UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ARTHUR MARQUES MEDEIROS

# Interactive Photorealistic Scene Manipulation using Path Tracing

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Manuel Menezes de Oliveira Neto

Porto Alegre
June 2021

## ABSTRACT

Rendering pleasing photorealistic images requires both a high-quality renderer and well-crafted scenes. While rendering algorithms and systems have made some impressive progress over the last two decades, creating nice scenes still remains highly dependent of the artistic skills of the modeler. As a result, researchers tend to rely on a small number of existing good-looking scenes to test their algorithms. While creating new scenes from scratch is difficult for non-artists, editing existing scenes to achieve new and desired results is a task at the reach of the average graphics user. We present a system that allows users with no special artistic skills to create new scenes by modifying existing ones through a simple user interface. Enabled by modern hardware and software advancements, we render the scenes photorealistically using path tracing and provide instant feedback on the user modifications. The quality of the images generated by our system is comparable to established offline renderers, such as PBRT, while still maintaining interactive performance. Our system should stimulate the creation of new scene datasets, and allow anyone to customize existing scenes with shapes and materials according to his/her specific needs or desires. The easy customization of scenes and the high-quality renderings produced by our system may also stimulate other professionals, such as designers, scenographers, architects, decorators, etc. to make more intense use of computer generated imaging in their daily tasks.



**Keywords:** Real time rendering. physically-based rendering. ray tracing. path tracing. monte carlo ray tracing. light transport. computer graphics.

**Manipulação Interativa de Cenas Fotorealistas usando Path Tracing**

## RESUMO

**Palavras-chave:** Renderizar imagens fotorealistas agradáveis requer tanto um renderizador de alta qualidade quanto cenas bem feitas. Enquanto sistemas e algoritmos de rendering tiveram progressos impressionantes nas últimas duas décadas, a criação de cenas interessantes ainda é altamente dependente nas habilidades artísticas do modelador. Como resultado, pesquisadores tendem a usar uma porção pequena de boas cenas já existentes para testar seus algoritmos. Embora a criação de cenas do zero seja difícil para não-artistas, editar cenas existentes para conseguir novos resultados é uma tarefa ao alcance do usuário médio de computação gráfica. Nós apresentamos um sistema que permite usuários sem habilidades artísticas especiais a criar novas cenas modificando cenas existentes através de uma interface simples. Baseado em avanços recentes em hardware e software nós renderizamos as cenas fotorealisticamente usando path tracing, provendo feedback instantâneo nas modificações do usuário. A qualidade das imagens geradas pelo nosso sistema é comparável a renderizadores offline estabelecidos, como o PBRT, enquanto ainda mantendo performance interativa. Nosso sistema deve estimular a criação de novos datasets de cenas, e permitir a qualquer um a customizar cenas existentes com formas e materiais de acordo com sua necessidade ou desejo. A fácil customização de cenas e as imagens de alta qualidade produzidas pelo nosso sistema também podem estimular outros profissionais, como designers, cenógrafos, arquitetus, decoradores, etc. a fazer uso mais intenso de imagens geradas por computador em suas tarefas diárias..

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| CGI | Computer Generated Imagery |
| GPU | Graphics Processing Unit |
| PBR | Physically Based Rendering |
| GPGPU | General Purpose GPU |
| BVH | Bounding Volume Hierarchies |
| AABB | Axis Aligned Bounding Box |
| BRDF | Bidirectional Reflectance Distribution Function |
| BTDF | Bidirectional Transferance Distribution Function |
| BSDF | Bidirectional Scattering Distribution Function |
| LCG | Linear Congruential Generator |
| PDF | Probability Distribution Function |
| CDF | Cumulative Distribution Function |
| MIS | Multiple Importance Sampling |
| FOV | Field of View |
| SBT | Shading Binding Table |
| LDR | Low Dynamic Range |
| HDR | High Dynamic Range |

**CONTENTS**

# 1 INTRODUCTION

## 1.1 The Rendering Problem

Drawing realistic images has been an interest of mankind for centuries. Renaissance artists like Leonardo da Vinci were already studying optics and light in the fifteenth century, and applying that knowledge to create realistic depictions of our world.

With the advent of technology, computers entered every facet of our society. With ever increasing computational power, processors are capable of simulating the way that light interacts with our own human visual system using mathematics, and are capable of rendering results that are very close to indistinguishable from reality.

Rendering involves many overlapping disciplines, since representing and visualizing virtual scenes in a way that fools human perception takes a lot of work and decades of research. Physics equations (some of them derived before the age of computers) are utilized extensively, and a lot of research regarding our human visual system and how different objects interact with light was necessary to model realistic worlds.

Computer graphics' popularity exploded in the 1980s and 1990s, with the advent of video games and movies that had a massive global cultural impact, as well as significant technological advancements. Over the years, a lot of different techniques were utilized to render graphics on a screen, each with their own level of fidelity and characteristics. In our current world 3D graphics are everywhere, from cellphones to household appliances.

## 1.2 Uses for Realistic Rendering

Realistic rendering is a subset of computer graphics, where the objective is to render images that are as close to reality as possible. This generally involves simulating complex light transport and material scattering distributions, which causes these techniques to be computationally expensive. Figures 1.1 and 1.2 shows examples of realistic images rendered by a computer.

This kind of rendering is very popular today, due to the heavy use of computer graphics imagery (CGI) in the movies industry. While obviously being used for special effects and fantastical elements, CGI is also widely applied for more subtle uses, like shot compositing and allowing a variety of changes in post production.

Video games are another extremely popular use of realistic graphics. They tend

Figure 1.1 – A realistic looking virtual scene, modeling a modern living room with complex objects and materials.

Author: Mateusz Wielgus

to have severe limitations compared to other applications , since they need to run on real-time and have interactive framerates, but over the years a number of clever techniques have been developed to bridge that gap between offline and online rendering. Games have also been crucial to the advent of the GPU, which were originally designed for the raster pipeline but now have many uses.

In the industry, there are non-entertainment uses for this kind of rendering too. The architecture market has been revolutionized by computer tools (O'CONNOR, 2015), which allows for beautiful visualizations of house designs (CHAOS-GROUP, 2021). These tools are also a great way of showing to clients a more clear view of scenes that can be hard to imagine for non-experts. Realistic rendering is also widely used by marketing and design teams around the world to quickly prototype and develop new approaches to packaging and implementation of products, greatly benefiting from realistic material models.

The medical industry is also being revolutionized by graphics tools (GROSS, 1998), alongside with other technologies like robotics. Realistic visuals can be used to aid surgeons in a remote surgery, and can also be used to teach new medical students what to look for without access to a real-life counterpart.

## 1.3 Physically-Based Rendering

To produce realistic images, we need to simulate how light interacts with the scene in a robust way. Physically based rendering, generally based on radiometric quantities, tries to mimic physical properties of light and objects virtually. These properties can come from carefully designed approximations or from real-life measurements by means of laboratory experiments.

PBR is generally associated with ray tracing based algorithms. These techniques approximate light as rays traversing through the scene. Materials are generally modeled with a light scattering distribution. This formulation can be bent on purpose by artists to better suit their current needs and produce interesting scenes, even if the result would break the laws of physics in the real world.

Figure 1.2 – A synthetic scene, displaying a white room with complex materials.

Author: Mateusz Wielgus

Ray tracing techniques are known to come with a steep computational price. The main bulk of this cost comes from intersection tests: each ray needs to perform several of them to figure out what things are hit by it. There are several techniques designed to optimize this process, but these costs, historically, have been the main reason why ray tracing is not used in real time settings. This distance between real-time (based on rasterization) and offline techniques (based on ray tracing) only grew larger with the popularization of GPUs, which greatly accelerate the raster pipeline.

## 1.4 Hardware-Accelerated Ray Tracing

Over the years, there were several attempts to develop ray-tracing specific hardware to accelerate the overall algorithm. These attempts were generally focused on research and high-end consumers, not really aiming for widespread adoption by the public. Traditional GPUs also evolved, with the advent of general purpose computation (GPGPU). This relaxed the limitations of what could be computed in a GPU, and allowed for ray tracing algorithms to be parallelized and run on this hardware.

The first company to have ray-tracing specific hardware on consumer cards was NVIDIA, with its RTX series of cards (NVIDIA, 2018). Ray-tracing dedicated hardware was now accessible to the general consumer. This architecture introduced the RT core, which accelerated intersections calculations in hardware. This allowed for ray tracing operations to be used concurrently with the raster pipeline, and opened the possibilities for a lot of new techniques to be utilized by games and other applications.

Since the launch of RTX cards, hardware ray tracing has been widely accepted by the industry. AMD has its own ray-tracing technology on its new series of cards. The current generation of video game consoles, the Playstation 5 and the Xbox Series X, also have ray tracing acceleration built in on their architectures. Ray-tracing based techniques are now widely utilized in high-end games, allowing for high quality effects like shadows and global illumination.

## 1.5 Scene Creation

Creating photorealistic scenes is an arduous task that requires artistic talent and skill. It is also a laborous task: these scenes need to have highly detailed geometry with appealing materials, textures, lighting and camera work. To produce a high-end scene, an artist needs to not only train their sensibilities, but also master sophisticated software tools that can deal with all of these complexity.

This difficulty limits the number of high-quality scenes available, and researchers tend to rely on a small number of existing scenes to prototype and test their systems and algorithms. Thus, it is common to see the same scenes re-used across many different domains. These scenes might be chosen out of convenience, even if they are not the best choice for testing and demonstrating new rendering features.

While creating high-quality scenes from scratch is difficult for non-artists, editing

existing ones to obtain new and desired results and effects is a task at the reach of the average graphics user. This is a key observation behind our work. Unfortunately, two aspects still inhibits the editing of available scenes. First, good modeling software tend to have a highly-steep learning curve (and in many cases, are expensive). Second, the high-quality scenes containing features used by photorealistic renderers are available in the renderers' proprietary formats, making it difficult to import these scenes in a modeler for further editing. Thus, a system capable of importing and allowing for easy editing such scenes would be a high-valuable tool for the graphics community.

Intending to fill this gap, we present a system that allows users with no special artistic skills to create new scenes (from scratch or) by modifying existing ones through a simple user interface. Our system was designed to satisfy the following requirements and goals:

- **Import scenes** from existing physically-based renderers, providing initial high-quality scenes to be edited. The system should also be able to export the modified scenes in these formats;
- **Ease of use**, based on an intuitive user interface, primarily based on pointing (with a mouse), and drag-and-drop operations, for fast learning. These operations should be applied directly on the system's viewport (despite being applied in 3D), and should support object manipulation (e.g., translation, rotation, copy, deletion, addition), material changes, texturing modifications, etc.;
- **Support high-quality global illumination rendering**, allowing the user to know exactly how the scene should look like in the definite rendering;
- **Focus on interactivity**, providing instant feedback even for complex scene modifications;
- **Easy extension**, allowing for easy addition of new rendering components, such as new new materials (e.g., BRDF shaders), and high-level modeling operations;

In order to satisfy these design goals, we leverage modern hardware and software advancements. To provide high-quality photorealistic images at interactive rates we implemented a path tracer (KAJIYA, 1986) based on NVIDIA Optix (PARKER et al., 2010) that uses its built-in denoiser. The use of the denoiser significantly improves rendering quality with a small number of samples, thus producing high-quality renderings at interactive rates. This allows us to create a simple and natural interface and provide instant user feedback. Our path tracer is modular and can be easily extended to support new

Figure 1.3 – Example of scene created using our system. (a) Original scene (*Country Kitchen*). (b) Modified version of the original scene showcasing a bigger table with additional chairs, plates and cups. In addition, the kitchen island was removed and some wall materials were changed.

(a) Original scene

(b) Modified scene created with our system

Source: Our system.

materials and other rendering features. Currently, our system prototype imports scenes in the PBRT v3 (PHARR; JAKOB; HUMPHREYS, 2016) format.

Figure 1.3 illustrates the use of our system to obtain a new scene shown in (b) by modifying the original PBRT *Country Kitchen* one in (a). For this example, the kitchen island was removed; the table was enlarged; chairs, plates and cups were cloned and rearranged, and walls were painted with different colors. Although the two scenes look considerably different, these editing operations required no artistic skills or knowledge of specific modeling software. Nevertheless, the aesthetic quality of the resulting scene is comparable to the original one.

The quality of the images generated by our system is comparable to established offline renderers, such as PBRT, while still maintaining interactive performance. Our system should stimulate the creation of new scene datasets, and allow anyone to customize

existing scenes with shapes and materials according to his/her specific needs or desires.

The **contributions** of this work include a system that allows users to create high-quality scenes by modifying existing ones through a simple user interface. Our system requires no artistic skills and can produce studio-quality scenes in just a few minutes; Our system supports extensive scene manipulation, allowing the user to interact and manipulate several properties like objects, materials, and others with instant feedback.

## 1.6 Thesis Structure

This thesis will detail from the ground up all the concepts necessary for understanding the system, and how they were implemented in the code. Chapter 2 discusses related work in the area, comparing our solution to existing light transport algorithms and our implementation with established renderers. Chapter 3 will focus on the mathematical concepts that are necessary for understanding the path tracing algorithm. Chapter 4 will detail the system design, discussing the decisions that were made and their consequences on the solution. Chapter 5 will show results and comparisons that are useful to understand the capabilities and limitations of our work. Chapter 6 presents the conclusion and discussion for extensions and future work.

# 2 RELATED WORK

This chapter discusses previous work in the area of realistic rendering and scene creation.

## 2.1 Light Transport

Light transport was revolutionized with the introduction of ray tracing (WHITTED, 1980), which allowed for complex scenes and materials to be computed with relative simplicity (but high computational costs). This work spawned a new family of rendering algorithms based on ray intersections.

Path tracing was later formalized (KAJIYA, 1986), introducing important concepts like the light transport (or rendering) equation. Veach's PhD thesis (VEACH, 1997) presented a new framework to the equation, alongside many other important contributions that we use to this day.

There are several expansions to traditional path tracing that were also developed, and are extremely popular. Bidirectional path tracing (LAFORTUNE; WILLEMS, 1993) (VEACH; GUIBAS, 1995) creates paths from the viewer and also from the light sources, attempting connections and finding better contributions in scenes with hard-to-reach light. Metropolis Light Transport (VEACH, 1997) involves local domain space explorations to improve sample quality, allowing for faster convergence.

## 2.2 Other Renderers

Our work is based on other existing renderers that have helped shape the field of computer graphics.

### 2.2.1 PBRT

PBRT (PHARR; HUMPHREYS, 2015) is one of the most influential renderers in the history of computer graphics. It is also the main inspiration for our work, serving as a great source of consultation and comparisons. The PBRT architecture is described in the book Physically Based Rendering (PHARR; JAKOB; HUMPHREYS, 2016), which also

served as a great source of reference for this thesis.

The last official PBRT version is PBRT-v3, the third big iteration of the original system. It is a CPU renderer, and its main purpose is to educate: therefore, it is not the most performant renderer in the market.

At the time of this writing, a v4 version of PBRT is announced and in beta (PHARR; HUMPHREYS, 2021). It features GPU acceleration using NVIDIA OptiX (much like our own work), providing a substantial speed improvement on rendering. We started this work before the v4 version was even announced, so our decisions were not influenced by this upcoming version.

None of these PBRT versions allow for user interaction or live scene modification. Without external tools, modifying a scene in PBRT requires a manual change in the text-based scene file. After that, the user must start the rendering all over again - a process that can take hours.

### 2.2.2 Mitsuba 2

Mitsuba 2 (NIMIER-DAVID et al., 2019) is a modern renderer, designed to run on both CPU and GPU. It accelerates its vector computation using the Enoki library, that also provides autodifferentiation capabilities to its application. Much like its predecessor, Mitsuba 2 is a research focused rendered, designed for fast development of new techniques and algorithms.

