UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

Real-Time Rendering of Translucent Materials with Directional Subsurface Scattering

RELATORE: Emanuele Menegatti

CORRELATORE: Jeppe Revall Frisvad

LAUREANDO: Alessandro Dal Corso

A.A. 2013-2014

Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Via Gradenigo, 6/B,
35131 - Padova, Italy
Phone +45 4525 3351
+39-049-827-7600
www.dei.unipd.it

見ぬが花。

Not seeing is a flower.
(Japanese proverb)

Summary

The goal of this thesis is to provide a fast rendering technique to render translucent materials. The method we developed has low memory requirements, does not require a mesh UV mapping and requires little or not pre-processing. The method is well suited for real-time rendering applications such as computer games and digital interactive visualization.

We will employ new BSSRDF analytical model that considers the directionality of the incoming light into account. This additional parameter introduces some interesting challenges that are not easily dealt with by previous approaches.

The result is built by using a special sampling pattern based on the optical properties of the material. Our method incrementally builds the result over a certain number of frames, rendering the model from different directions and storing it in a texture. The texture is then sampled using shadow mapping in order to obtain the final rendering.

Using this approach, we obtained real-time results of 30 FPS for complex models of the magnitude normally employed in the computer game industry (10^4 triangles). The results we generate are close in appearance to a path traced solution. Our method then provides a fast and robust way to account for the direction of the incoming light in the computation, providing a more realistic results that the ones reachable with previous analytical models.

Preface

This thesis was prepared at the DTU Compute department at the Technical University of Denmark in fulfillment of the requirements for acquiring an M.Sc. in Digital Media Engineering.

The thesis deals with the efficient rendering of translucent materials, using an innovative model proposed by the author's M.Sc. thesis supervisor, Jeppe Revall Frisvad. Translucent materials consist of a particular class of materials like fruit, marble, skin, and other materials where subsurface scattering effects cannot be neglected.

The interest for real time rendering in the author arose during the course of his M.Sc. in Digital Media Engineering, where he focused on the study line in Computer Games. For this study line, he had to take several courses in real time computer graphics, and from this courses he got his interest in advanced shading techniques. In the spring 2014, professor Jeppe Revall Frisvad of DTU Compute proposed a research-oriented M.Sc. thesis, that had the final goal of creating a method for implementing the directional dipole in real time. The author deemed the topic to be a great opportunity to research in real time rendering techniques, and so he registered his application for this master thesis, *Real-Time Rendering of Translucent Materials with Directional Subsurface Scattering.*

The thesis consists of a software implementation in C++, Qt and OpenGL, and this report. The initial Qt framework used was taken from DTU course 02564, *Real Time Graphics*, and then expanded in order to fit the needs of the thesis. All the code reported in this document was written by the author an does not

come from the original framework. All the screenshots in this document were generated using the developed software. Other images in the report, unless the original source is reported in the caption, were generated by the author.

Lyngby, 03-July-2014

Alessandro Dal Corso

Acknowledgements

I would like to thank my supervisor for this MSc project, Jeppe Revall Frisvad, for the constant support and help during the whole duration of this thesis. His constant help and meetings in his office, as well as his prompt replies to my e-mails, have been determinant in the development of this thesis. I would also like to thank professor Neils Jørgen Christiansen for his suggestions and support during the weekly group meetings.

A special acknowledgment should be given to the T.I.M.E. double degree program, for giving me the opportunity to fulfill my dream of completing a Master Degree abroad. Without them, all my achievements in the past two years would not have been possible. From the University of Padua, I would like to thank professor Maria Elena Valcher, for helping with my study plan and all the bureaucracy a double degree program requires, and professor Emanuele Menegatti, for accepting on being my advisor for my MSc defense in Italy.

Finally, on a personal note, I would like to thank my family for the constant support they give to me, supporting also in my decision to study abroad. It is really impossible to list all the people that helped and supported me during this master thesis, but I would like to thank them all for cheering me up during the most difficult moments of this thesis.

