the solution implemented in this work to approximate the total radiance at a surface point regarding the hemisphere of light incident directions. This technique is based on Monte Carlo integration to sample incident directions according to the implemented BRDF and then approximates the color of the pixels to render the image.

The performance achieved with the implemented concepts depends primarily on the scene and the number of objects within it. An average above 100 FPS was obtained with simple scenes like the Cornell box, accounting for all global illumination effects possible with the implemented BRDF. In this regard, the rendering process was in real-time and highly interactive, allowing the camera to move smoothly across the scene and display different perspectives. It was also observed that BVH traversal algorithms have a direct impact on the renderer's performance. Comparing the algorithms implemented in this work showed that the ray-direction sign-based method was faster than the direction-based method. However, the renderer's performance drops significantly when several objects are present. Accounting for both global illumination effects and multiple bounces drastically reduces performance, making it no longer interactive, though still much faster if compared to CPU rendering. Overall, the ray-tracing renderer remains solid for understanding real-time and physically-based rendering to generate photorealistic lighting in a scene. Ultimately, this thesis presents several of the fundamentals of ray tracing laying the foundation for a specialization in computer graphics.

Bibliography

[1] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-down Approach with WebGL.* Pearson, 2015. [Online]. Available: https://books.google.com.ec/books?id=9K0nngEACAAJ

[2] J. de Vries, *Learn OpenGL: Learn Modern OpenGL Graphics Programming in a Step-by-step Fashion.* Kendall & Welling, 2020. [Online]. Available: https://books.google.com.ec/books?id=koWWzQEACAAJ

[3] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering.* AK Peters/CRC Press, 2019.

[4] Fulvio34 (https://commons.wikimedia.org/wiki/User:Fulvio314), "Light spectrum (precise colors)," 2016, licensed under CC BY-SA 4.0. [Online]. Available: https://en.wikipedia.org/wiki/File:Light_spectrum_(precise_colors).svg#metadata

[5] A. D. Logvinenko, "The color cone," *JOSA A*, vol. 32, no. 2, pp. 314–322, 2015.

[6] E. Maldonado, "Spheres in a cornell box," 2013. [Online]. Available: https://blenderartists.org/t/spheres-in-a-cornell-box/592505

[7] A. S. Glassner, *An introduction to ray tracing.* Morgan Kaufmann, 1989.

[8] E. Lengyel, *Foundations of Game Engine Development. Volume 2: Rendering*, 4th ed. Terathon Software, 2019.

[9] S. Ho, "OpenGL Projection Matrix," Accessed on May 20, 2024. [Online]. Available: https://www.songho.ca/opengl/gl_projectionmatrix.html

[10] T. Möller and B. Trumbore, “Fast, minimum storage ray/triangle intersection,” in *ACM SIGGRAPH 2005 Courses.* ACM New York, NY, USA, 2005, pp. 7–es.

[11] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation.* MIT Press, 2023. [Online]. Available: https://pbr-book.org

[12] F. Ganovelli, M. Corsini, S. Pattanaik, and M. Di Benedetto, *Introduction to Computer Graphics: A Practical Learning Approach*, 1st ed. CRC Press, 2015.

[13] E. Dinur, *The Complete Guide to Photorealism for Visual Effects, Visualization and Games.* Routledge, 2021.

[14] P. Dutre, P. Bekaert, and K. Bala, *Advanced global illumination.* AK Peters/CRC Press, 2018.

[15] E. K. Henriksen, C. Angell, A. I. Vistnes, and B. Bungum, “What is light? students’ reflections on the wave-particle duality of light and the nature of physics,” *Science & education*, vol. 27, pp. 81–111, 2018.

[16] A. Appel, “The notion of quantitative invisibility and the machine rendering of solids,” in *Proceedings of the 1967 22nd national conference*, 1967, pp. 387–393.

[17] R. A. Goldstein and R. Nagel, “3-d visual simulation,” *Simulation*, vol. 16, no. 1, pp. 25–31, 1971.

[18] R. L. Cook, T. Porter, and L. Carpenter, “Distributed ray tracing,” in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 1984, pp. 137–145.

[19] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, “State of the art in ray tracing animated scenes,” in *Computer graphics forum*, vol. 28, no. 6. Wiley Online Library, 2009, pp. 1691–1722.