Mitsuba 2 also does not allow for any meaningful user interaction, even though it is much faster than PBRT. It also does not have easy ways of modifying an existing scene.

## 2.3 Modeling Software

Blender (Blender Online Community, 2018) has a complex 3d toolkit, and allows for modeling, texturing, scene creation, and much more. It also contains several different renderers, that range from full path tracers to more simple rasterization based approaches. Cycles is an interative CPU path tracer that allows for relatively fast updates to the user, and Cycles X (LOMMEL, 2021) is its GPU accelerated counterpart. While its many tools centralize anything that an artist may need, it also greatly steepens the learning curve of the software, making it hard for beginner users to pick it up quickly.

SketchUp (TRIMBLE, 2021) is an architectural 3D application focused on modeling. It can also import models from an online repository directly, and photorealistically render the resulting scene. The rendering process is slow, and the renderer used when working on scenes is very basic (to facilitate geometry modeling operations).

V-Ray (CHAOS-GROUP, 2021) is a renderer plugin that is available for a variety of software products. It contains a GPU accelerated path tracer, making it able to do real-time performance. It is, however, only a renderer, and depends on its overlying software to modify the scene.

## 2.4 Other Work

Hagemann and Oliveira presented a system designed to convert scenes between popular renderers (HAGEMANN; OLIVEIRA, 2018). This system works by converting whole scenes from one format to the other, but does not allow for user interaction or for modifications in existing scenes. It also only deals with the scene formats themselves, and does not render scenes in any way.

Bitterli, in his public scene repository (BITTERLI, 2016), details the laborous process of creating a PBR scene from scratch. The process took several days per scene, and involved manually setting PBR materials, lighting and camera for each one.

# 3 GLOBAL ILLUMINATION BACKGROUND

In this chapter, we review some key theoretical concepts that are of utmost importance to the thesis. Most of these topics are deep and complex, so this chapter only covers the basics and focus on the relevant areas. Implementation details are not present here, but are detailed in the next chapter. Appendices A and B present more basic concepts, introducing algebraic operations and a very brief introduction to radiometric units. Both appendices and this chapter are heavily based on the book "Physically Based Rendering, From Theory to Implementation" (PHARR; JAKOB; HUMPHREYS, 2016).

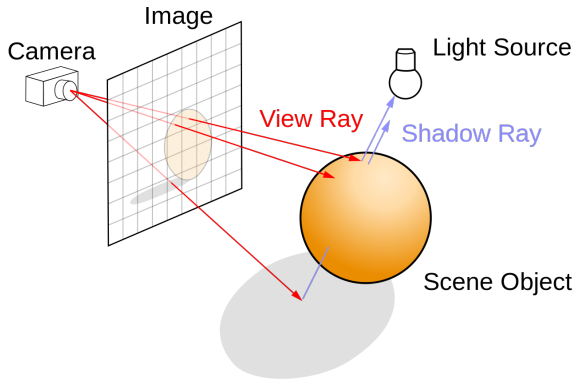
## 3.1 Ray Tracing

The field of computer graphics was revolutionized with the introduction of ray tracing (WHITTED, 1980). It is an algorithm developed to render complex images of globally illuminated scenes. The original paper presented by Whitted already contains most of the fundamentals that we still use today, but several extensions to the traditional ray tracing algorithm were researched and developed over the years.

The main idea is to approximate light traveling in the scene as a geometrical ray. A valid light path that goes from the light source to the camera (or eye, or observer) can be inverted to produce another valid light pact that goes the opposite way. Using this intuition, the algorithm starts by tracing rays for each pixel of the image starting from the camera, and adds each contribution that it finds from light sources to the final pixel value. At each surface hit, the algorithm can trace subsequent rays from the intersection point, creating a light path that can simulate global illumination. The final color of the pixel will be affected by several things like camera model, participating media, material of objects hit, etc. A diagram showing a representation of the ray tracing algorithm is shown in Figure 3.1.

It is worth noting that the nomenclature between ray tracing techniques is a point of contention in the graphics community. Even though ray tracing can be used to refer to a traditional Whitted ray tracer, it can also be used to refer to a stochastic (distributed) ray tracer (COOK; PORTER; CARPENTER, 1984), a path tracer (KAJIYA, 1986) or even more complicated techniques that use the ray casting operation. Here, "ray tracing" and the techniques explained ahead will refer to stochastic ray tracing, a simple extension to the original ray tracer that utilized random numbers to sample different parts of the

Figure 3.1 – A diagram explaining the traditional ray tracing setup.




Source: extracted from Wikipedia

domain.

### 3.1.1 Camera Model

The initial rays that leave the camera and its contributions to each pixel's final color will be determined by the camera model. There are several different abstractions that can be utilized here, and each abstraction allows for different image effects.

There are advantages from the algorithm's structure that we can exploit. First, we do not need to simulate the internal part of a camera: shooting rays from the lens to the focus plane is enough. Simulating more complex parts of a camera (like an imperfect sensor of a compound lens) could lead to interesting results, but luckily, most of the resulting effects can be simulated by more straight-forward means.

#### *3.1.1.1 Pinhole Camera*

The pinhole camera has an infinitely small lens, which to our purposes means that every single ray will leave the same point. This creates an image that is always in focus, and it is the simplest, fastest camera to utilize in a ray tracing algorithm.

#### *3.1.1.2 Aperture*

The pinhole virtual camera can easily be extended to produce effects such as defocus blur (commonly referred to as depth of field). Objects that are not in the focus plane will not be focused in the image, and the amount of blur will be determined by the size of what is called the circle of confusion. This size can be influenced by changing the

aperture of the lens.

To simulate a lens with a non-zero radius, we just need to sample a disk with the same radius centered at the camera's center and use this value as the current's ray origin. The sampling can be done with a simple rejection based approach. This calculation is done per ray, meaning that every time a new ray is cast a new value will possibly be chosen. This can introduce noise, so an acceptable number of samples per pixel must be chosen.

#### *3.1.1.3 Ray Direction*

Since we're simulating a traditional lens, all rays need to converge in the virtual film plane. Now, we also must divide the image plane in several areas - one for each pixel in the final image.

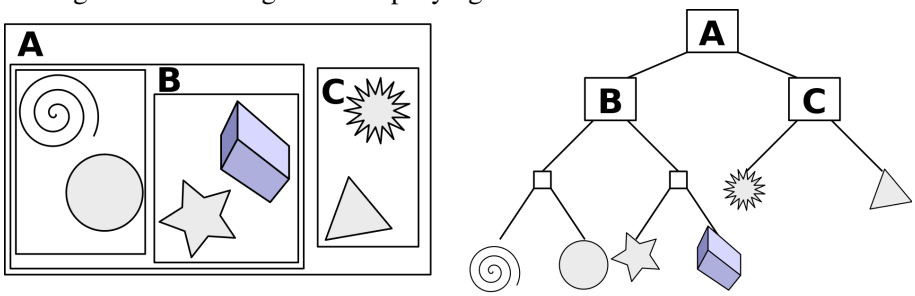
We could just draw a ray to the middle of this area, and be content with only one ray per pixel. This leads to severe aliasing, however, since what that single ray hit will be the only information that we will have for the entire pixel. To mitigate that, we can sample the whole area of the pixel randomly. Each ray will go through a different point in that pixel's portion of the film plane, and when all samples are added together, a closer approximation of the original signal will be obtained.

### 3.1.2 Acceleration Structures

As we discussed before, ray intersections are key to the ray tracing algorithm: they will define which surface will color the final pixel of the image. There is a major problem, however: since scenes can contain millions of primitives, intersection would need to be tested for each primitive every ray. This is unfeasible even in scenes with a low number of primitives and samples.

To mitigate that issue, we use acceleration structures. They are intermediary primitives that can contain multiple pieces of the scene, restructuring it in a tree-like structure. That way, if the intersection fails for the overlying structure, the algorithm can discard everything below, saving a lot of computation.

Figure 3.2 – A diagram exemplifying the tree-like structure of a BVH.




Source: extracted from Wikipedia

*3.1.2.1 Bounding Volume Hierarchies*

Bounding Volume Hierarchies (BVH) are probably the most common type of acceleration structure. The technique consists in creating volumes (usually a bounding box) that contains parts of the scene, and possibly even other bounding boxes. Figure 3.2 illustrates the tree-like structure of a BVH, representing objects as shapes.

Optimal BVH construction is a research subject on its own: there are several common strategies (each with its own drawbacks), ranging from simple to complex. A badly built BVH can still help with intersections, but an optimal BVH is crucial to the performance of a ray tracer, since intersection calculations will generally be the most costly part of the whole algorithm.

## 3.2 Path Tracing

Path tracing (KAJIYA, 1986) is a major overhaul to the ray tracing algorithm that defined light transport in a more robust way and allowed for complex interactions to be modeled and rendered. The algorithm was computationally expensive but produced incredible results, paving the way for a new era of physically-based rendering. Several additions to the original path tracing definition were later developed, improving even more its capabilities.

Path tracing extends ray tracing by allowing nondelta distributions to be sampled. Every surface interaction can generate rays based on its distribution, making area lights and complex light scattering distributions possible to simulate. To achieve this, the rendering equation is defined: a seminal contribution to graphics and the basis of every subsequent physically based technique.

### 3.2.1 Rendering Equation

To compute complex distributions, a path tracer solves the rendering (or light transport integral) equation (KAJIYA, 1986): a representation that presents the rendering problem as a recursive integral over several domains. This makes the equation difficult to solve: not only is the integrand discontinuous, but it is also infinitely-dimensional for each new vertex in the path.

The rendering equation has many forms. In this one, it is designed to represent the spectral radiance that is reflected from a point $\boldsymbol{p}$ to a direction $\omega_o$. It can be written as:

$$L_o(\boldsymbol{p}, \omega_o) = L_e(\boldsymbol{p}, \omega_o) + \int_\Omega f_s(\boldsymbol{p}, \omega_i, \omega_o) L_i(\boldsymbol{p}, \omega_i) |\cos \theta_i| \, d\omega_i, \tag{3.1}$$

where $L_o$ and $L_i$ are outgoing and incoming radiance, $\omega_i$ is an incoming direction and $L_e$ is the radiance emitted by the surface. Also, $\Omega$ represents the hemisphere (centered at the normal) of the surface, and $\theta_i$ represents the angle between $\omega_i$ and the surface's normal at $p$.

There is one term that deserves a special introduction: $f_s$ is also called the bidirectional scattering distribution function (BSDF), which is the combined distributions of reflectance and transmission of light in the surface. All of these terms will be explained and expanded upon later.

To sample this equation, we simply cast rays in the scene. They can be calculated by sampling each dimension in the domain with random distributions. For every intersection, more rays can be cast, by sampling each domain in the rendering equation once again. This series of points in space that the original ray reaches by subsequent rays form what is called a path. With $n$ being the number of vertices in the path:

$$\bar{p}_n = p_0, p_1, ..., p_{n-1}. \tag{3.2}$$

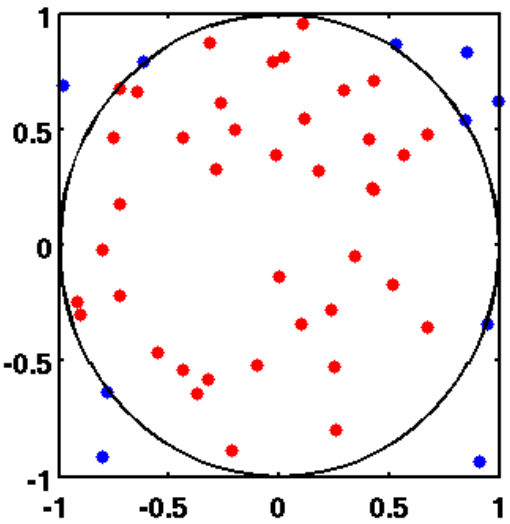
This definition has an useful implication: if every domain can be sampled by one or more random variables $\epsilon$ and the vertices in a path are only determined by these values, a path can also be represented by this series of values alone.

### 3.2.2 Monte Carlo Integration

The rendering equation is tricky: light contributions have complex occlusion distributions, making the function discontinuous. Light paths can have an unbounded number of vertex, making its domain infinitely-dimensional. All of these factors combine to make the equation impossible to solve analytically. We can, however, sample it easily by tracing rays in the scene. This allows the use of a very powerful technique: Monte Carlo integration.

The Monte Carlo method is numerical (not analytical) and uses random samples to approximate the solution of a difficult integral. As the number of samples increases, the approximated result will also get closer to the real solution. It has several interesting properties, including a convergence rate that is not dependent on the dimensionality of the integrand. Figure 3.3 showcases one of the most famous (and simple) problems that can be solved by this method.

Figure 3.3 – A classic example of Monte Carlo integration: computing the area of a circle via random sampling of points. All that is needed is a way to check if each sample is within the circle and know the square's area.




Source: extracted from Wikipedia

Differently than Las Vegas algorithms, that have random elements but always return the same result, Monte Carlo algorithms are not guaranteed to return the same thing: every independent execution of the method can return different values. On average, they will all converge to the integral's true mathematical result.

To use the Monte Carlo method, all that is needed is a way to randomly sample the function domain that we're trying to integrate. In rendering, that is done by sampling each

dimension of the path: things like the original position of the ray in a pixel's boundary; a direction in a unit sphere to reflect a ray when an object is intersected; a point in an area light source to be sampled by an object.

Undersampling with the Monte Carlo method, in rendering, will result in noise: pixels in the image with lower or higher value than expected. There are several ways of mitigating this noise (and some of them will be covered later). Increasing the number of samples is an option, but it is extremely expensive: the convergence of Monte Carlo algorithms is $O(n^{-1/2})$. This means that to cut the error in half, we need to use four more samples.

#### *3.2.2.1 Random Variables*

To create our samples, we will need to use several random variables $\xi$, which have continuous values where $\xi \in [0, 1)$. Every $\xi$ is an uniform variable, meaning that every value has the same probability of being chosen. Since for rendering we need to randomly sample many dimensions with different domains and every $\xi$ has the same range of values, we need to apply many different transformations to map the variables to their desired results.

With computers, deterministic by nature, generating true random values is actually an extremely complicated task. Even though those methods exist, generally consisting on measuring natural phenomena like atmospheric noise or ocean waves, random numbers are generally approximated with pseudo-random number generation. These are sequences that have an approximated uniform distribution but can have noticeable patterns.

For our purposes, this method of generating samples is more than enough and even has some advantages: using pseudo-random sequences guarantees that each run of the algorithm is deterministic and always results on the same value (if, of course, the same initial random seed is provided). The algorithm used in the implementation (that will be detailed later) is a Linear Congruential Generator (LCG), one of the simpler pseudo-random number generators there is. It uses a minimal memory footprint and is extremely fast to compute the next number in the series.

#### *3.2.2.2 Probability Functions*

In a random uniform variable, each value $x$ in the domain has a chance to be chosen. This probability is called $p$, and is defined by the probability density function

(PDF) $p(x)$. An uniform random variable like $\xi$ has a constant $p$ over its entire domain, which means that every value has the same probability of being chosen. The integral of $p(x)$ must also integrate to $1$ when taken over the whole domain.

It is also useful to know the probability of a random variable being less or equal than a value $x$. For that, we utilize the cumulative distribution function (CDF) $P(x)$. For example, when rolling a 20-sided dice (commonly known as D20), $P(5) = 1/4$ because there are 5 values out of 20 that is less or equal than 5.