Contents

Chapter 1

Introduction

Subsurface scattering (SS) is a physical phenomenon that naturally occurs in a wide range of natural materials. Some materials that exhibit a strong subsurface scattering effect in everyday life are milk, human skin and marble. Subsurface scattering occurs when light is partially absorbed by an medium, bounces repeatedly inside ("scatters") and finally exits the surface on another point of the material (as in Figure 1.1). The phenomenon that results is generally known as *translucency*. We can see some examples of translucent objects in Figure 1.2.

1.1 Background

Since the beginning of computer graphics, various attempts have been performed in order to model subsurface scattering. Some of these models involve Monte Carlo simulations of the light entering the medium [Pharr and Hanrahan, 2000], or other numerical techniques Fattal [2009], Kaplanyan and Dachsbacher [2010]. Other focus on approximating the diffusion of light within the material using an analytical approach, like [Jensen et al., 2001].

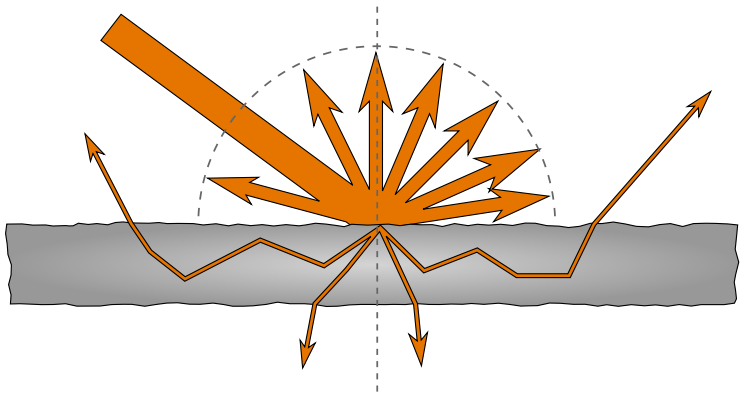


Figure 1.1: Diagram of subsurface scattering. Most of the incoming light gets reflected, but some of it enters the material and leaves it at a different point.

The first model that proposed an analytical approach was the one by Jensen et al. [2001], as an approximation of the radiative transfer equation. This approximation, called the *diffusion approximation* [Ishimaru, 1997] has been exploited by different authors, in order to account for multi-layered materials [Donner and Jensen, 2005], heterogeneous materials [Wang et al., 2010] and thin surfaces[Wang et al., 2010]. A recent analytical model, proposed by Frisvad et al. [2014], extends the approximation in order to account for the directionality of the incoming light. All these analytical methods are based on BSSRDF models. A BSSRDF function is a functions that describes how light is transmitted between two points in a material, and is a generally dependent on the incoming light direction, the distance between the two points and the outgoing light direction.

In recent years, with the advent of programmable graphics cards (GPU), it has become possible to exploit these algorithms and bring them to interactive frame rates, and in some cases even to real time rendering. Jensen and Buhler [2002] were the first to propose an efficient implementation for rendering subsurface scattering using an octree. More recently, several methods have been proposed, including image-based splats [Shah et al., 2009], sum-of-Gaussians filtering [d'Eon et al., 2007], and grid-propagation based methods [Børlum et al., 2011]. We will introduce in detail some of these methods in Chapter 2, were we will review the existing literature in more detail.

Figure 1.2: Some examples of translucent materials: marble, leaves and wax. The marble image and the candle are under the cc-by-sa license, and are courtesy of Wikimedia Commons. The two images were cropped to fit the document. The leaves image was taken by Alberto Bedin and is used with permission.

1.2 Problem statement

The goal of this thesis is to implement a real-time rendering technique in order to render directional subsurface scattering using the analytical model proposed by Frisvad et al. [2014]. The technique should ideally obtain the same results as a offline rendered solution of the original model, but reducing the rendering times to a few milliseconds. To do this, we employ the aid of the GPU programmable pipeline [Fernando, 2004].