[20] A. L. Dos Santos, D. Lemos, J. E. F. Lindoso, and V. Teichrieb, “Real time ray tracing for augmented reality,” in *2012 14th Symposium on Virtual and Augmented Reality.* IEEE, 2012, pp. 131–140.

[21] A. Siekawa, M. Chwesiuk, R. Mantiuk, and R. Piórkowski, "Foveated ray tracing for vr headsets," in *MultiMedia Modeling: 25th International Conference, MMM 2019, Thessaloniki, Greece, January 8–11, 2019, Proceedings, Part I 25.* Springer, 2019, pp. 106–117.

[22] Y. Pulijala, M. Ma, and A. Ayoub, "Vr surgery: Interactive virtual reality application for training oral and maxillofacial surgeons using oculus rift and leap motion," *Serious Games and Edutainment Applications: Volume II*, pp. 187–202, 2017.

[23] A. A. Rad, R. Vardanyan, A. Lopuszko, C. Alt, I. Stoffels, B. Schmack, A. Ruhparwar, K. Zhigalov, A. Zubarevich, and A. Weymann, "Virtual and augmented reality in cardiac surgery," *Brazilian journal of cardiovascular surgery*, vol. 37, no. 01, pp. 123–127, 2022.

[24] A. E. Hassanien, D. Gupta, A. Khanna, and A. Slowik, *Virtual and Augmented Reality for Automobile Industry: Innovation Vision and Applications.* Springer, 2022.

[25] M. E. Portman, A. Natapov, and D. Fisher-Gewirtzman, "To go where no man has gone before: Virtual reality in architecture, landscape architecture and environmental planning," *Computers, Environment and Urban Systems*, vol. 54, pp. 376–384, 2015.

[26] S. Ahmed, M. M. Hossain, and M. I. Hoque, "A brief discussion on augmented reality and virtual reality in construction industry," *Journal of System and Management Sciences*, vol. 7, no. 3, pp. 1–33, 2017.

[27] L. P. Berg and J. M. Vance, "Industry use of virtual reality in product design and manufacturing: a survey," *Virtual reality*, vol. 21, pp. 1–17, 2017.

[28] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of computer graphics*, 4th ed. AK Peters/CRC Press, 2016.

[29] "Ieee standard for head-mounted display (hmd)-based virtual reality(vr) sickness reduction technology," *IEEE Std 3079-2020*, pp. 1–74, 2021.

[30] P. Shirley, T. D. Black, and S. Hollasch, “Ray tracing in one weekend,” August 2023, `https://raytracing.github.io/books/RayTracingInOneWeekend.html`. [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html

[31] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, “A data distributed, parallel algorithm for ray-traced volume rendering,” in *Proceedings of the 1993 symposium on Parallel rendering*, 1993, pp. 15–22.

[32] E. Reinhard and F. W. Jansen, “Rendering large scenes using parallel ray tracing,” *Parallel Computing*, vol. 23, no. 7, pp. 873–885, 1997.

[33] G. Johansson, O. Nilsson, A. Söderström, and K. Museth, “Distributed ray tracing in an open source environment (work in progress),” in *SIGRAD 2006. The Annual SIGRAD Conference, Special Theme: Computer Games, November 22–23, 2006, Skövde, Sweden.* Linköping University Electronic Press, 2006, pp. 7–11.

[34] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, “Realtime ray tracing on gpu with bvh-based packet traversal,” in *2007 IEEE Symposium on Interactive Ray Tracing.* IEEE, 2007, pp. 113–118.

[35] S. G. Parker, H. Friedrich, D. Luebke, K. Morley, J. Bigler, J. Hoberock, D. McAllister, A. Robison, A. Dietrich, G. Humphreys *et al.*, “Gpu ray tracing,” *Communications of the ACM*, vol. 56, no. 5, pp. 93–101, 2013.

[36] M. Castorina and G. Sassone, *Mastering Graphics Programming with Vulkan: Develop a modern rendering engine from first principles to state-of-the-art techniques.* Packt Publishing Ltd, 2023.

[37] A. Marrs, P. Shirley, and I. Wald, Eds., *Ray Tracing Gems II.* Apress, 2021, http://raytracinggems.com/rtg2.