#### *3.2.2.3 Expected Value*

When paired with a probability distribution $p(x)$, the expected value of a function $f(x)$ over a domain is:

$$E_p[f(x)] = \int_D f(x)p(x)dx. \tag{3.3}$$

#### *3.2.2.4 Variance*

To measure the error in our approximations, we utilize variance. The variance of a function is defined as:

$$V[f(x)] = E[(f(x) - E[f(x)])^2]. \tag{3.4}$$

There are several techniques designed to improve Monte Carlo convergence. Variance and expected value are both extremely useful tools in measuring how good each of them are, and allow for a baseline to compare different approaches.

#### *3.2.2.5 Bias*

When the expected value of our approximation is the same as the function it is trying to replicate we say that our method is unbiased. An unbiased estimator will always eventually converge to the right result. We can, however, have a biased estimator, where the bias $\beta$ is defined as:

$$\beta = E[F] - \int f(x)dx. \tag{3.5}$$

Bias is generally avoided in rendering: we want to represent the light interactions

in the most "pure" way possible. There are, however, some techniques that are considered worth this downside, since they can drastically reduce convergence times.

#### *3.2.2.6 Monte Carlo Estimator*

The Monte Carlo estimator is defined, for an integral $\int_a^b f(x)dx$ and random variables $X_i \in [a, b]$, as:

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i), \tag{3.6}$$

where $N$ is the number of samples. This is only valid for a constant probability function, so for every $x$, $p(x) = 1/(b-a)$. This is limiting, but a simple modification allows for the more general:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}. \tag{3.7}$$

This formulation has a clear limitation: $p(x)$ must be higher than $0$ for every $x$ in the domain.

The rate of convergence can also now be calculated. This estimator converges at the rate of O($\sqrt{N}$). In lower dimensions this is not the optimal convergence rate, but it is great for our possibly infinite dimensional integral. This convergence rate also shows why simply throwing more samples may not be the best idea to improve our approximation: in our case, more samples means casting more rays, which has a significant computational cost. Luckily, there are lots of mathematical tools to improve convergence.

#### *3.2.2.7 Russian Roulette*

Apart from variance, running time is also an important parameter to keep in mind when comparing two different estimators. Russian roulette is a technique designed to trade off variance for a lower running time.

The key concept in russian roulette is that each sample has a probability $q$ to not be evaluated and added to the sum. Instead, a constant $c$ is used (often $0$). The probability can be defined in any way, but the way this value is defined is key to the success of this method.

$$\begin{cases} \frac{F-qc}{1-q} & \xi > q \\ c & \text{otherwise.} \end{cases} \tag{3.8}$$

Samples that are still evaluated have the probability of $1-q$, and must be weighted by the term $1-(1-q)$ to compensate for the lost samples.

Russian roulette will never reduce variance. In fact, a bad implementation will significantly increase it, even though the approximation would still eventually converge to the right result, since the expected value of the new estimator is the same as the old one. The advantage of using russian roulette is that samples that add little to the overall result may not even be computed, saving computational time and improving the chances that the computed samples are actually good.

#### *3.2.2.8 Splitting*

Splitting consists on separating the number of samples per dimension of the integrand to achieve better results. This can be useful when certain parts of the domain do not contribute more to the final result with a higher number of samples.

A classic example is in rendering itself: splitting is traditionally used to sample more than one light sample per surface interaction, or even for sampling more than one distribution. Without this key insight, to draw more light samples we would need to compute another whole light path.

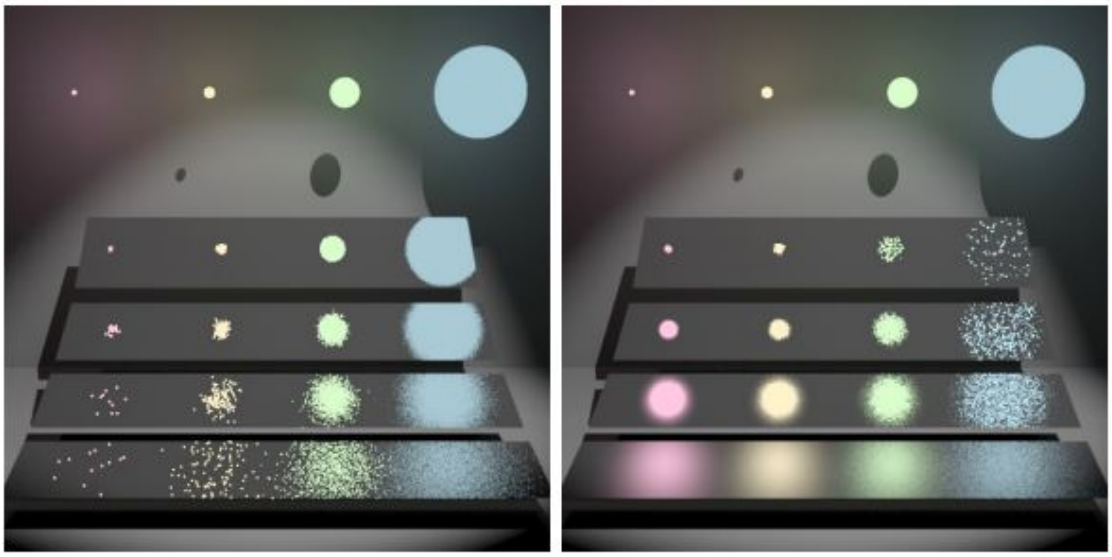
#### *3.2.2.9 Importance Sampling*

Importance Sampling is an essential Monte Carlo technique. It is based on the notion that an estimation will converge faster if $p(x)$ is proportional to $f(x)$. What that means, intuitively, is that parts of the domain a with higher contribution should have a higher probability of being sampled than parts with a lower value. The impact caused by different sampling strategies can be seen in Figure 3.4.

Ideally, we would have a $p(x)$ so that $p(x) = cf(x)$, with a normalizing constant. Of course, this is not possible, since the value of $f(x)$ is what we're looking for in the first place. Luckily, even general approximations can result in a good reduction of variance.

In rendering, our function $f(x)$ is a product of several other functions, as we've seen on the rendering equation. Even though an approximation of some of these terms can be easily found, an accurate approximation of the final product is usually not. These single-term approximations can still be used, and in some cases result in satisfactory improvements. A commonly used, extremely versatile $p(x)$ is a cosine-weighted hemisphere distribution, since the geometric term $|\cos\theta_i|$ is present in the rendering equation.

Figure 3.4 – A comparison between two sampling strategies: the first figure samples based on a BSDF-shaped distribution, and the second figure samples the light sources directly. Each plate has different surface roughness.



Source: extracted from (VEACH, 1997)

*3.2.2.10 Multiple Importance Sampling*

The rendering equation's integrand is a product of many different terms. Each of these terms can be easily approximated, but there's no easy way to multiply all of these probability functions to create an all encompassing $p(x)$. Using only one of these distributions may not be enough in many cases. With a near-specular BSDF, for example, there is only a small part of the hemisphere where any light is actually reflected from, so using a distribution not based on the $f_s(\boldsymbol{p}, \omega_i, \omega_o)$ term will result in bad samples. Likewise, in the case of a small light, drawing samples directly from the light distribution is optimal, otherwise very few rays will actually find the light in question.

Multiple Importance Sampling (VEACH, 1997) addresses this problem, by allowing for the combination of different samples from different distributions. These samples are weighted by a term designed to reduce these large variance spikes. The impact that MIS can have in an image can be seen in Figure 3.5.

Trying to approximate $\int f(x)g(x)dx$, with probabilities $p_f$ and $p_g$, the MIS Monte Carlo estimator is:

$$\frac{1}{n_f}\sum_{i=1}^{n_f}\frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)} + \frac{1}{n_g}\sum_{j=1}^{n_g}\frac{f(Y_j)g(Y_j)w_g(Y_j)}{p_g(Y_j)}. \tag{3.9}$$

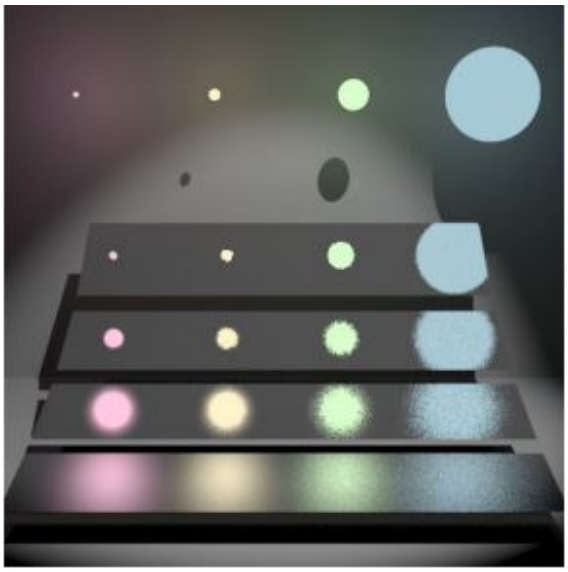
The term $w_f$ and $w_g$ are the weight terms for each sample, and $n_f$ and $n_g$ are the number of samples of each distribution. There are some common weighting functions

used, starting with the balance heuristic:

$$w_s(x) = \frac{n_s p_s(x)}{\sum_i n_i p_i(x)}. \tag{3.10}$$

If a sample $X$ is taken from $p_f$ and $p_f(X)$ is small, $f(X)$ will probably be small too (considering that $p_f$ is an approximation of $f_x$). But if $g(X)$ ends up having a big contribution, $f(X)g(X)$ will be divided by a very small $p_f$, drastically enlarging an already big value and resulting in big variance. The balance heuristic counteracts this, by adding $n_g p_g(X)$ in the denominator.

Figure 3.5 – The same scene from figure 3.4, but now using both samples weighted by the balance heuristic.



Source: extracted from (VEACH, 1997)

A common modification of this weighting function is called the power heuristic. With $\beta$ being an arbitrary exponent:

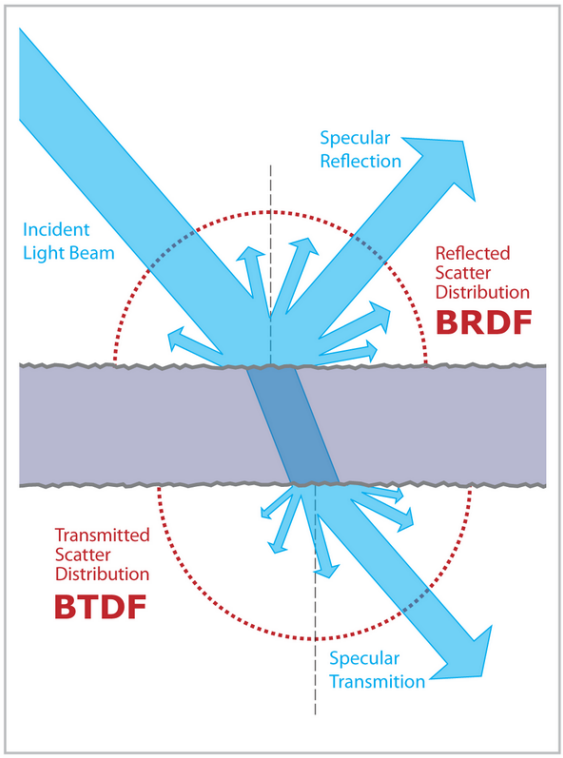
$$w_s(x) = \frac{(n_s p_s(x))^\beta}{\sum_i (n_i p_i(x))^\beta}. \tag{3.11}$$

A common value for $\beta$ is 2, but other values can also be used.

### 3.2.3 Distribution Functions

In our rendering equation formulation, the term $f_s$ represents a bidirectional scattering distribution: a function that, given two directions, determines how much light is transmitted from one to the next. This is the term that will determine all of the character-

Figure 3.6 – A figure illustrating a BSDF with both reflectance (BRDF) and transmittance (BTDF).




Source: extracted from Wikipedia

istics of the material of the intersected object. For example: if a ray intersects a car, its BSDF will determine if the car's paint is matte or glossy, or if the car is made of metal or glass.

The BSDF is generally also represented as a combination of two other functions: a bidirectional reflectance distribution function (BRDF) and a bidirectional transmitance distribution function (BTDF). When a light ray hits a surface, reflection will make the ray "bounce" and leave the surface through the same hemisphere it entered, and transmission will make the ray go through the surface and leave the surface from the other side. Figure 3.6 illustrates the difference between BRDFs and BTDFs.

BSDFs can come from a lot of different places. They can be simple formulas that approximate real materials with acceptable precision; they can also be data-driven, via measurements made in labs with real world objects. Most BSDFs also have independent terms that can be tweaked to create different looks, which are extremely useful for artists trying to make an object look a certain way.

Figure 3.7 – A model (the white dragon) rendered with a perfectly Lambertian distribution. Original "dragon" model by Christian Schüller.

Source: our system.

#### *3.2.3.1 Lambertian*

The Lambertian reflection function is a theoretical distribution that is commonly used in computer graphics. It is characterized by scattering light equally in all directions. It is considered theoretical because a perfectly Lambertian object does not really exist, even though very close approximations are possible. The distribution is named after Johann Heinrich Lambert, for his work on photometry.

A surface that has this distribution will reflect the same color (that is, amount of light) independent of the observer's direction. It can be said, then, that this surface is ideally diffuse, and that it presents perfectly diffuse reflection. Figure 3.7 shows an object rendered with such a distribution.

When dealing with a Lambertian surface, we use a cosine-weighted hemisphere function as our importance sampling distribution. The reason for that is simple: since light is scattered equally in every direction, for any $x$ and $x'$ in the hemisphere: $p(x) = p(x')$.

#### *3.2.3.2 Specular*

A perfect mirror can be considered the opposite of a Lambertian surface: instead of reflecting incoming light to all directions, it will only reflect to a single direction. This means that the color of a specular object will change depending on the observer's position.

In a specular distribution, the reflected ray will always be the same for an incoming direction, and it can be found by "mirroring" the original ray around the normal of

Figure 3.8 – A model rendered with the distribution of perfectly specular reflection. Original "dragon" model by Christian Schüller.

Source: our system.

the surface. This means that, for the direction vector of an incoming ray $v_i$ and for the direction vector of the reflected ray $v_r$:

$$(\boldsymbol{v_i} \cdot \boldsymbol{n}) = (-\boldsymbol{v_r} \cdot \boldsymbol{n}). \tag{3.12}$$

A specular distribution needs special care during sampling. Since for every possible outgoing direction there's only a single relevant incoming direction, usual multiple importance sampling techniques might be useless. In those cases, only the surface distribution should matter for sampling purposes, and the sampled direction should always be the perfect reflection.

The best example of a specular distribution is a mirror: once again, a perfectly specular surface is also not possible, but mirrors are a very close approximation. Figure 3.8 shows an object rendered with a mirror-like distribution. A specular distribution can also be used for transmission: in this case, we would have perfectly smooth glass. The perfect transmission direction is calculated by Snell's law:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t. \tag{3.13}$$

Both $\eta$ terms are called the indices of refraction, which are the ratio between the speed of light in a vacuum (represented by $c$, where $c \approx 3 * 10^5 km/s$) and the speed of

light in the medium $v$:

$$\eta = \frac{c}{v}. \tag{3.14}$$

Indices of refraction tend to vary by wavelength, causing dispersion on transmission (an effect immortalized in pop culture by the album cover of Pink Floyd's Dark Side of the Moon). A common simplification used by rendering algorithms is to ignore this property, and treat every wavelength as having the same index of refraction.

#### *3.2.3.3 Dieletrics and Conductors*

An important distinction to make is between dieletric and conductor mediums. The first does not absorb light, so all of it is either reflected or transmitted. Dieletrics do not conduct electricity. Examples of dieletrics include glass, air and water.