We found that there is a current gap in the knowledge on current real-time subsurface scattering techniques regarding the approach to directional models. In fact, most of the methods rely on the assumption that the BSSRDF function depends only on the distance between the entering and exiting point Jensen et al. [2001]. This allows, for example, to pre-compute the BSSRDF function and use it in the computations, greatly increasing the performance [Shah et al., 2009]. However, in the model proposed by Frisvad et al. [2014], this is not possible, as the direction of the incoming light must be taken into account. In fact, the model has too many degrees of freedom to make a pre-computation feasible.

The model proposed by Frisvad et al. [2014] offers a more realistic evaluation of subsurface scattering effects. A real-time working implementation would improve the quality of scattering materials in real-time graphics applications, such as real-time architectural visualization and computer games. In the latter field, in recent years there has been a renewed interest in real-time SS techniques, especially to model faithfully the appearance of skin on human faces.

1.3 Requirement analysis

In this section, we will introduce some constraints and assumptions to limit the scope of our work. Some of these assumptions and constraints are well known to the graphics community, and they are generally introduced to allow better performance, quality and flexibility. Being a real-time rendering method implies that performance plays a big part in the decisions we have made in the process, but since the method uses a physically based approximation the final quality of the result is also important. In the process the aspect of flexibility has been taken into account, i.e. the capacity of the method to set the tradeoff between quality and performance. We will now list the assumptions we made in all the three described domains, quality, performance and flexibility.

1.3.1 Quality constraints

1. Be visually close as much as possible to a offline rendered solution.
2. Be consistent with the directional dipole model for a wide range of material properties. In particular, the method should perform well in the domain of quality where the directional dipole model excels (highly scattering materials).
3. Be potentially able to render an object under an arbitrary number of different types of lights (point, directional, environment, etc.).

1.3.2 Flexibility requirements

1. Work with the less amount as possible of provided model data, i.e. only the position data and eventually the normals should be provided in order for the method to run. In particular, no unwrap of the mesh (UV mapping) should be necessary.
2. Being able to be integrated in a game engine environment, using data from other computations (e.g. other lighting computations) and being adaptable to different lighting paths (forward and deferred shading).
3. The quality versus performance tradeoff should be set by a potential artist or developer, with the fewest number of parameters as possible.

1.3.3 Performance requirements

1. Being real-time on a high-end modern GPU, i.e. one frame should take less that 100 ms (10 FPS) to render. The ideal result would be to reach a rendering time of less than 16 ms (60 FPS).
2. Being as less dependent as possible from the geometrical complexity of the model.
3. Being as less dependent as possible from the screen resolution.
4. If the desired quality is not reachable within one frame, converge towards a result in a reasonable amount of time. Techniques should be used to approximate the required quality for the intermediate result.
5. Maintain a reasonable performance under changing light conditions, deformations and change of parameters, with little or none performance penalties.
6. Employ the advantages of the directional dipole model to improve performance.

7. Support up to a certain number of directional and point lights (up to 3 to 5 pixel lights, as in commercial engines[Unity, 2012]).
8. Require little or no pre-processing in order to be able to perform. If there is any pre-processing involved, it should be performed only at the beginning of the life cycle of the program.

1.4 Thesis Outline

In this Chapter, we have given an introduction to the problem and stated the assumption that will guide us through the choices that we will make through our thesis. In Chapter 2, Related Work, we will describe in more detail some of the different approaches to subsurface scattering in literature. In Chapter 3, we will give a theoretical introduction to subsurface scattering and light transport theory, with a special focus on BSSRDF functions. In Chapter 4, we will describe our method on approaching the problem on a theoretical basis. In Chapter 5 we will describe the actual implementation of the method, and the problems and limitations met during the process. In Chapter 6, we will describe the tests we made and show the results, both in the domain of performance and quality, comparing them with the requirement analysis we made in the previous section. Then, we will describe some possible extensions to the method in Chapter 7. We will wrap up everything in Chapter 8, where we will give our conclusions.