[38] Khronos Group, “OpenGL - The Industry’s Foundation for High Performance Graphics,” Accessed on March 11, 2024. [Online]. Available: https://www.opengl.org/

[39] Microsoft, “Directx developer blog,” Accessed on March 11, 2024. [Online]. Available: https://devblogs.microsoft.com/directx/

[40] Khronos Group, “Vulkan | Cross Platform 3D Graphics,” Accessed on March 11, 2024. [Online]. Available: https://www.vulkan.org/

[41] Apple, “Metal Overview - Apple Developer,” Accessed on March 11, 2024. [Online]. Available: https://developer.apple.com/metal/

[42] Nvidia, “CUDA Toolkit,” Accessed on March 11, 2024. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[43] G. Sellers and J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan.* Addison-Wesley Professional, 2016, p. 21.

[44] A. R. Smith, “A pixel is not a little square, a pixel is not a little square, a pixel is not a little square!” *Microsoft Computer Graphics, Technical Memo*, vol. 6, 1995.

[45] H. D. Young, R. A. Freedman, A. L. Ford, and F. W. Sears, *Sears and Zemansky’s University Physics: with Modern Physics*, 13th ed. San Francisco: Pearson Addison-Wesley, 2012, ch. The Nature and Propagation of Light.

[46] M. Kalloniatis and C. Luu, “The perception of color,” in *Webvision: The Organization of the Retina and Visual System*, H. Kolb, E. Fernandez, and R. Nelson, Eds. Salt Lake City, UT: University of Utah Health Sciences Center, 1995, updated on 2007 Jul 9. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK11538/

[47] E. H. Land, “The retinex theory of color vision,” *Scientific american*, vol. 237, no. 6, pp. 108–129, 1977.

[48] M. C. Stone and M. Stone, “A survey of color for computer graphics,” *Course at SIGGRAPH*, vol. 744, 2001.

[49] K. Plataniotis and A. N. Venetsanopoulos, *Color image processing and applications.* Springer Science & Business Media, 2000.

[50] S. Westland and T. L. V. Cheung, “Rgb systems.” 2012.

[51] P. Shirley, I. Wald, T. Akenine-Möller, and E. Haines, “What is a ray?” in *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, E. Haines and T. Akenine-Möller, Eds. Apress, 2019, pp. 15–19. [Online]. Available: https://doi.org/10.1007/978-1-4842-4427-2_2

[52] J. Bloomenthal and J. Rokne, “Homogeneous coordinates,” *The Visual Computer*, vol. 11, pp. 15–26, 1994.

[53] V. Skala, "Barycentric coordinates computation in homogeneous coordinates," *Computers & Graphics*, vol. 32, no. 1, pp. 120–127, 2008.

[54] I. Shafarevich, *Basic Algebraic Geometry 1 - Vars. in Projective Space*, 2nd ed. Springer, 1988, p. 41.

[55] H. S. M. Coxeter, "What is Projective Geometry?" in *Projective geometry.* Springer Science & Business Media, 2003.

[56] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision.* Cambridge university press, 2003, p. 3.

[57] G. Gambetta, "Camera Transform," in *Computer Graphics from Scratch: A Programmer's Introduction to 3D Rendering*, 1st ed. No Starch Press, 2021, pp. 122–124.

[58] S. J. D. MacIntosh, "Generalized projection matrices," 2022.

[59] R. Kooima, "Generalized perspective projection," *J. Sch. Electron. Eng. Comput. Sci*, vol. 6, no. 1, 2009.

[60] J. F. Hughes, *Computer graphics: principles and practice*, 3rd ed. Addison-Wesley Professional, 2014.

[61] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Addison-Wesley Professional, 2013.

[62] S. Beck, A. Bernstein, D. Danch, and B. Fröhlich, "Cpu-gpu hybrid real time ray tracing framework," *The Eurographics Association and Blackwell Publishing Ltd*, pp. 1–8, 1981.

[63] T. L. Sabino, P. Andrade, E. W. Gonzales Clua, A. Montenegro, and P. Pagliosa, "A hybrid gpu rasterized and ray traced rendering pipeline for real time rendering of per pixel effects," in *Entertainment Computing-ICEC 2012: 11th International Conference, ICEC 2012, Bremen, Germany, September 26-29, 2012. Proceedings 11.* Springer, 2012, pp. 292–305.