Conductors, obviously, conduct electricity, and are generally metals. They also absorb a significant portion of light. This causes any transmitted light that travels through the conducting media to be absorbed really quickly. For that reason, only reflection is calculated for conductor-based materials. Because of this conductors have a complex index of refraction $\underline{\eta} = \eta + ik$.

#### *3.2.3.4 Fresnel Equations*

For physically accurate materials, the proportion of light that is reflected or transmitted is described by the Fresnel equations. They can be computed by solving Maxwell's equations at smooth surfaces.

The equations are defined for polarized light, and as such, have both perpendicular and parallel terms. The value of reflectance for dieletrics can be defined as:

$$r_{\parallel} = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}, \tag{3.15}$$

$$r_{\perp} = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}. \tag{3.16}$$

We do not take into account polarized light, so we need to use the Fresnel reflectance for unpolarized light:

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2). \tag{3.17}$$

Since dieletrics either reflect or transmit light, Fresnel transmittance is defined as $1 - F_r$.

Conductors have a slightly more complicated Fresnel reflectance, since we also need to account for absorption. The Fresnel reflectance between a conductor and a dieletric medium is:

$$r_\perp = \frac{a^2 + b^2 - 2a\cos\theta + \cos^2\theta}{a^2 + b^2 + 2a\cos\theta + cos^2\theta}, \tag{3.18}$$

$$r_\parallel = r_\perp \frac{\cos^2\theta(a^2 + b^2) - 2a\cos\theta\sin^2\theta + \sin^4\theta}{\cos^2\theta(a^2 + b^2) + 2a\cos\theta\sin^2\theta + \sin^4\theta}, \tag{3.19}$$

$$a^2 + b^2 = \sqrt{(\eta^2 - k^2 - \sin^2\theta)^2 + 4\eta^2k^2}. \tag{3.20}$$

When $k = 0$ the result will be the same as the previous formulation. Since the complex equation is significantly more costly to compute, we should only use it when needed.

#### *3.2.3.5 Fresnel-Specular*

Figure 3.9 – A model rendered with the distribution of specular reflection and refraction weighted by the fresnel equations. Original "dragon" model by Christian Schüller.

Source: our system.

We now can simulate a more physically accurate dieletric material, like glass. This distribution will reflect and transmit light according to the Fresnel equations. The proportion of light reflected is $F_r$, and $1 - Fr$ is the proportion of light transmitted.

This Fresnel term will be dependent on the viewing angle. This phenomenon can even be observed in the real world: when looking straight at a water surface it will be close to transparent, allowing for what's underwater to be seen, but at grazing angles the surface will behave more and more like a mirror. An object rendered with this glass-like distribution can be seen in Figure 3.9.

#### *3.2.3.6 Microfacet Models*

A common abstraction to represent photorealistic materials are microfacet-based models. The basic intuition behind them is that any surface is composed of tiny height variances that can be approached by microfacets, and the overall material scattering properties are affected by their distribution. Diffuse objects would have more microfacet variance, and smoother ones would have less.

The nomenclature can cause some confusion. The overall geometry of the object is called macrosurface (represented by a triangle mesh, for example) as opposed to the tiny microsurface that contains the microfacets.

Microfacets are extremely tiny, so we consider that whenever the macrosurface is lit a significant amount of microsurfaces are hit as well. This means that most microfacet-based approximations are represented by statistical models that represent the overall distribution of microfacets that contribute light to the desired direction.

There are also several popular microfacet models, and some of them will be discussed here. Some of the problems that these models need to account for are the way that microfacets interfere with each other. Just like scene geometry, they can end up occluding each other in regards to the light, and the light also can reflect over more then one microsurface before leaving the surface.

#### *3.2.3.7 Microfacet Definitions*

There are important definitions that tend to be common between microfacet models. Even though the value of these properties may change, each model can be represented by their own formulas to compute these values. This creates a lot of variability on the resulting BSDF, allowing for a number of different materials to be represented by the same models.

The first important concept is $D(\omega_h)$, where $\omega_h$ is the surface normal. It represents the differential area of microfacets with the surface normal. It has some limitations if we

want the model to be physically plausible:

$$\int_{H^2(n)} D(\omega_h) \cos\theta_h d\omega_h = 1. \tag{3.21}$$

To represent the occlusion effects described at the end of the last section, we use Smith's masking-shadowing function $G_1(\omega, \omega_h)$, that returns the proportion of microfacets with normal $\omega_h$ that are visible to the direction $\omega$. Meaning that, if $A^+(\omega)$ is the distribution of forward-facing microfacets and $A^-(\omega)$ is the equivalent for backwards-facing ones, then:

$$G_1(\omega) = \frac{A^+(\omega) - A^-(\omega)}{A^+(\omega)}. \tag{3.22}$$

The $\Lambda(\omega)$ function measure the proportion of occluded microfacets to visible microfacet area:

$$\Lambda(\omega) = \frac{A^-(\omega)}{A^+(\omega) - A^-(\omega)} = \frac{A^-(\omega)}{\cos\theta}. \tag{3.23}$$

This leaves the simplified $G_1$ term to be defined as:

$$G_1(\omega) = \frac{1}{1 + \Lambda(\omega)}. \tag{3.24}$$

There are several ways to define a $\Lambda(\omega)$ function, and each model can decide their approach. One common distinction is if this function will assume no correlation between heights of nearby points, which may result in a unique $Lambda(\omega)$ for a determined $D(\omega_h)$.

Finally, we also have $G(\omega_o, \omega_i)$, which gives the fraction of microfacets visible to both directions. There are also several ways of representing this function, but a common model is:

$$G(\omega_o, \omega_i) = \frac{1}{1 + \Lambda(\omega_o) + \Lambda(\omega_i)}. \tag{3.25}$$

These concepts are common to most microfacet models, and their definition will be what gives each model its characteristics.

Figure 3.10 – A model rendered with the Oren-Nayar distribution. Original "dragon" model by Christian Schüller.

Source: our system.

### *3.2.3.8 Oren-Nayar*

The Oren-Nayar model (OREN; NAYAR, 1994) is a distribution utilized to render diffuse objects. It is more physically accurate than the more simple Lambertian model. In practice, Oren-Nayar objects will be brighter when lit from a light close to the viewing direction. Figure 3.10 shows an object rendered with an Oren-Nayar BRDF.

The model represents the microsurface as many V-shaped diffuse microfacets. Their distribution is then defined by a spherical Gaussian function, with $\sigma$ being the standard deviation of the orientation angle.

The full model is not suitable for our purposes (it does not have a a closed form solution), but there exists a suitable approximation where ($\sigma$ in radians):

$$f_r(\omega_i, \omega_o) = \frac{R}{\pi}(A + B \max(0, \cos(\phi_i - \phi_o)) \sin\alpha \tan\beta), \tag{3.26}$$

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)}, \tag{3.27}$$

$$B = \frac{0.45\sigma^2}{\sigma^2 + 0.09}, \tag{3.28}$$

$$\alpha = \max(\theta_i, \theta_o), \tag{3.29}$$

$$\beta = \min(\theta_i, \theta_o). \tag{3.30}$$

For importance sampling, the same strategy used for the Lambertian distribution can also be used for the Oren-Nayar model.

#### *3.2.3.9 Beckmann-Spizzichino*

Figure 3.11 – A model rendered with the Beckmann-Spizzichino microfacet distribution, and with the Torrance-Sparrow BRDF model. Original "dragon" model by Christian Schüller.

Source: our system.

The Beckmann-Spizzichino model (BECKMANN; SPIZZICHINO, 1963) represents each microsurface as a perfect reflector/transmitter. It is also based on a gaussian distribution of microfacets. The isotropic formulation of this model is:

$$D(\omega_h) = \frac{e^{-\tan^2 \theta_h / \alpha^2}}{\pi \alpha^2 \cos^4 \theta_h}, \tag{3.31}$$

where if $\sigma$ is the root mean square slope, $\alpha = \sqrt{2}\sigma$. The representation can also be extended to support an anisotropic distribution:

$$D(\omega_h) = \frac{e^{-\tan^2 \theta_h (\cos^2 \phi_h / \alpha_x^2 + \sin^2 \phi_h / \alpha_y^2)}}{\pi \alpha_x \alpha_y \cos^4 \theta_h}. \tag{3.32}$$

Note that this distribution will become equal as its isotropic counterpart when $a_x = a_y$. The distribution's $\Lambda(\omega)$ function assumes that there are no correlation of height

between nearby points. Its definition for the isotropic distribution is:

$$\Lambda(\omega) = \frac{1}{2}\left(\text{erf}(a) - 1 + \frac{e^{-a^2}}{a\sqrt{\pi}}\right), \tag{3.33}$$

where $a = 1/(\alpha \tan\theta)$ and erf, the error function, is:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x'^2} dx'. \tag{3.34}$$

Figure 3.11 shows an object rendered with this BRDF. For anisotropic distributions, a common technique is interpolating an $alpha$ value and using that with the isotropic $\Lambda$ function.

#### *3.2.3.10 Trowbridge-Reitz (GGX)*

Figure 3.12 – A model rendered with the Trowbridge-Reitz (GGX) microfacet distribution, and with the Torrance-Sparrow BRDF model. Original "dragon" model by Christian Schüller.

Source: our system.

The Trowbridge-Reitz model (TROWBRIDGE; REITZ, 1975) is another useful model, which can be seen in Figure 3.12. It was popularized later (WALTER et al., 2007), when it was independently derived and named 'GGX'. Its anisotropic microfacet distribution is given by:

$$D(\omega_h) = \frac{1}{\pi\alpha_x\alpha_y \cos^4\theta_h (1 + \tan^2\theta_h(\cos^2\phi_h/\alpha_x^2 + \sin^2\phi_h/\alpha_y^2)^2}. \tag{3.35}$$

The $\Lambda$ function of this model is quite simple, and also assumes no height correlation between nearby points:

$$\Lambda(\omega) = \frac{-1 + \sqrt{1 + \alpha^2 \tan^2 \theta}}{2}. \tag{3.36}$$

The same technique (interpolating $\alpha$) can be used for anisotropic distributions.

#### *3.2.3.11 Torrance-Sparrow*

The Torrance-Sparrow model (TORRANCE; SPARROW, 1967) was designed to model metallic surfaces. It was introduced to computer graphics by Blinn (BLINN, 1977), and one variant of this model was used by Cook and Torrance (COOK; TORRANCE, 1981) (COOK; TORRANCE, 1982), which became an extremely popular model on its own right.

This model uses perfectly smooth mirrored microfacets. This means that only the ones that have the normal aligned with the half-vector between the incident and outgoing directions reflect light from $\omega_i$ to $\omega_o$.

The Torrance-Sparrow model finally gives us a BRDF that we can utilize in the rendering equation:

$$f_r(\omega_o, \omega_i) = \frac{D(\omega_h) G(\omega_o, \omega_i) F_r(\omega_o)}{4 \cos \theta_o \cos \theta_i}. \tag{3.37}$$

The only thing that the Torrance-Sparrow model assumes is the perfectly specular microfacets. Apart from that, it can be used with any microfacet distribution and with any Fresnel function $F_r$. A BTDF can also be derived:

$$f_t(\omega_o, \omega_i) = \frac{D(\omega_h) G(\omega_o, \omega_i)(1 - F_r(\omega_o))}{((\omega_o \cdot \omega_h) + \eta(\omega_i \cdot \omega_h))^2} \frac{|\omega_i \cdot \omega_h||\omega_o \cdot \omega_h|}{\cos \theta_o \cos \theta_i}, \tag{3.38}$$

where $\eta = \eta_i / \eta_o$. The half-angle transmission vector is $\omega_h = \omega_o + \eta \omega_i$.

### 3.2.4 Light Sources

In our abstraction, only light sources are allowed to emit light in the scene. Because of that, all the energy present in the scene must come from them. That also means any scene without light sources would be pitch black.

There are several different types of light sources that are useful when creating

scenes. Some of them are not completely realistic, but are still powerful tools when designing interesting scenes.

#### *3.2.4.1 Point Lights*

Point lights are infinitely small, and have constant radiance throughout its entire hemisphere. For ray tracing, since the light consists of a single point, there is only one point to sample.

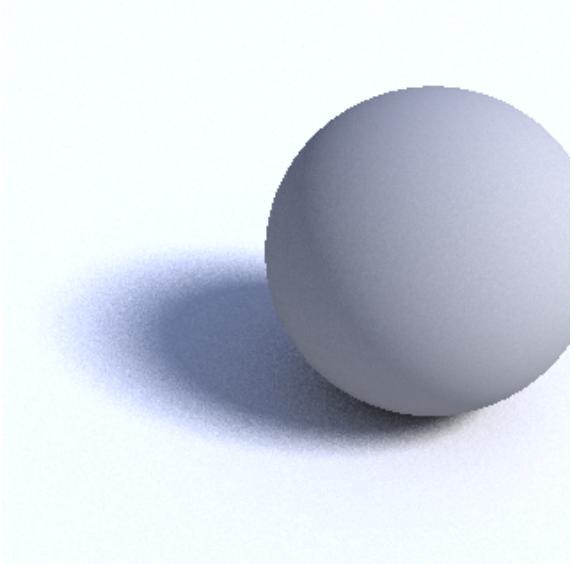
Point lights are the cheapest to incorporate in a scene, but have known problems: they are theoretical and don't really exist in the real world, so point lights almost always are approximation of area lights. That can be problematic, since point lights will cast very defined shadows, as opposed to the soft shadows created by area lights.

These lights are defined by a delta function. Since they are infinitely small, we need to sample the light directly to get valid samples, after all the chances of finding this point in space by sampling a separate distribution is mathematically 0. This makes implementing point lights quite tricky.

#### *3.2.4.2 Area Lights*

Area lights are an interesting abstraction: we define an area that has constant flux, and that flux will influence the scene accordingly. This area is generally defined via a geometric primitive, like a triangle. Area lights are more accurate to the real-world compared to their point counterparts, allowing for more sophisticated light interactions.

Figure 3.13 – Area lights are capable of casting soft shadows, like the one shown in this picture.



Source: extracted from Wikipedia

Area lights are capable of casting soft shadows in the scene, an interaction that can be seen in Figure 3.13. This effect comes from what is called partial occlusion: since the light is defined as an area, any point in the scene can see a fraction of the total area of the light. This proportion will then inform how big the contribution of the light is to that point.

Area lights can be sampled based on the geometry they are based on. These samples are important when raycasting for visibility checks. Ideally, a sampling strategy should be uniform in the whole area (if flux is constant). The implementation used will be described in the Implementation chapter.

#### *3.2.4.3 Infinite Area Lights*

For certain scenes, it makes sense to have a "background": pre-computed radiance values that are used when a ray misses everything in the scene. For that, we define an infinite area light, which is a light that encapsulates the whole scene and traditionally has its radiance values based on an already existing distribution.

This distribution can come from a lot of different places, but it generally comes from a high dynamic range image. This image contains values that are unbounded (not capped to 1), which represent radiance values for each direction in the hemisphere.

Mapping texture coordinates to the hemisphere is pretty simple. Considering $u, v \in [0, 1)$, and an outgoing ray defined by the angles $\theta$ and $\phi$:

$$u = \frac{\phi}{2\pi}, \tag{3.39}$$

$$v = \frac{\theta}{\pi}. \tag{3.40}$$

Sampling this type of infinite area light is simple: we just need to sample a point in the sphere that surrounds the scene. However, since the distributions used here generally contain a wide range of values, there are some caveats that will be discussed in a future section.