Chapter 2

Related Work

In rendering of subsurface scattering, most approaches rely on approximating the *Radiative Transfer Equation* (RTE). We identified two main approaches to the problem in literature:

Analytical One class of solutions consists of approximating the RTE or one of its approximations via an analytical model. These models can have different levels of complexity and computation times, and are often adaptable to a wide range of materials. However, often they rely on assumptions on the scattering parameters that limit their applicability.

Numerical In this other class of solutions, a numerical approach is used instead of approximating the RTE with an analytical model. This methods include finite element methods and discrete ordinate methods, for which a numerical solution for the RTE is actually computed. While providing an exact solution, the computation times are longer. Other numerical approaches focus more on the appearance of the model and do not provide an exact solution for the RTE.

In this thesis, we focus on efficiently implementing a model that falls in the first category, the analytical models. In the following sections, we are going to describe approaches for each one of the mentioned categories, comparing them to our method.

2.1 Analytical techniques

In the analytical techniques, two different areas of research must be distinguished. The first area is the research on the actual models, while the second is research on how the actual models can be implemented efficiently. Each model is usually represented by a specific function called BSSRDF (*Bidirectional Subsurface Scattering Reflectance Distribution Function*), that describes how light propagates between two points on the surface. Two integrations, one on the surface and one from all the directions, must be performed in order to get the amount of light that actually exits from a point on the surface (see chapter 3). Implementation techniques focus on efficiently implementing this integration steps, often making assumptions for which points computations can be avoided.

2.1.1 Models

Regarding the models, the first and most important is the dipole developed by Jensen et al. [2001]. This models relies on an approximation of the RTE called the *diffusion approximation*, which again relies on the assumption of highly scattering materials. In this case, a BSSRDF for a planar surface in a semi-infinite medium can be obtained. The BSSRDF needs only the distance between two points to be calculated, and with some precautions it can be also used with arbitrary geometry. This model does not include any single scattering term: it needs to be evaluated separately. The model has been further extended in order to account for thin object regions and multi-layered materials [Donner and Jensen, 2005].

A significant improvement of the model was later given by D'Eon [2012], that improved the model to fit path traced simulations. The new model can be evaluated without nearly any additional computation cost. A more advanced model based on quantization was proposed by D'Eon and Irving [2011], that introduced a new physical foundation in order to improve the accuracy of the original diffusion approximation. Finally, some higher order approximations exist [Menon et al., 2005, Frisvad et al., 2014], in order to account for the directionality of the incoming light and single scattering. This allows a more faithful representation of the model at the price of extended computation times. A comparison between the directional and the standard dipole can be seen in Figure 3.11.

2.1.2 Implementations

Most research on efficient implementations of a subsurface scattering analytical model has been made on the original model by Jensen et al. [2001]. The first efficient implementation was proposed by Jensen and Buhler [2002], based on a two-pass hierarchical integration approach. Samples on the model are organized in an octree data structure, that then is used to render the object. In the first step, the radiance from the light is stored in the points. In the second pass, using the octree, the contribution from neighboring points is computed, clustering far points in order to speed up calculations. This approach can be adopted for the Jensen model, where the only parameter is the distance between the entering and exiting point. However, using the directional dipole, the samples cannot be clustered because of the directionality of the light: once we sum up the contribution from multiple lights, the contribution cannot be separated anymore. In fact, we would need a different clustering of the points for each light, that quickly becomes inefficient since whole octree would have to fit the GPU's limited memory.

Lensch et al. [2002] approached the problem by subdividing the subsurface scattering contribution into two, a direct illumination part and a global illumination part (i.e. the light shining through the object). The global illumination part is pre-computed as vertex-to-vertex throughput and then summed to the direct illumination term in real-time. Compared to our method, this method requires a pre-computation step that depends on the geometry of the model, and thus deformation effects are not possible. Moreover, a coefficient has to be stored for each couple of vertices, that means a quadratic increase in memory for linearly increasing model size. Our method, on the other hand, occupies a memory space that depends linearly on the number of vertices.