[64] S. Hargreaves and M. Harris, "Deferred shading," in *Game Developers Conference*, vol. 2, 2004, p. 31.

[65] F. Policarpo, F. Fonseca, and C. Games, "Deferred shading tutorial," *Pontifical Catholic University of Rio de Janeiro*, vol. 31, p. 32, 2005.

[66] C. Barré-Brisebois, H. Halén, G. Wihlidal, A. Lauritzen, J. Bekkers, T. Stachowiak, and J. Andersson, "Hybrid rendering for real-time ray tracing," *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pp. 437–473, 2019.

[67] D. Koch, "Ray tracing in vulkan," Dec 2020, Accessed on July 3, 2024. [Online]. Available: https://www.khronos.org/blog/ray-tracing-in-vulkan#Acceleration_Structures

[68] "Directx raytracing (dxr) functional spec," 2023, Accessed on July 3, 2024. [Online]. Available: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

[69] J.-H. Nah, J.-W. Kim, J. Park, W.-J. Lee, J.-S. Park, S.-Y. Jung, W.-C. Park, D. Manocha, and T.-D. Han, "Hart: A hybrid architecture for ray tracing animated scenes," *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, vol. 21, no. 3, 2015.

[70] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister *et al.*, "Renderman: An advanced path-tracing architecture for movie rendering," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 3, pp. 1–21, 2018.

[71] Unity, "High definition render pipeline," Accessed on May 25, 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/index.html

[72] Unreal Engine, "Real-time ray tracing," Accessed on May 25, 2024. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/real-time-ray-tracing?application_version=4.27

[73] Blender, "Rendering," Accessed on May 25, 2024. [Online]. Available: https://www.blender.org/features/rendering/

[74] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o (n log n)," in *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006, pp. 61–69.

[75] M. Vinkler, V. Havran, and J. Bittner, "Bounding volume hierarchies versus kd-trees on contemporary many-core architectures," in *Proceedings of the 30th Spring Conference on Computer Graphics*, 2014, pp. 29–36.

[76] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," in *Computer Graphics Forum*, vol. 40, no. 2. Wiley Online Library, 2021, pp. 683–712.

[77] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.

[78] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[79] J. Pantaleoni and D. Luebke, "Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry," in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 87–95.

[80] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster hlbvh with work queues," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 2011, pp. 59–64.

[81] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2007, pp. 33–40.

[82] T. Karras, "Maximizing parallelism in the construction of bvhs, octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, 2012, pp. 33–37.

[83] C. Apetrei, "Fast and simple agglomerative lbvh construction," 2014.

[84] F. M. Chitalu, C. Dubach, and T. Komura, "Binary ostensibly-implicit trees for fast collision detection," in *Computer Graphics Forum*, vol. 39, no. 2. Wiley Online Library, 2020, pp. 509–521.

[85] J. O'Rourke, "Finding minimal enclosing boxes," *International journal of computer & information sciences*, vol. 14, pp. 183–199, 1985.

[86] S. A. Gottschalk, *Collision queries using oriented bounding boxes.* The University of North Carolina at Chapel Hill, 2000.

[87] C.-T. Chang, B. Gorissen, and S. Melchior, “Fast oriented bounding box optimization on the rotation group so (3, r),” 2012.

[88] T. Larsson and L. Källberg, “Fast computation of tight-fitting oriented bounding boxes,” *Game Engine Gems 2*, p. 1, 2011.

[89] P. Konecnỳ and K. Zikan, “Lower bound of distance in 3d,” in *Proceedings of WSCG*, vol. 3, 1997, pp. 640–649.

[90] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan, “Efficient collision detection using bounding volume hierarchies of k-dops,” *IEEE transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[91] M. Káčerik and J. Bittner, “Sah-optimized k-dop hierarchies for ray tracing,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 7, no. 3, pp. 1–16, 2024.

[92] N. Vitsas, I. Evangelou, G. Papaioannou, and A. Gkaravelis, “Parallel transformation of bounding volume hierarchies into oriented bounding box trees,” in *Computer Graphics Forum*, vol. 42, no. 2. Wiley Online Library, 2023, pp. 245–254.