### 3.2.5 Summary

Now that we are familiar with each of its parts, let's summarize the full path tracing algorithm. The process that will be described here will happen for each pixel in the image, every frame. Each pixel is also completely independent from each other, which is great news for optimization purposes.

We start by sampling a point in the boundary of the current pixel. This point can be sampled by utilizing two independent random variables. As we've discussed, the number of samples of this initial dimension is directly responsible by the aliasing present in the final image.

Then, we trace a ray that starts in the camera's origin and goes through the sampled point. The film plane is generally defined at the focal distance from the camera, and its width and height will be defined by camera properties (vertical and horizontal FOV). The camera's origin can also be perturbed by a random number, simulating a lens with non-zero area. This can be useful for simulating effects such as defocus blur.

Now that we have our ray, we need to figure out which geometry in the scene is the closest possible intersection. This operation is made faster by using acceleration structures that represent the scene in tree-like fashion. Due to the number of primitives, intersection checking is generally the most costly operation of the whole process.

Once we reach an intersection, we have to figure out how much light that point is reflecting in the opposite direction of the ray. For that, we do two things: we first compute the effect of direct light on the surface, by directly sampling lights on the scene. This can either be done by sampling a probability distribution based on the light distribution of the scene, or by sampling a probability distribution based on the surface's light scattering distribution. Both can also be done, by weighting the samples with an heuristic. This is known as multiple importance sampling.

We also need to compute the light that comes reflected from other, non-emissive surfaces. For that, we cast another ray into the scene. This ray will start from this intersected point, and will go to a random direction on the unit sphere. The contribution of light found will be weighted by the surface's distribution, so the next ray's direction can also be importance sampled with the surface's distribution.

For each bounce, we have a chance $q$ of ending the ray (russian roulette). At the end of the path, all the contributions are then computed, and the final color of the sample is finished. For simplicity we use a simple box filter, so the final pixel color (before any

post processing, like temporal refinement, denoising or gamma correction) will be the average value of all samples in a pixel.

# 4 SYSTEM DESIGN

This chapter discusses our system, which implements most of the techniques discussed on Chapter 3. We discuss the approach taken for each of our requisites (defined on Chapter 1), and the decisions made when designing our system. We also discuss the technologies we used and how they affected our design.

## 4.1 Design Goals

Scene creation is a complicated process, but our system was designed to make it as simple as possible. To achieve this objective, our system had to be designed with some goals in mind.

### 4.1.1 Intuitive and Easy to Use

A big part of the difficulty in scene manipulation is created by the lack of accessibility in available tools. Our system should be easy to understand and use, and should not have a steep learning curve. We attack this problem by having a simple and clear user interface, alongside other straightforward commands. Most of the work should be done in the background by the system (and not by the user). If the user wants to simply change a material, then this operation should be as simple as selecting the new material itself.

### 4.1.2 Fast and High Quality Rendering

Even experienced artists may have difficulty understanding how a material change in a scene may affect the final result. This problem is aggravated if the user is not intimately familiar with PBR pipelines, requiring lots of iteration to perfect a certain look. This makes it immediately obvious why having fast feedback is important, but it also shows why having good feedback is crucial: ideally, the working image should not be an approximation, but as close as possible to a final render. Having different techniques for working and final rendering may cause unwanted visuals in the scene.

Recent progresses in hardware and software made real-time path tracing possible, allowing for scenes with complex light interactions to be rendered in a matter of millisec-

onds. To achieve this design goal, we utilize this advancements by rendering the working image directly with GPU-accelerated path tracing. To improve our results, we utilize of temporal refinement and machine learning-based denoising, greatly improving the quality of undersampled images and allowing for interactive framerates.

### 4.1.3 Familiar and Extendable

Graphics professionals are already familiar with a lot of other similar applications, which would mean that ignoring established conventions would not be wise. We designed our controls so that they can be as similar as possible to existing software, without sacrificing our other goals of ease of use and accessibility.

The architecture of the system itself should also be familiar. Adding new components, like materials, should be as easy as possible. And of course, perhaps the most important of all: the system should be compatible with existing scenes, since we want to use them as a base to create our own.

To achieve all of those goals, we heavily based our solution on PBRT (PHARR; JAKOB; HUMPHREYS, 2016), even though its use case is significantly different than ours: by aiming for GPU acceleration and user interaction, several changes and compromises are required when compared to a more traditional offline renderer. Still, this provided us a good baseline of comparison, and also easy compatibility of existing PBRT scenes and materials.

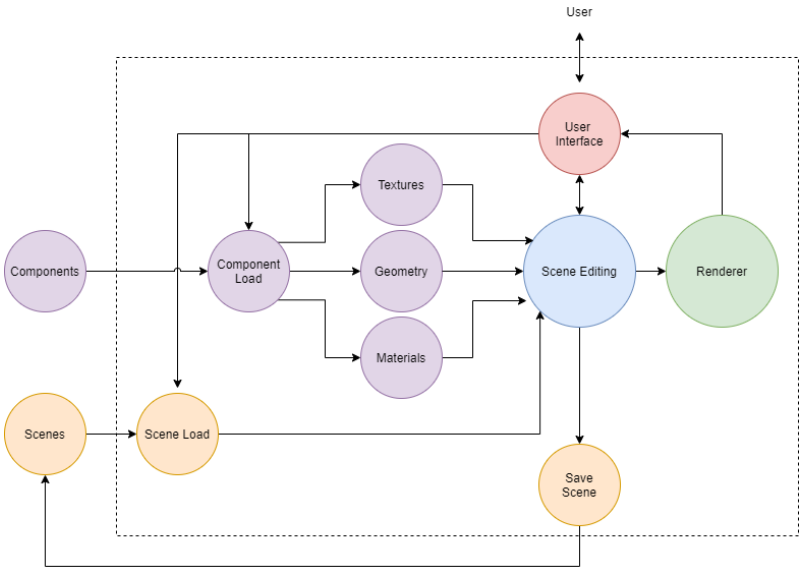
## 4.2 Architecture

Figure 4.1 shows a simplified overview of our system's architecture. In it, we see how the different modules communicate with one another, and also have a better idea of each part of the system's dependencies.

The nodes "Scenes" and "Components" represent files that are outside of our system, but can be loaded by their respective modules. We use external libraries to help the load process as well. At any time, the user can choose to load a new scene, or to load a new component (like a new texture) and use it in the current scene.

The "Scene Editing" node encapsulates all changes to the scene. It can modify a wide array of parameters, including material properties and position of objects, the

Figure 4.1 – An abstraction of our system's architecture.




Source: the authors.

current skybox, and path tracing parameters. Section 4.8 contains further discussion on its capabilities. The scene modifications can also be saved to create a new scene.

The information contained in the scene editing node will also be used to create a representation of the scene that can be rendered. This representation must also be updated whenever the scene is modified. More details on our renderer will also be provided on later sections.

## 4.3 Base Technologies

The code utilizes several established technologies to reach real-time performance. The main CPU code is written in C++. The GPU code utilizes CUDA via the OptiX framework (PARKER et al., 2010), optimized to utilize the existing ray-tracing acceleration hardware. We use Dear ImGui (CORNUT, 2021) for UI purposes.

A particularly influential work to us is NVIDIA's 2019 SIGGRAPH OptiX 7 course (WALD; PARKER, 2019). The original code of our renderer started with the course's provided code as a base, and until this day it uses some parts of it (like basic math classes).

### 4.3.1 NVIDIA CUDA

One of the most popular technologies among researchers today, CUDA (acronym for Compute Unified Device Architecture) allows for compiling code to run on the GPU. The code itself can be written with a traditional programming language (with extensions), before being compiled into PTX (CUDA's instruction set architecture).

After compiling, CUDA programs (called kernels) are dispatched to the GPU in grids, blocks and threads. This configuration can be changed by the programmer, and it is crucial for performance. They are then mapped to GPU lanes, wraps and streaming multiprocessors (SM). Threads mapped to the same SM have their own local memory, but they can also have access to their multiprocessor's shared memory (which also has its own cache). A lot of GPU optimizations involve optimizing memory fetches and writes for these reasons.

We write our GPU code in C++, and compile to CUDA at build time with NVIDIA's own nvcc compiler. It is interesting to note that there are ways of compiling CUDA code at runtime, but those result in a performance cost so they were not used. After compiling, we can utilize our PTX code with OptiX's API.

### 4.3.2 NVIDIA OptiX

To utilize the existing ray tracing hardware, we use the NVIDIA OptiX API. Since version 7, it is stateless and asynchronous. It can be considered an extension of CUDA, and much of its functionalities are based on top of the existing CUDA framework.

Differently than traditional CUDA, OptiX programs can't assume some key factors of GPU allocation: tasks (like rays) can be moved at any time to a different lane, warp or streaming multiprocessor. For this reason, many of CUDA's tools (like barriers or shared memory) are unavailable.

OptiX is designed to facilitate ray-tracing applications. To use it, we need to comply with its directives and data structures. Even though OptiX takes care of some important steps, there are still things that need to be defined by the programmer. Further discussion on OptiX's requirements and specification will occur on later sections.

### 4.3.3 OpenGL

For a simple way to draw a texture on the screen, we utilize OpenGL. It is a very high level API, and as such, is generally slower than its lower-level counterparts (like Vulkan or DirectX). We utilize it in conjunction with the GLFW library, which interacts with the current operating system's API to create and manage windows.

### 4.3.4 Scene Loading

To parse and include PBRT scenes, we utilize Ingo Wald's PBRT-Parser (WALD, 2021). Not all PBRT features are supported by our renderer, so only a subset of scenes work properly.

To parse and include obj files, we utilize tinyobjloader (FUJIYA, 2021b). It is single file with no dependencies (except C++'s standard library).

For loading images, we utilize stb (BARRETT, 2021) for standard formats, and for high dynamic range one we use tinyexr (FUJIYA, 2021a).

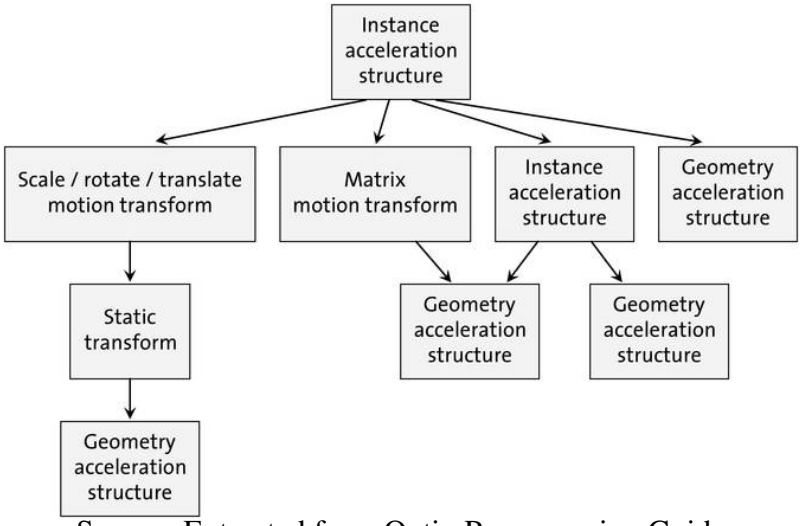
## 4.4 Acceleration Structures

To accelerate ray tracing, OptiX uses traditional acceleration structures. They must be built calling OptiX's respective functions, obfuscating the underlying algorithms to the user. The resulting acceleration structure will be a BVH, but its properties will be defined by the current device (like GPU model). This makes the tree optimized for different architectures automatically.

To construct the tree itself, the user has two building blocks: they are called "Geometry Acceleration Structure" (GAS) and "Instance Acceleration Structures" (IAS). GAS-type nodes can be seen as leaves of the tree, as they contain geometric primitives. IAS-type can point to other nodes, including transform nodes, that are used to differentiate between instances of the same geometry. Figure 4.2 shows the different type of "tree nodes" that OptiX provides.

In our system, we build one GAS for every mesh when they are loaded. These GASes are then instantiated and linked by an IAS, resulting in a two-level scene overall. Duplicating meshes will only result in a new instance being created, which is way

Figure 4.2 – An example of acceleration structure tree, showing possible connections between nodes.




Source: Extracted from Optix Programming Guide

faster and less memory intensive than duplicating geometry information (like triangles). Instances are differentiated by their transform.

When the transform properties of any instance is changed or when a instance is duplicated or deleted, the acceleration structures must be updated. When this happens, the overlying IAS must always be rebuilt, but only the affected instance structure must be changed. GASes are never rebuilt.

## 4.5 Ray Programs

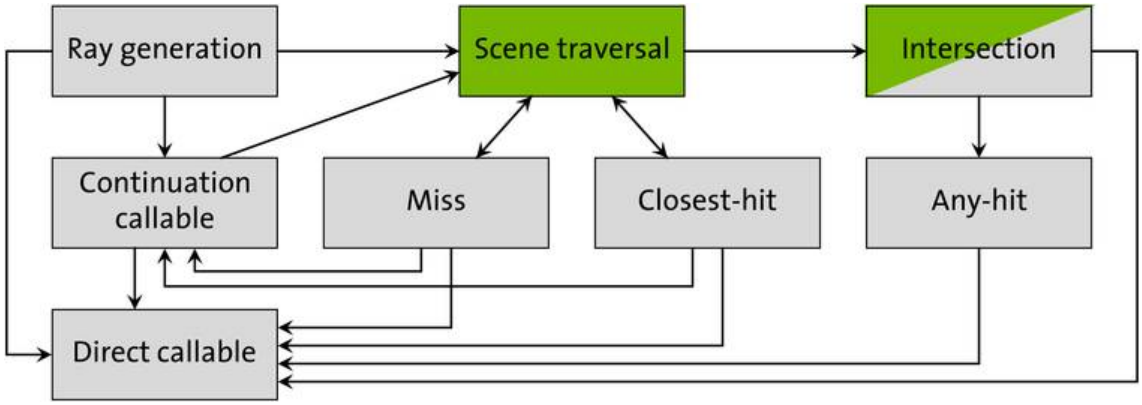
The actual code that is run on the GPU is called a program (by the OptiX API). These are functions written in CUDA with special keywords that indicate their extra ray-tracing capabilities. Other ray tracing APIs also have similar paradigms in regards to program flow.

### 4.5.1 Types

A program can have several different types, indicating the role that they play in the overall ray tracing algorithm. Figure 4.3 shows the possible data flows that an OptiX application can have.

The ray generation type (also called "raygen") is the first one in the pipeline, and will be the one that will be directly called by the OptiX launch. The launch itself will

Figure 4.3 – A diagram showing the usual data flow of an OptiX program. Green nodes represents fixed function code, and gray represents user programs.




Source: Extracted from Optix Programming Guide

specify some parameters (e.g. the launch dimensions), and those parameters will determine how many instances of the ray generation program will be called. Rays can be traced from the ray generation code via API calls, and as such, at the ray generation program we must determine the position and direction of the rays we will trace.

The miss program is called whenever a ray does not intersect anything from its minimum $t$ value to its maximum $t$ value. The miss program can be used to add the skybox contribution to the scene, allowing for the incorporation of synthetic or real life probes that light the entire scene.

The intersection program is an optional step that must be invoked for every custom primitive in the scene. It is called if the bounding box of the primitive is intercepted by a ray, and will determine if the hit is valid or not. For triangle meshes there is no need for this kind of program, since the ray-triangle intersection is directly accelerated by hardware. The context change caused by intersection program calls (and running the code itself) is significant, and as such, it can be beneficial to represent custom geometry with triangle meshes approximations.