[93] R. Sabino, C. A. Vidal, J. B. Cavalcante-Neto, and J. G. R. Maia, “Building oriented bounding boxes by the intermediate use of odops,” *Computers & Graphics*, vol. 116, pp. 251–261, 2023.

[94] D. Meister and J. Bittner, “Parallel locally-ordered clustering for bounding volume hierarchy construction,” *IEEE transactions on visualization and computer graphics*, vol. 24, no. 3, pp. 1345–1353, 2017.

[95] J. T. Kajiya, “The rendering equation,” in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[96] E. P. Lafortune and Y. D. Willems, “Bi-directional path tracing,” 1993.

[97] E. Veach and L. J. Guibas, “Metropolis light transport,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997, pp. 65–76.

[98] S. Bako, T. Vogels, B. McWilliams, M. Meyer, J. Novák, A. Harvill, P. Sen, T. Derose, and F. Rousselle, "Kernel-predicting convolutional networks for denoising monte carlo renderings." *ACM Trans. Graph.*, vol. 36, no. 4, pp. 97–1, 2017.

[99] A. Firmino, J. R. Frisvad, and H. W. Jensen, "Progressive denoising of monte carlo rendered images," in *Computer Graphics Forum*, vol. 41, no. 2. Wiley Online Library, 2022, pp. 1–11.

[100] H. E. Rushmeier and G. J. Ward, "Energy preserving non-linear filters," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 1994, pp. 131–138.

[101] N. K. Kalantari and P. Sen, "Removing the noise in monte carlo rendering with general image denoising algorithms," in *Computer Graphics Forum*, vol. 32, no. 2pt1. Wiley Online Library, 2013, pp. 93–102.

[102] R. Xu and S. N. Pattanaik, "A novel monte carlo noise reduction operator," *IEEE Computer Graphics and Applications*, vol. 25, no. 2, pp. 31–35, 2005.

[103] M. D. McCool, "Anisotropic diffusion for monte carlo noise reduction," *ACM Transactions on Graphics (TOG)*, vol. 18, no. 2, pp. 171–194, 1999.

[104] H. Dammertz, D. Sewtz, J. Hanika, and H. P. Lensch, "Edge-avoiding a-trous wavelet transform for fast global illumination filtering," in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 67–75.

[105] P. Bauszat, M. Eisemann, and M. Magnor, "Guided image filtering for interactive high-quality global illumination," in *Computer Graphics Forum*, vol. 30, no. 4. Wiley Online Library, 2011, pp. 1361–1368.

[106] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi, "Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination," in *Proceedings of High Performance Graphics*, ser. HPG '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3105762.3105770

[107] C. Schied, C. Peters, and C. Dachsbacher, "Gradient estimation for real-time adaptive temporal filtering," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 1, no. 2, pp. 1–16, 2018.

[108] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 148–1, 2020.

[109] C. Wyman, M. Kettunen, D. Lin, B. Bitterli, C. Yuksel, W. Jarosz, and P. Kozlowski, "A gentle introduction to restir path reuse in real-time," in *ACM SIGGRAPH 2023 Courses*. ACM New York, NY, USA, 2023, pp. 1–38.

[110] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, "Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, pp. 1–12, 2017.

[111] B. Xu, J. Zhang, R. Wang, K. Xu, Y.-L. Yang, C. Li, and R. Tang, "Adversarial monte carlo denoising with conditioned auxiliary feature modulation." *ACM Trans. Graph.*, vol. 38, no. 6, pp. 224–1, 2019.

[112] N. Hofmann, J. Martschinke, K. Engel, and M. Stamminger, "Neural denoising for path tracing of medical volumetric data," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, pp. 1–18, 2020.

[113] NVIDIA, "Nvidia | world leader in ai computing," Accessed on June 2, 2024. [Online]. Available: https://www.nvidia.com/en-us/

[114] AMD, "Amd | together we advance ai," Accessed on June 2, 2024. [Online]. Available: https://www.amd.com/en.html

[115] M. Jarzynski and M. Olano, "Hash functions for gpu rendering," *UMBC Computer Science and Electrical Engineering Department*, 2020.

[116] TheCherno, "Opengl," 2017, youTube Series. [Online]. Available: https://www.youtube.com/playlist?list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2

[117] M. Grieco, "C++ opengl tutorial," 2020, youTube Series. [Online]. Available: https://www.youtube.com/playlist?list=PLysLvOneEETPlOI_PI4mJnocqIpr2cSHS

[118] GetIntoGameDev, “Opengl with c++,” 2023, youTube Series. [Online]. Available: https://www.youtube.com/playlist?list=PLn3eTxaOtL2PHxN8EHf-ktAcN-sGETKfw

[119] B.-T. Phong, “Illumination for computer generated pictures,” *CACM*, 1975.

[120] S. Cook, *CUDA programming: a developer’s guide to parallel computing with GPUs.* Morgan Kaufmann, 2012.

[121] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley, 2010.

[122] Khronos Group, “Opencl - the open standard for parallel programming of heterogeneous systems,” 2013, Accessed on June 28, 2024. [Online]. Available: https://www.khronos.org/opencl/

[123] E. Haines, J. Günther, and T. Akenine-Möller, “Precision improvements for ray/sphere intersection,” *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pp. 87–94, 2019. [Online]. Available: https://doi.org/10.1007/978-1-4842-4427-2_7

[124] L. E. S. Stephen H. Friedberg, Arnold J. Insel, *Linear Algebra (5th Edition)*, 5th ed. Pearson Education, 2019.

[125] D. M. Heffernan and S. Pouryahya, “Scalar and vector triple products,” Accessed on June 20, 2024. [Online]. Available: http://www.thphys.nuim.ie/Notes/EE112/04_Triple_Products.pdf

[126] A. Majercik, C. Crassin, P. Shirley, and M. McGuire, “A ray-box intersection algorithm and efficient dynamic voxel rendering,” *Journal of Computer Graphics Techniques Vol*, vol. 7, no. 3, pp. 66–81, 2018.

[127] S. Woop, C. Benthin, I. Wald, G. S. Johnson, and E. Tabellion, “Exploiting local orientation similarity for efficient ray traversal of hair and fur.” *High Performance Graphics*, vol. 3, 2014.

[128] S. Widmer, D. Pająk, A. Schulz, K. Pulli, J. Kautz, M. Goesele, and D. Luebke, “An adaptive acceleration structure for screen-space ray tracing,” in *Proceedings of the 7th Conference on High-Performance Graphics*, 2015, pp. 67–76.

[129] T. L. Kay and J. T. Kajiya, “Ray tracing complex scenes,” *ACM SIGGRAPH computer graphics*, vol. 20, no. 4, pp. 269–278, 1986.

[130] S. H. Peter Shirley, Trevor David Black. (2024, April) Ray tracing: The next week. [Online]. Available: https://raytracing.github.io/books/RayTracingTheNextWeek.html

[131] A. Fujimoto, T. Tanaka, and K. Iwata, “Arts: Accelerated ray-tracing system,” *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16–26, 1986.

[132] T. Whitted, “An improved illumination model for shaded display,” in *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, 1979, p. 14.

[133] S. M. Rubin and T. Whitted, “A 3-dimensional representation for fast rendering of complex scenes,” in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 1980, pp. 110–116.

[134] A. S. Glassner, “Space subdivision for fast ray tracing,” *IEEE Computer Graphics and applications*, vol. 4, no. 10, pp. 15–24, 1984.

[135] T. Ize, I. Wald, and S. G. Parker, “Ray tracing with the bsp tree,” in *2008 IEEE Symposium on Interactive Ray Tracing.* IEEE, 2008, pp. 159–166.

[136] M. Vinkler, V. Havran, and J. Bittner, “Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing,” in *Computer Graphics Forum*, vol. 35, no. 8. Wiley Online Library, 2016, pp. 68–79.

[137] “Ray tracing,” Accessed on July 3, 2024. [Online]. Available: https://developer.nvidia.com/discover/ray-tracing

[138] Get Into Game Dev, “Surface area heuristic,” 2023, Implementation of the SAH for BVH construction. Accessed on July 5, 2024. [Online]. Available: https://github.com/amengede/getIntoGameDev/tree/main/Realtime%20Raytracing/python/15%20Surface%20Area%20Heuristic/4%20data%20oriented%20design

[139] J. Goldsmith and J. Salmon, “Automatic creation of object hierarchies for ray tracing,” *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, 1987.