The any-hit program is called for every possible closest hit intersection in the scene. Then, its code can determine if the hit is accepted or discarded and if the trace should continue or not. Since it can also write on the ray payload (and memory) it can also be used for more sophisticated uses. A common use for these kinds of programs is for checking shadow rays[1], since they are generally only interested if there's a intersection at all as opposed to figuring out the closest hit. Another common use is alpha testing, where a texture lookup is required to determine if the hit is valid or not.

[1]OptiX also allows for ray flags on each trace, allowing for instant termination on first hit and, in the case of shadow rays, not even requiring an any-hit shader.

The closest-hit program is called after the closest point of intersection is found. It's generally where the actual shading is done, even though some architectures just use it to write intersection information on the payload and shade elsewhere.

There are also callable programs: essentially, functions that can be called from any program, but since they are declared on the shader binding table (discussed in section 4.6), a different callable program can be called at runtime based on its index. This allows for greater flexibility in general. Callables were not used in our renderer.

### 4.5.2 Compilation

All OptiX programs must be written in CUDA code, and compiled before the OptiX launch. We decided to compile our programs at build time using NVIDIA's nvcc compiler, but there are alternatives that allow for runtime compilation. Obviously, this would cost runtime performance.

The resulting PTX code is then utilized by the API functions to create the required internal structures, allowing for them to be used by the ray tracing pipeline.

## 4.6 Shader Binding Table

When a ray intersects geometry, it needs to know which programs to call. That is determined by the shading binding table (SBT), which associates each instance with its hitgroup programs. Using information contained in the instance's acceleration structure, the OptiX framework knows which address of the SBT to access for each primitive.

The SBT may also store any information related to the primitive that can be accessed in the GPU programs. These can be any parameter traditionally used in shading, like vertex normals, albedo, textures or anything else that the shading model requires. The SBT also has similar entries for miss and raygen programs, allowing for those parameters and resources to be accessed on the GPU by each instance.

When any instance's properties change or when a instance is created or deleted, the hitgroup's shader binding table entry must be rebuilt to account for the change.

OptiX also supports the concept of ray types: each ray type may call different programs. For each ray type, the SBT must point correctly to the desired program for every primitive, and each ray type must also point correctly to a miss program. In our

renderer, we utilize this concept for light sampling.

## 4.7 Initialization

To aid users with easy scene and skybox selection, we load a config file at startup. It can register several scenes and skyboxes' system path, which may then be utilized at any time. The default config file should be in the same directory as the executable, but a different config file can be passed via command line arguments. The program only uses the config file to save a list of paths, which will then be loaded when requested by the user.

After that, initialization is pretty straightforward: the first scene in the list is parsed and loaded. It's here that the PBRT materials are adapted to our own representation: while our material implementations are very close to PBRT's, we only support a subset of PBRT materials. Unsupported ones, when possible, are approximated. This will be discussed further on chapter 5.

There are several APIs that need initialization too: not only CUDA and OptiX, but also Dear ImGui (that we use for our UI) and OpenGL. Initializing the OptiX data structures takes most of the initialization process. Luckily, most of this work will not be repeated after the initialization process is done.

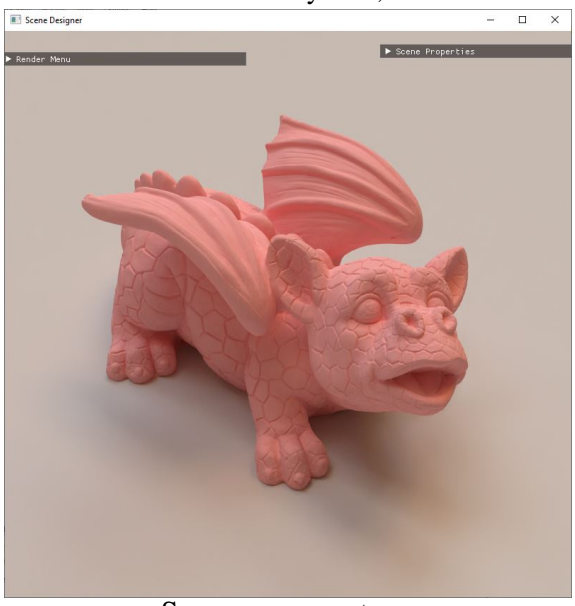
At initialization, we also need to build both the SBT and acceleration structures that OptiX will use. The SBT is built on the CPU and then transferred to the GPU, and the AS nodes are built using OptiX's own API and structures. After the build step (that happens on the GPU), the AS is ready to be traced against. Both will be rebuilt when necessary, as described in sections 4.4 and 4.6.

## 4.8 User Interaction

In this section, we will describe some of the features that can be utilized to modify scenes in real-time. Most of them are implemented using GLFW and Dear ImGui.

Figure 4.4 shows the main window, displaying the current approximated image to the user.

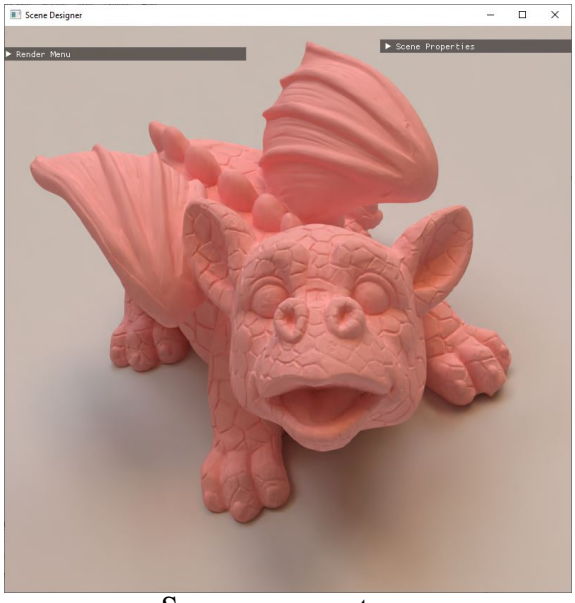
Figure 4.4 – A screenshot of our system, with menus minimized.




Source: our system.

### 4.8.1 Basic Controls

Figure 4.5 – A screenshot of our system, with menus minimized. The camera was altered by the user, getting closer to the dragon object and changing the angle.

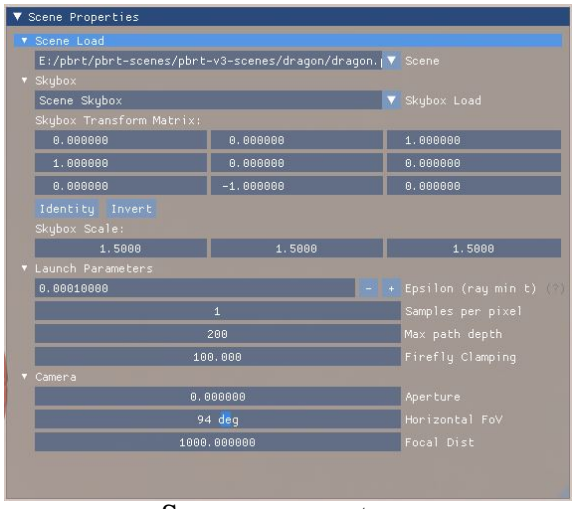



Source: our system.

The scene's camera can be re positioned, allowing for inspecting elements more closely or just changing the final image's framing (shown in Figure 4.5). Resizing the window will also change the resolution of the final image.

### 4.8.2 Scene Properties

Figure 4.6 – The scene properties menu, displaying all sections.




Source: our system.

The Scene Properties menu (Figure 4.6) controls several scene parameters. All of them can be changed at runtime. We will discuss some of the most important ones here.

Using the scene load list, the user can load different scenes at runtime. This list of scenes is built by reading a config file provided at startup. The scenes are loaded into memory when the selection is made, the list of strings is only presented to help the user.

We also can control skybox-related parameters. First, the texture itself can be changed at runtime, by simply selecting a new one from a list. This list is also built by the config file loaded at startup, and a skybox can also be provided directly by loading PBRT scenes that have a built in skybox. This feature is showcased in figure 4.7.

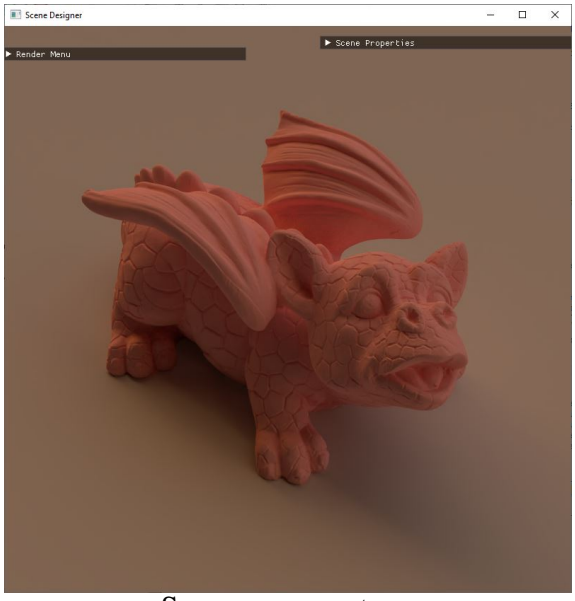
The current transform matrix applied to the map can also be modified. This transform is applied to all rays that hit the skybox's miss programs. It can determine things like the rotation of the skybox. The skybox's value itself can also be scaled directly by a constant vector that can be changed at runtime.

The "Launch Parameters" section controls properties for the path tracing algorithm itself.

Ray-triangle intersections are prone for floating point precision errors. For that reason, we need methods to avoid self-intersection when casting a ray starting at a triangle. The most common method is to add a very small value to the minimum ray distance. This small value (called epsilon) can be controlled here.

The max path depth can be also be determined. The ray may be terminated before this depth is reached by reaching a light source, having no throughput left or russian

Figure 4.7 – The dragon scene, with a modified skybox texture.




Source: our system.

roulette.

To improve convergence rates, it is common to clamp very high valued samples in your image. These values are called "fireflies" (they appear as white pixels on the screen). This method adds bias to the algorithm, but can help a lot in certain scenes, so it is generally considered an acceptable trade-off. The clamping threshold can also be changed at runtime. It is interesting to note that the denoiser (that will be discussed later) is particularly sensitive to fireflies, so this feature is very important.

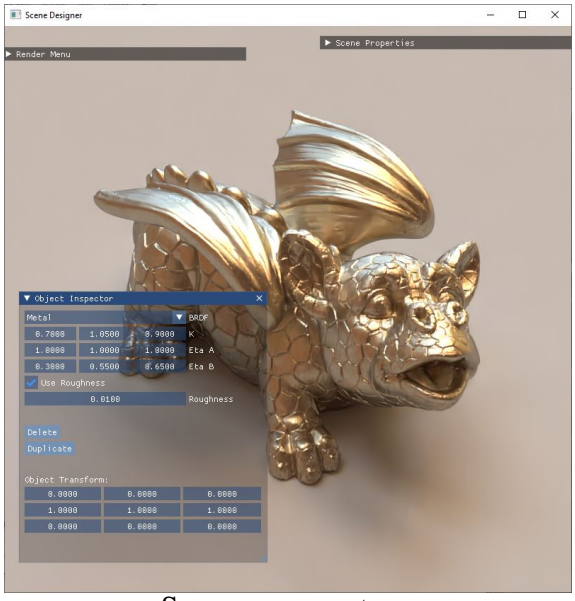
And finally, we also can change some camera properties in the "Camera" section. By changing the aperture, field of view and focal distance, we can directly influence how the rays are generated in the raygen program.

### 4.8.3 Object Inspector

One of the most useful features of our renderer is the object inspector. This menu can be seen in Figure 4.8. By selecting the desired instanced mesh, we can analyze its current material and change it or any of its properties. We can also change the instance's transform, duplicate it or delete it from the scene. Such an example is shown in Figure 4.9.

For the mesh selection to work, we use the mesh ID buffer. It is built in the OptiX launch, and it works essentially as a lookup that informs the mesh that was closest hit by a ray in each pixel.

Figure 4.8 – The object inspector menu, after selecting the dragon mesh and modifying it with a different material. It shows the current material (Metal), with its properties.




Source: our system.

Figure 4.9 – A modified dragon scene, with duplicated instances, each with their own materials and transforms.

Source: our system.

Each material has its own list of properties, and the UI will update accordingly to reflect that. Our material list is based on a subset of PBRT. The equivalence of our materials with PBRT's will be discussed later on Chapter 5.

Instances can be transformed using the UI, but they can also be manipulated using the mouse directly. Any instance can be translated, rotated or scaled. This is applied directly via a transform matrix that is composed by the individual transformation matrices. As such, these operations are limited by existing characteristics of the mesh (for example, texture coordinates will be stretched when a scale operation is performed).

New textures can be applied to an object at runtime. When selected, the texture will be loaded and applied correctly on an object using its texture coordinates. Textures

are only created once, and are shared for every object that uses them.

## 4.9 Render Loop

Starting the render process, we assume that the acceleration structures and the SBT are already built. Any changes necessary should be completed before rendering starts.

Every frame starts by updating the current launch parameters. The structure contains information that must be utilized by the next launch and that can changed at any time. For example, our launch parameters include properties of the current camera: not only position and direction, but also aperture and focal distance. These will be read by the next step in the pipeline when generating rays. It also contains pointers to all final buffers that must be written (which will be discussed later), and final image sizes. These can change based on the active window, so they must be updated accordingly.

### 4.9.1 Raygen Program

After the launch parameters update we can finally deploy GPU code. One instance of our raygen program for each final image pixel is deployed. All of them can be run at parallel, and will be scheduled by OptiX itself.

The raygen program is responsible for the core loop in the path tracing algorithm. To avoid wasteful stacking and maximize memory usage, the algorithm is iterative (as opposed to recursive), and always returns to the raygen program between new nodes in the path. It starts by generating a pixel sample, which will then be converted to a ray based on camera properties. This pixel sample is randomized for each pixel and each pixel starts with a different random seed (using LCG), avoiding noise that might arise from correlation between pixels.

After that, the ray payload is prepared. A pointer to this structure will be passed to subsequent programs in the ray path, making it the main form of communication between the programs in this pipeline. With all of the preparations done, we finally trace the prepared ray.

### 4.9.2 Closest-Hit Program

If a primitive is hit, its associated closest hit program will be deployed. Most of the material-dependent calculations are done via this program: as such, there are a lot of different closest-hit program variants implemented by our code. The program associated with the mesh is decided based on the desired material. Each material expects different parameters, and as such, they must be prepared beforehand in the SBT build step. To avoid stacking (and declaring OptiX callables), our implementations make heavy use of function inlining.

After getting from SBT the current primitive's properties, we start by calculating the direct lighting contribution for that particular point (a practice that is popularly called as next event estimation). We draw two samples per point: one will be sampled from a distribution that approximates the light contribution, and the other from one that approximates the material's BSDF distribution. In this context, sampling means tracing rays in the sampled directions to check for emissive contributions. Of course, if the closest hit intersection of that ray is not emissive, the contribution will be zero. These two samples will then be scaled by multiple importance sampling (MIS) using the power heuristic.

The light distribution is sampled by randomizing one of the light sources in the scene to sample. This is a uniform selection: for improvements, a distribution based on light power could be utilized. If the chosen light is a triangle area light, a point is uniformly sampled across the triangle area, which will then be used as the ray destination. If the chosen light is the skybox of the scene, a simple cosine distribution based on the surface normal is utilized.

The material distribution, obviously, changes depending on the material. For example, Lambertian materials sample the cosine distribution (again, based on the surface's normal), and microfacet models have each their own way of sampling (mostly using a sample of its microfacets normal distributions).

It is interesting to note that perfectly specular materials do not have two samples: since it has only one direction where any contribution counts, checking for other directions would be wasteful. Because of this, specular materials require special attention even on the main path tracing loop in the raygen program.