[140] J. D. MacDonald and K. S. Booth, “Heuristics for ray tracing using space subdivision,” *The Visual Computer*, vol. 6, pp. 153–166, 1990.

[141] E. Seneta, K. H. Parshall, and F. Jongmans, "Nineteenth-century developments in geometric probability: Jj sylvester, mw crofton, j.-e. barbier, and j. bertrand," *Archive for history of exact sciences*, vol. 55, no. 6, pp. 501–524, 2001.

[142] M. J. Evans and J. S. Rosenthal, *Probability and Statistics.* Self-published, 2004. [Online]. Available: https://www.utstat.toronto.edu/mikevans/jeffrosenthal/book.pdf

[143] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 89–99.

[144] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, pp. 6–es, 2007.

[145] P. S. Heckbert and G. I. Course, "Introduction to global illumination," *SIGGRAPH 92 Global Illumination course notes*, vol. 18, 1992.

[146] S. A. Antinozzi, "An overview of modern global illumination," *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal*, vol. 4, no. 2, p. 1, 2017.

[147] D. Fleet and A. Herrtzmann, "Radiometry and reflection," 2005, Lecture Notes CSC418/CSCD18/CSC2504, University of Toronto. [Online]. Available: https://www.cs.toronto.edu/~jepson/csc2503/readings/Lighting.pdf

[148] P. Debevec, "Image-based lighting," in *ACM SIGGRAPH 2006 Courses.* ACM New York, NY, USA, 2006, pp. 4–es.

[149] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 1977, pp. 192–198.

[150] K. Torrance and E. Sparrow, "Theory for off-specular reflection from roughened surfaces," *Journal of the Optical Society of America (1917-1983)*, vol. 57, no. 9, p. 1105, 1967.

[151] C. Schlick, "An inexpensive brdf model for physically-based rendering," in *Computer graphics forum*, vol. 13, no. 3. Wiley Online Library, 1994, pp. 233–246.

[152] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, "Microfacet models for refraction through rough surfaces." *Rendering techniques*, vol. 2007, p. 18th, 2007.

[153] R. L. Cook and K. E. Torrance, “A reflectance model for computer graphics,” *ACM Transactions on Graphics (ToG)*, vol. 1, no. 1, pp. 7–24, 1982.

[154] P. Beckmann and A. Spizzichino, “The scattering of electromagnetic waves from rough surfaces,” *Norwood*, 1987.

[155] T. Trowbridge and K. P. Reitz, “Average irregularity representation of a rough surface for ray reflection,” *JOSA*, vol. 65, no. 5, pp. 531–536, 1975.

[156] L. Neumann, A. Neumannn, and L. Szirmay-Kalos, “Compact metallic reflectance models,” in *Computer Graphics Forum*, vol. 18, no. 3. Wiley Online Library, 1999, pp. 161–172.

[157] C. Kelemen and L. Szirmay-Kalos, “A microfacet based coupled specular-matte brdf model with importance sampling.” in *Eurographics (short presentations)*, 2001.

[158] B. Karis and E. Games, “Real shading in unreal engine 4,” *Proc. Physically Based Shading Theory Practice*, vol. 4, no. 3, p. 1, 2013.

[159] R. Ramamoorthi, “Importance sampling,” 2020, lecture Notes, CSE 168, University of California San Diego. [Online]. Available: https://cseweb.ucsd.edu/~viscomp/classes/cse168/sp20/lectures/168-lecture9.pdf

[160] B. Burley and W. D. A. Studios, “Physically-based shading at disney,” in *Acm Siggraph*, vol. 2012. vol. 2012, 2012, pp. 1–7.

[161] S. H. Peter Shirley, Trevor David Black, “Ray tracing: The rest of your life,” July 2024, `https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html`. [Online]. Available: https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html

[162] Cornell University Program of Computer Graphics , “History of the cornell box,” accessed on July 20, 2024. [Online]. Available: https://www.graphics.cornell.edu/online/box/history.html

[163] R. Butler, “Stylized low poly sci-fi buildings,” 2018, available under the Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0). Changes were made to the original model. [Online]. Available: https://sketchfab.com/3d-models/stylized-low-poly-sci-fi-buildings-701381fef32444e79ad804315e563049

[164] C. Kulla and M. Fajardo, “Importance sampling techniques for path tracing in participating media,” in *Computer graphics forum*, vol. 31, no. 4. Wiley Online Library, 2012, pp. 1519–1528.