Each ray will then be traced using a different ray type than the usual path-node one. This ray type will only return contribution from light sources, and will terminate if it hits anything else on the way. These rays are traced directly from the closest hit programs

(without returning to the raygen program), but since the resulting programs can't spawn more rays themselves it still caps the maximum program depth at 3.

The contributions are then scaled by the power heuristic and written into the payload structure. The next ray in the path is sampled from the material distribution, and as such, must also be done in the closest-hit. Next-ray properties and the new overall path throughput (based on the next ray's PDF) are also written in the payload.

### 4.9.3 Miss Program

If the raygen-traced ray misses the scene's primitives, the miss program is deployed. It samples the scene's skybox (if it has any) based on the ray direction, and returns the appropriate contribution. This contribution can also be scaled by a parameter provided by the scene (or overwritten by the user at runtime).

### 4.9.4 Accumulation and Denoising

After that, the program returns to the main loop. The current frame's contributions are accumulated, and if this was the first bounce, the values for albedo, normal and mesh ID buffers are saved. If the path is past a threshold then Russian roulette is enabled, allowing for paths to be terminated by a stochastic process. After that, the actual buffers pointed to by the launch parameters are updated, accumulating the current frames' samples with the already existing ones.

This process (called temporal refinement) works wonderfully for us: it allows each frame to be reasonably fast (only drawing one sample per pixel) and, with accumulation, still result in a converged image in a short period of time.

The resulting buffers will then be used for denoising the current image. We use OptiX's built-in denoiser, that takes as parameters all three: color, albedo and normal buffers. The denoising itself is AI-based, and runs on the GPU. It runs on the image still on linear space (before gamma correction), and greatly reduces noise. Figure 4.10 shows the impact that the denoiser has on the final output.

Denoising is crucial to achieving real-time performance: even with all the optimizations possible, it still isn't viable to path trace the amount of samples required to eliminate noise from an image and still have frame times that are low enough to be inter-

Figure 4.10 – Living Room scene, rendered with a small number of samples (a). This image, when applied to the denoiser, produces a much more appealing result (b).

(a) Original

(b) Denoised

Source: our system.

active.

After all those steps, a simple gamma correction is applied to map the radiance to image values. This buffer is only used for showing the image (and saving it, if the user chooses to), temporal refinement is done directly on the output buffers from the OptiX launch. The final image is then rendered to the window with OpenGL.

# 5 RESULTS

This chapter presents some scenes rendered by our system. We also discuss similarities and differences from their PBRT counterparts, and some limitations of our renderer. All results reported in the paper were rendered using a Core i5 6500 3.2 GHz CPU with 16 GBs of RAM, NVIDIA RTX 2080 TI. The images generated with our system were rendered on the GPU, while the PBRT images shown for comparison were rendered on the CPU.

## 5.1 Modified Scenes

Figure 5.1 – Modified "Country Kitchen" scene, with instance duplication, deletion, material changes and transform manipulation.

(a) Original Scene

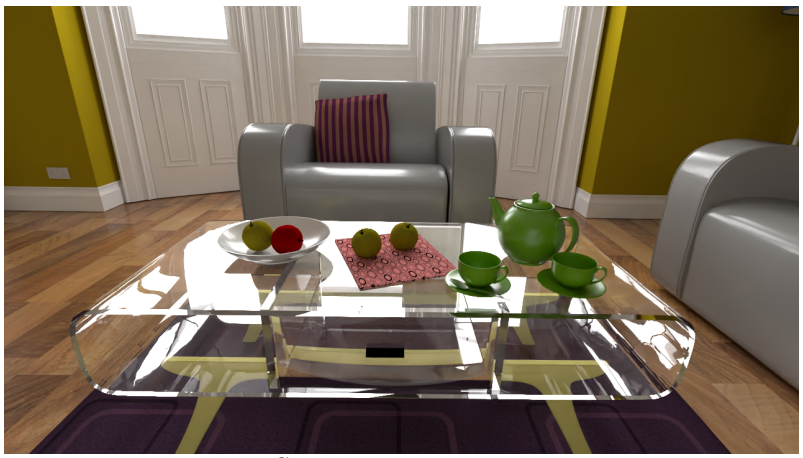
(b) Modified Scene

Source: our system.

We present here some scenes created by our system based on existing ones. The original scenes are compared with our modified versions directly, since in this section

Figure 5.2 – Modified "The White Room" scene, with a different camera, materials and textures.
(a) Original Scene

(b) Modified Scene



Source: our system.

the original scenes had their camera moved for a closer comparison. The original scenes (with their original cameras) can be seen in the next section.

Figure 5.1 shows a kitchen scene created with our system by modifying an existing one. Figure 5.1 (a) shows the original scene. Its edited version is shown in Figure 5.1 (b). It showcases a bigger table with additional rearranged chairs, plates and cups. For this scene, the kitchen island was removed and two of the walls were recolored with different materials. This scene was created during an interactive session under just a few minutes by a non-artist. The quality of this result supports our claim that while creating new scenes from scratch is a difficult task, editing existing ones can be successfully done by an average user.

Figure 5.2 shows another example of modified scene. Figure 5.2 (a) shows a view of the original scene. Figure 5.2 (b) shows the resulting scene created using our system and illustrates several of its features. These include, for instance, change of material properties (e.g., coffee table, teapot, and teacup), texture change (pillow and small table

Figure 5.3 – Variations of "The White Room" scene, with different cameras, materials and textures.

Source: our system.

cloth), object replication (teacup and fruits), object translation (pillow and fruits), and color changes (walls and fruit). This modified scene was created in just three minutes and the entire process is illustrated in the supplementary video.

Figure 5.3 shows more variations that can be created using the same initial scene as Figure 5.2. Both variations only differ in material properties and textures. This shows that meaningfully different scenes can be created by only using simple changes that can be done in a matter of minutes.

Figure 5.4 provides another example of scene edited with our system. Figure 5.4 (a) shows a view of the original *Bathroom* scene exhibiting different kinds of materials and textures. Figure 5.4 (b) show a modified version of this scene created using our system in just a few minutes. Among the many changes, the larger mirror has been replaced by two smaller ones.

Figure 5.5 provides an additional example of scene edited with our system. Figure 5.5 (a) shows a modified view of the original *Wooden Staircase* scene exhibiting standard materials and textures, while Figure 5.4 (b) shows the same view but with vastly

Figure 5.4 – Modified "Bathroom" scene, with a different camera, materials and textures. The large mirror has also been replaced by two smaller ones.

(a) Original Scene (b) Modified Scene

Source: our system.

different materials and textures.

## 5.2 Scene Comparisons

Table 5.1 – Render times (128 spp)

| *Scene* | *PBRT* | *Our Renderer* | *Speedup* |
|---|---|---|---|
| Contemporary Bathroom | 617.2s | 8.795s | 70.17x |
| Salle de bain | 597.4s | 6.409s | 93.21x |
| Bedroom | 742.5s | 6.378s | 116.41x |
| Cornell Box | 252.9s | 2.173s | 116.38x |
| The Breakfast Room | 637.2s | 7.269s | 87.66x |
| Country Kitchen | 602.5s | 5.484s | 109.86x |
| The Grey & White Room | 767.6s | 5.481s | 140.06x |
| The White Room | 682.8s | 5.588s | 122.19x |
| The Wooden Staircase | 418.8s | 2.471s | 169.49x |
| Modern Hall | 449.8s | 7.856s | 57.26x |

We present some results comparisons between PBRT and our own renderer. All images were rendered with 128 samples, and with the denoiser post-pass in our solution. These scenes were all taken from Bitterli's rendering resources (BITTERLI, 2016) and PBRT's own scene repository. Figures 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13 and 5.14 show direct comparisons between our system's renders and PBRT's. The images show the quality of the rendering results produced by our system at interactive framerates, while the scenes themselves contain several interesting features, such as area light sources and

Figure 5.5 – Modified "The Wooden Staircase" scene, with different camera, materials and textures.

(a) Original Scene (b) Modified Scene

Source: our system.

translucent materials.

Despite being generated by an interactive system, the quality of the images generated by our system is comparable to ones produced by offline physically-based renderers, such as PBRT. When using the same number of samples, the use of temporal refinement and denoising allows our system to produce better results than plain PBRT without denoising. This is illustrated in Figure 5.7, where both images were rendered using 128 samples per pixel. Figure 5.7 (a) shows the results produced by our system using denoising, while the results in (b) were generated by PBRT without denoising. Note how our denoised results are significantly better than the noisy ones. If noise is disregarded, one can see that the results produced by both systems are qualitatively similar.

Our renderer was heavily based on PBRT's architecture, so it's natural that the images are very close. They are not equal, however: we can notice subtle differences on most comparisons. These can come from a variety of factors, like purposeful design decisions to simple bugs.

Figure 5.6 – "Cornell Box" scene, rendered (128 spp) in our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

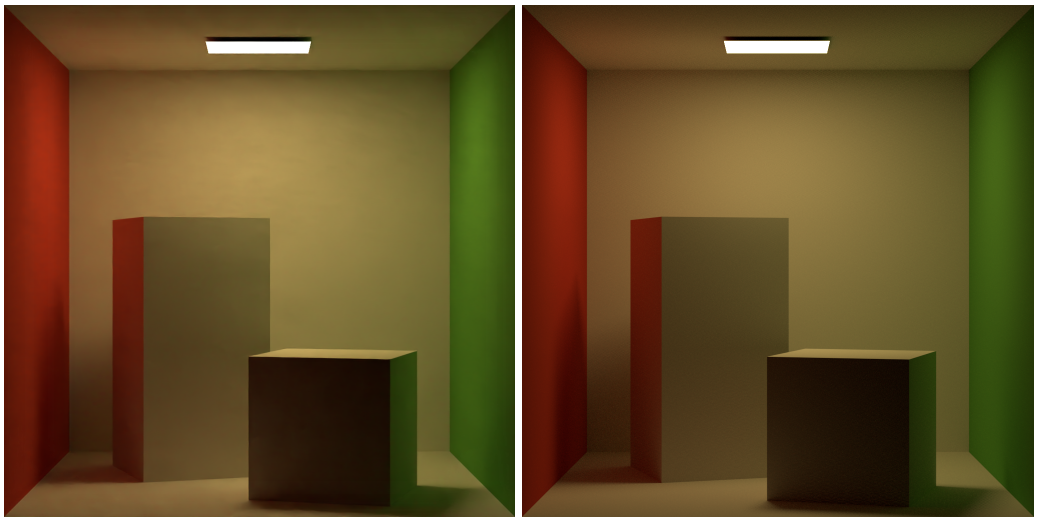


Figure 5.7 – "Contemporary Bathroom" scene, rendered (128 spp) in our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

A good example of purposeful changes can be seen in figure 5.10. The way that the plant's materials deal with textures was changed so that it behaves better with scenes like this (these scenes were converted from Tungsten, another renderer).

It can also be noted that some of these images have not yet converged on PBRT with 128 samples, but generally, the OptiX denoiser makes so that is more than enough for us. It could be said that, for a more accurate comparison, PBRT's number of samples should be higher, increasing convergence but also render times. Table 5.1 shows a performance comparison between both renderers using the same number of samples per pixel.

Figure 5.8 – "Salle de bain" scene, rendered (128 spp) in our renderer (a) and PBRT (b).
(a) Our Renderer

(b) PBRT

## 5.3 Material Comparisons

Our renderer implements several BRDFs (see section 3.2.3 for explanations and more result images). Some of them can be utilized to replicate or approximate PBRT's materials. This is shown in Figures 5.15, 5.16, 5.17, and 5.18.

All of our materials are implemented with closest-hit programs. This raises some challenges when adapting PBRT materials, since their collection of BRDFs can change based on each intersection point's properties. This adaptation process can result in slightly different end results.

Another challenge was adapting PBRT's coordinate system to our own. Skybox rotation in particular was hard to replicate, and incorrect rotation might still be a problem in some scenes (like 5.17).

Figure 5.9 – "Country Kitchen" scene, rendered (128 spp) in our renderer (a) and PBRT (b).
(a) Our Renderer

(b) PBRT

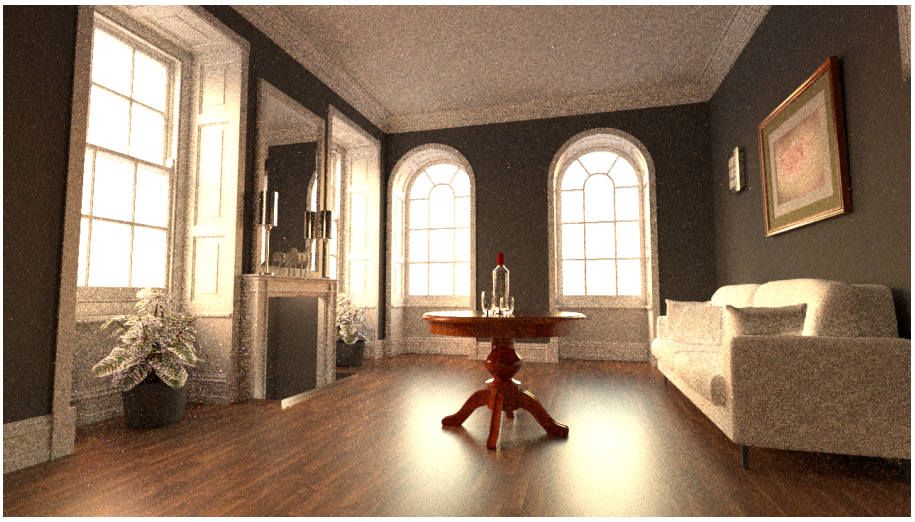
Figure 5.17 also highlights another challenge: PBRT has a robust method of dealing with floating point precision, severely mitigating the self-intersection problem. Our solution is a simple (but common) one: we simply add a little epsilon value for each intersection (that is controllable by the user at runtime as described in section 4.8). This is not nearly as accurate as PBRT's method, and results in self-intersection artifacts. To further mitigate this problem we also use a index-based method (an intersection is ignored if the previous intersection has the same primitive and mesh index).

Figure 5.18 showcases that even though the overall surface "roughness" is very close, the color of our metal material approximation is a little bit off.

Figure 5.10 – "The Grey & White Room" scene, rendered (128 spp) in our renderer (a) and PBRT (b).

(a) Our Renderer

(b) PBRT



## 5.4 Limitations

Our renderer supports only a subset of PBRT scenes. Features like participating media and subsurface scattering were not implemented, and more complex materials (like the Uber Material) were greatly simplified. The decision to not implement these features was taken to limit the scope of this project, but future improvements to the system are extremely likely.

A particularly interesting subject to discuss is that our renderer only supports traditional path tracing. Future work could implement more robust rendering techniques, like bidirectional path tracing and Metropolis-Hastings based approaches. Hardware-accelerated ray tracing is so fast, however, that in most scenes these methods are beyond

Figure 5.11 – "The White Room" scene, rendered (128 spp) in our renderer (a) and PBRT (b).
(a) Our Renderer

(b) PBRT

overkill, and convergence (with a denoiser) is reached in a few seconds.

A characteristic of our renderer (and other GPU-based solutions) is that a scene must be able to be loaded entirely in memory. Techniques like instancing do help, but for extremely complex movie-quality scenes that might not be enough.

Figure 5.12 – "The Wooden Staircase" scene, rendered (128 spp) in our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

Figure 5.13 – "Modern Hall" scene, rendered (128 spp) in our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

Figure 5.14 – "Bedroom" scene, rendered (128 spp) in our renderer (a) and PBRT (b).

(a) Our Renderer

(b) PBRT

Figure 5.15 – Dragon model, with a "Matte" material. Rendered with the same parameters on our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

Figure 5.16 – Dragon model, with a "Substrate" material. Rendered with the same parameters on our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

Figure 5.17 – Dragon model, with a "Glass" material. Rendered with the same parameters on our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

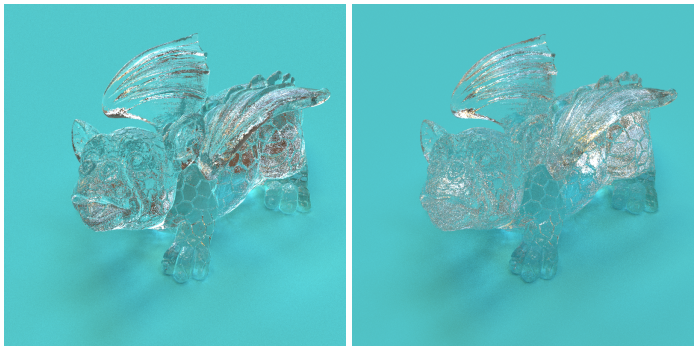


Figure 5.18 – Dragon model, with a "Metal" material. Rendered with the same parameters on our renderer (a) and PBRT (b).

(a) Our Renderer (b) PBRT

# 6 CONCLUSION AND FUTURE WORK

We presented a powerful scene creation system, with cutting edge rendering powered by modern technology that achieves interactive frame rates while doing full path tracing on the GPU. It allows for easy and powerful modifications of existing scenes, empowering users with the capability to unleash their creativity without having to learn complex toolsets and concepts.

Real time ray tracing techniques are no longer a distant future. With the advent of dedicated intersection hardware, even offline renderers present significantly improved performance. The incorporation of machine learning for denoising images was also key to achieving faster convergence. Our system shows that there's already a place for real-time path tracing solutions in problems that we face today.

We showed that our solution's results are comparable to PBRT, an established render, while still being faster by at least an order of magnitude. When compared to even higher sampled images, it can still present better results by utilizing the built-in OptiX denoiser.

This system could be easily extended in the future, and logical follow-up work could add more interactive functionalities to the already existing ones. The renderer could also be improved, since not all of PBRT features are implemented (and possibly features that aren't even on PBRT). Another interesting addition would be to add support for more scene formats, allowing for easy manipulation of scenes from different solutions altogether.

## REFERENCES

BARRETT, S. **stb**. 2021. Available from Internet: <https://github.com/nothings/stb>.

BECKMANN, P.; SPIZZICHINO, A. The scattering of electromagnetic waves from rough surfaces. Pergamon, New York, NY, USA, 1963.

BITTERLI, B. **Rendering resources**. 2016. Https://benedikt-bitterli.me/resources/.

Blender Online Community. **Blender - a 3D modelling and rendering package**. Blender Institute, Amsterdam, 2018. Available from Internet: <http://www.blender.org>.

BLINN, J. F. Models of light reflection for computer synthesized pictures. **SIGGRAPH Comput. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 2, p. 192–198, jul. 1977. ISSN 0097-8930. Available from Internet: <https://doi.org/10.1145/965141.563893>.

CHAOS-GROUP. **V-Ray**. 2021. Available from Internet: <https://www.chaosgroup.com/vray/collection>.

COOK, R. L.; PORTER, T.; CARPENTER, L. Distributed ray tracing. **SIGGRAPH Comput. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 3, p. 137–145, jan. 1984. ISSN 0097-8930. Available from Internet: <https://doi.org/10.1145/964965.808590>.

COOK, R. L.; TORRANCE, K. E. A reflectance model for computer graphics. In: **Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: Association for Computing Machinery, 1981. (SIGGRAPH '81), p. 307–316. ISBN 0897910451. Available from Internet: <https://doi.org/10.1145/800224.806819>.

COOK, R. L.; TORRANCE, K. E. A reflectance model for computer graphics. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 1, n. 1, p. 7–24, jan. 1982. ISSN 0730-0301. Available from Internet: <https://doi.org/10.1145/357290.357293>.

CORNUT, O. **Dear ImGui**. 2021. Available from Internet: <https://github.com/ocornut/imgui>.

FUJIYA, S. **tinyexr**. 2021. Available from Internet: <https://github.com/syoyo/tinyexr>.

FUJIYA, S. **tinyobjloader**. 2021. Available from Internet: <https://github.com/tinyobjloader/tinyobjloader>.

GROSS, M. H. Computer graphics in medicine: From visualization to surgery simulation. **SIGGRAPH Comput. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 32, n. 1, p. 53–56, feb. 1998. ISSN 0097-8930. Available from Internet: <https://doi.org/10.1145/279389.279462>.

HAGEMANN, L.; OLIVEIRA, M. Scene conversion for physically-based renderers. In: . [S.l.: s.n.], 2018. p. 226–233.

KAJIYA, J. T. The rendering equation. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 20, n. 4, p. 143–150, aug. 1986. ISSN 0097-8930. Available from Internet: <http://doi.acm.org/10.1145/15886.15902>.

LAFORTUNE, E. P.; WILLEMS, Y. D. Bi-directional path tracing. In: **Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)**. Alvor, Portugal: [s.n.], 1993. p. 145–153.

LOMMEL, B. V. **Cycles X**. 2021. Available from Internet: <https://code.blender.org/2021/04/cycles-x/>.

NIMIER-DAVID, M. et al. Mitsuba 2: A retargetable forward and inverse renderer. **Transactions on Graphics (Proceedings of SIGGRAPH Asia)**, v. 38, n. 6, dec. 2019.

NVIDIA. Nvidia turing gpu architecture. sep. 2018. Available from Internet: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>.

O'CONNOR, R. **Autodesk 3Ds Max 2016 - Modeling and Shading Essentials**. 1st. ed. USA: CreateSpace Independent Publishing Platform, 2015. ISBN 1517185815, 9781517185817.

OREN, M.; NAYAR, S. K. Generalization of lambert's reflectance model. In: **Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: Association for Computing Machinery, 1994. (SIGGRAPH '94), p. 239–246. ISBN 0897916670. Available from Internet: <https://doi.org/10.1145/192161.192213>.

PARKER, S. G. et al. Optix: A general purpose ray tracing engine. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, jul. 2010. ISSN 0730-0301. Available from Internet: <https://doi.org/10.1145/1778765.1778803>.

PHARR, M.; HUMPHREYS, G. **PBRT, Version 3**. 2015. Available from Internet: <https://github.com/mmp/pbrt-v3>.

PHARR, M.; HUMPHREYS, G. **PBRT, Version 4**. 2021. Available from Internet: <https://github.com/mmp/pbrt-v4>.

PHARR, M.; JAKOB, W.; HUMPHREYS, G. **Physically Based Rendering, from Theory to Implementation**. 3. ed. [S.l.]: Morgan Kaufmann, 2016.

TORRANCE, K. E.; SPARROW, E. M. Theory for off-specular reflection from roughened surfaces$*$. **J. Opt. Soc. Am.**, OSA, v. 57, n. 9, p. 1105–1114, Sep 1967. Available from Internet: <http://www.osapublishing.org/abstract.cfm?URI=josa-57-9-1105>.

TRIMBLE. **SketchUp**. 2021. Available from Internet: <https://www.sketchup.com/>.

TROWBRIDGE, T. S.; REITZ, K. P. Average irregularity representation of a rough surface for ray reflection. **J. Opt. Soc. Am.**, OSA, v. 65, n. 5, p. 531–536, May 1975. Available from Internet: <http://www.osapublishing.org/abstract.cfm?URI=josa-65-5-531>.

VEACH, E. **Robust Monte Carlo Methods for Light Transport Simulation**. Thesis (PhD) — Stanford University, 1997.

VEACH, E.; GUIBAS, L. Bidirectional estimators for light transport. In: . [S.l.: s.n.], 1995.

WALD, I. **PBRT-Parser**. 2021. Available from Internet: <https://github.com/ingowald/pbrt-parser>.

WALD, I.; PARKER, S. G. Rtx accelerated ray tracing with optix. In: **ACM SIGGRAPH 2019 Courses**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGGRAPH '19). ISBN 9781450363075. Available from Internet: <https://doi.org/10.1145/3305366.3340297>.

WALTER, B. et al. Microfacet models for refraction through rough surfaces. In: . [S.l.: s.n.], 2007. p. 195–206.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, ACM, New York, NY, USA, v. 23, n. 6, p. 343–349, jun. 1980. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/358876.358882>.

# APPENDIX A — MATH CONCEPTS

There are some basic key math concepts that are crucial for understanding the techniques presented here. These concepts are all applicable to 3D space, and most of these concepts are also applicable to other dimensions. All formulas and explanation will focus on their 3-dimensional versions.

## A.1 Points

We need to represent places in our space. Points, represented by $(x, y, z)$, are infinitely small. The coordinates $x$, $y$ and $z$ are determined by the current coordinate space. The origin of a coordinate space can be represented by the point $(0, 0, 0)$. For any point, $x$, $y$ and $z$ are also their distance from the origin in each axis.

## A.2 Vectors

Vectors are represented with $\boldsymbol{v} = (x, y, z)$. They have significant differences from points. First, they are not a "place" in space: they have direction and magnitude, but vectors don't start or end in predetermined locations. A vector's magnitude (that can also be called length) can be represented by $\|\boldsymbol{v}\| = \sqrt{x^2 + y^2 + z^2}$.

Two vectors are considered orthogonal to each other if they have a 90° angle between them.

### A.2.1 Vector Math

Vectors can be added or subtracted easily, by adding or subtracting each individual component. Subtracting two points (also done by subtracting each component) also results in a vector, with length equal to the distance between them. Adding a vector to a point will result in another point, which can be visualized as the original point dislocated in the direction and length of the original vector.

### A.2.2 Normalization

A vector which has a magnitude of $1$ is called a normalized or unit vector. They are generally utilized when only direction information is required. To normalize a vector, we just need to divide each component by the vector's length. Normalization will preserve the original direction of the vector.

If two unit vectors are also orthogonal to each other they are considered orthonormal vectors.

### A.2.3 Dot Product

A dot product is a very important operation in computer graphics. A dot product between vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is defined by:

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\|\|\boldsymbol{b}\| \cos\theta. \tag{A.1}$$

The smaller angle between the two vectors is called $\theta$. As we can see, the result of a dot product is a scalar value. There is an easier way to compute the dot product: multiplying each correspondent axis and then adding everything together.

$$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}_x\boldsymbol{b}_x + \boldsymbol{a}_y\boldsymbol{b}_y + \boldsymbol{a}_z\boldsymbol{b}_z. \tag{A.2}$$

This is extremely efficient for a computer to do. For that reason, a dot product is even commonly used to compute an easy cosine of the angle between two vectors: normalize both vectors first (if needed) then multiply and add their components.

### A.2.4 Cross Product

The cross product can be represented as $\boldsymbol{a} \times \boldsymbol{b}$, and its result will be a vector that is orthogonal to both $\boldsymbol{a}$ and $\boldsymbol{b}$. Each component of the cross product can be defined as:

$$(\boldsymbol{a} \times \boldsymbol{b})_x = \boldsymbol{a}_y\boldsymbol{b}_z - \boldsymbol{a}_z\boldsymbol{b}_y, \tag{A.3}$$

$$(\boldsymbol{a} \times \boldsymbol{b})_y = \boldsymbol{a}_z\boldsymbol{b}_x - \boldsymbol{a}_x\boldsymbol{b}_z, \tag{A.4}$$

$$(\boldsymbol{a} \times \boldsymbol{b})_z = \boldsymbol{a}_x \boldsymbol{b}_y - \boldsymbol{a}_y \boldsymbol{b}_x, \tag{A.5}$$

which means that:

$$\|\boldsymbol{a} \times \boldsymbol{b}\| = \|\boldsymbol{a}\| \|\boldsymbol{b}\| |\sin \theta|. \tag{A.6}$$

### A.3 Triangle meshes

A single triangle can be defined by three points. Multiple triangles that are connected by common edges are called triangle meshes. Triangle meshes are the most traditional way of representing a 3D scene in computer graphics.

### A.4 Normals

A normal is a vector that has a 90° angle with a surface. Normals are extremely important for computing several things, including shading calculations and orientation checking.

The normal of a triangle can be easily computed with a cross product between the vectors defined by subtracting its points and normalizing.

### A.5 Bounding Boxes

Bounding boxes are generally used to represent boundaries of complex objects. The boxes serve as a simplification of the shape of the bounded object (or objects). Bounding boxes are useful for a various of reasons in computer graphics, including occlusion queries and ray tracing acceleration.

Bounding boxes have several representations. If it is aligned with the axes of the coordinate space it is called an axis-aligned bounding box (AABB). In this case it can be represented by only two points, with the drawback of less flexibility in approximating the underlying shape.

## A.6 Rays

A ray is part of a line that points to a direction infinitely from a starting point. Different than vectors, a ray is placed in space and has infinite magnitude. The points that belong to the ray can be represented by a function:

$$R(t) = \boldsymbol{O} + \boldsymbol{vt}, \tag{A.7}$$

where $\boldsymbol{O}$ is the point of origin and $\boldsymbol{v}$ is the direction vector.

# APPENDIX B — RADIOMETRY

Radiometry is a series of tools, metrics and concepts designed to describe light interactions. It comes from an abstraction of light as a particle, and as such, is not capable of describing some light properties. It is, however, easily extendable to include most of them: for example, it was connected to Maxwell's equations by Preisendorfer in 1965.

The understanding of radiometry was directly taken from PBRT, and as such, they imply the same assumptions here as they did there. Perhaps the most important one is energy conservation: at any non-emissive point scattering light, the amount of energy that goes out must not surpass the energy that goes in.

This section will establish a basic glossary for radiometric quantities, and it is heavily based on PBRT's Radiometry section (PHARR; JAKOB; HUMPHREYS, 2016).

## B.1 Energy

A photon is a particle that is emitted by light sources. It resides in a particular wavelength, and carries an amount of energy. The energy of a photon at wavelength $\lambda$ is measured in joules (J):

$$Q = \frac{hc}{\lambda}, \tag{B.1}$$

where $c$ is the speed of light in the vacuum ($299,472,458 m/s$) and $h$ is Planck's constant $\approx 6.626 * 10^{-34} m^2 kg/s$.

## B.2 Flux

Radiant Flux (or Power) is the total energy that reaches a region per unit of time (t). It is measures in watts (W) or joules per second and can be calculated by:

$$\Phi = \lim_{\Delta t \to 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt}. \tag{B.2}$$

Flux is used to describe total emission from light sources.

### B.3 Irradiance and Radiant Exitance

A measurement of the density of flux that reaches a certain area A is called irradiance (if the flux is arriving at the surface) or radiant exitance (if the flux is leaving the surface). It is measured with $W/m^2$, and can be calculated by:

$$E = \frac{\Phi}{A}. \tag{B.3}$$

### B.4 Intensity

Intensity is the measurement of flux per solid angle. It is measured in watts per steradians ($W/sr$):

$$I = \lim_{\Delta\omega \to 0} \frac{\Delta\Phi}{\Delta\omega} = \frac{d\Phi}{d\omega}. \tag{B.4}$$

### B.5 Radiance

Finally, radiance is the flux density per unit area per solid angle. That means, intuitively, that radiance is different than irradiance by also distinguishing the directional distribution of power:

$$L(p, \omega) = \lim_{\Delta\omega \to 0} \frac{\Delta E_\omega(p)}{\Delta\omega} = \frac{dE_\omega(p)}{d\omega}, \tag{B.5}$$

where $E_\omega$ is the irradiance at a surface that is perpendicular to the direction $\omega$.