[165] W.-C. Lin, T.-S. Huang, T.-C. Ho, Y.-T. Chen, and J.-H. Chuang, “Interactive lighting design with hierarchical light representation,” in *Computer Graphics Forum*, vol. 32, no. 4. Wiley Online Library, 2013, pp. 133–142.

[166] P. Grittmann, I. Georgiev, P. Slusallek, and J. Křivánek, “Variance-aware multiple importance sampling,” *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–9, 2019.

[167] P. Moreau, M. Pharr, and P. Clarberg, “Dynamic many-light sampling for real-time ray tracing.” in *High Performance Graphics (Short Papers)*, 2019, pp. 21–26.

[168] A. Marrs, J. Spjut, H. Gruen, R. Sathe, and M. McGuire, “Adaptive temporal antialiasing,” in *Proceedings of the Conference on High-Performance Graphics*, 2018, pp. 1–4.

Appendices

A Main GitHub Repository

https://github.com/Mateo-Coello/Real-Time-Ray-Tracing-in-OpenGL

B Ray Traced Sphere

https://github.com/Mateo-Coello/Real-Time-Ray-Tracing-in-OpenGL/tree/main/examples/rt_sphere

C Ray Traced Triangle

https://github.com/Mateo-Coello/Real-Time-Ray-Tracing-in-OpenGL/tree/main/examples/rt_triangle

D BVH Construction

https://github.com/Mateo-Coello/Real-Time-Ray-Tracing-in-OpenGL/blob/main/src/scene.c

E Distance-Based Traversal

Algorithm 1 DistanceBased Traversal

```
function TRAVERSE(BVH, $ray$)
    $root \leftarrow$ BVH[0] contains all scene primitives
    $hit\_info \leftarrow none$ contains info regarding primitive hit
    $stack \leftarrow [root]$ stack starts with BVH root
    $t\_Max \leftarrow \infty$
    while $stack$ is not empty do
        pop $node$ of $stack$
        $t \leftarrow$ intersect $ray$ with $node$
        if $t \leftarrow nearest\_hit$ then
            if $node$ is not leaf then
                $t_1 \leftarrow$ intersect $ray$ with $left\_child$
                $t_2 \leftarrow$ intersect $ray$ with $right\_child$
                if $t_1 > t_2$ then
                    push $left\_child$ into stack
                    push $right\_child$ into stack
                else
                    push $right\_child$ into stack
                    push $left\_child$ into stack
                end if
            else
                for each $primitive$ in leaf do
                    $t, primitive\_info \leftarrow$ intersect $ray$ with $primitive$
                    if $t < nearest\_t$ then
                        $hit\_info \leftarrow primitive\_info$
                    end if
                end for
            end if
        end if
    end while
    return $hit\_info$
end function
```

F Ray-Direction Sign-Based Traversal

Algorithm 2 Ray Direction Sign-Based Traversal

```
function TRAVERSE(BVH, $ray$)
    $root$ ← BVH[0] contains all scene primitives
    $hit\_info$ ← $none$ contains info regarding primitive hit
    $stack$ ← [$root$] stack starts with BVH root
    $dir\_is\_neg \leftarrow [d_x < 0, d_y < 0, d_z > 0]$
    $t\_Max \leftarrow \infty$
    while $stack$ is not empty do
        pop $node$ of $stack$
        $t$ ← intersect $ray$ with $node$
        if $t \leftarrow nearest\_hit$ then
            if $node$ is not leaf then
                $split\_axis$ ← $node$'s partition axis
                if $dir\_is\_neg[split\_axis] > 0$ then
                    push $left\_child$ into stack
                    push $right\_child$ into stack
                else
                    push $right\_child$ into stack
                    push $left\_child$ into stack
                end if
            else
                for each $primitive$ in leaf do
                    $t, primitive\_info$ ← intersect $ray$ with $primitive$
                    if $t < nearest\_t$ then
                        $hit\_info \leftarrow primitive\_info$
                    end if
                end for
            end if
        end if
    end while
    return $hit\_info$
end function
```

G Science Fiction Building Model - Normal Map