Translucent shadow maps [Dachsbacher and Stamminger, 2003] use an approach similar to standard shadow maps: they render the scene from the light point of view, and then calculate the dipole contribution in one point only from a selected set of points, according to a specified sampling pattern. As in Lensch et al. [2002], the contribution is split into global and local to permit faster computations. In our approach we will reuse some of the ideas introduced by translucent shadow maps: we will render the scene from the light point of view and we will reuse the information stored in the map such as depth, vertices and normals. However, our approach to using the values from the map is different from the original paper, as we will explain in chapter 4. Mertens et al. [2003b] propose a fast technique based on radiosity hierarchical integration techniques, that unlike the previous implementation can handle deformable geometry.

Another important family of methods is screen space techniques. Mertens et al.

[2003a] propose an image space GPU technique that pre-computes a set of sample points for the area integration and then performs the integral over multiple GPU passes. d'Eon et al. [2007], Eugene and David [2007] propose a method in image-space, interpreting subsurface scattering as a sum of images blurred with a gaussian filter. The gaussians are then weighted to fit the diffusion approximation. Jimenez et al. [2009] improves further the technique, giving more precise results in the case of skin. All these techniques assume that the diffusion profile can be precomputed and then fitted with a sum of gaussians: as we have already mentioned, this is not possible for the directional dipole, where the diffusion profile varies depending on the angle of incidence of the incoming ray of light. Moreover, even if we were able to compute the coefficients for each possible combination of parameters, it would not be possible to apply a gaussian filter with a kernel that varies per pixel.

Shah et al. [2009] present a fast screen space technique that render the object as a series of splats, using GPU blending to sum over the various contributions. The diffusion profile in this case is pre-computed and stored as a texture. As in the previous techniques, the directionality of the incoming light does not allow the pre-computation of a diffusion profile. Moreover, the directional dipole is not symmetrical, so the splats would have to use a bigger radius in order to account for all the contribution, increasing computation and blending costs.

2.2 Numerical techniques

Numerical techniques for subsurface scattering are often not specific, but come for free or as an extension of a global illumination numerical approximation, since the governing equations are essentially the same. Given their generality, they are usually slower than their analytical counterpart, and often rely on heavy pre-computation steps in order to achieve interactive framerates. The volumetric version of Jensen's Photon Mapping[Jensen and Christensen, 1998] was originally developed to render participating media in general, but it has been adapted for subsurface scattering[Dorsey et al., 1999]. Classical approaches as a full Monte-Carlo simulation implementation of the light-material interaction, and finite-difference methods exist in literature[Stam, 1995].

Some less general methods have been introduced in order to devise more efficient approximations when it comes to subsurface scattering. Stam [1995] uses the diffusion approximation with the finite difference method on the object discretized on a 3D grid. Fattal [2009] uses as well a 3D grid, that is swept with a structure called light propagation map, that stores the intermediate results until the simulation is complete. All the numerical methods described so far are

not real-time, and they are generally not well-suited for a GPU environment.

Wang et al. [2010], instead of performing the simulation on a discretized 3D grid, makes the propagation directly in the mesh, converting it into a connected grid of tetrahedrons called *QuadGraph*. This grid can be optimized to be GPU cache friendly, and provides a real-time rendering of not-deformable heterogeneous objects. The problem in this method is that the QuadGraph is slow to compute (20 minutes for very complex meshes) and has heavy memory requirements for the GPU. Compared to our method, this one requires an heavy pre-computation step, and allows only not-deformable objects. However, as most of propagation techniques, it can handle heterogeneous materials, while our method can not.

Precomputed radiance transfer methods are another class of general global illumination methods. These methods pre-compute part of the lighting and store it in tables[Donner et al., 2009], allowing to retrieve it efficiently with an additional memory cost. The problem with this methods compared to ours is that its memory requirements increase exponentially if we want to handle deformable materials and changing light conditions. Moreover, it requires an heavy pre-computation in order to calculate the lighting coefficients. Our method, being analytical, does not required either a lot of memory or an heavy pre-computation step.

A recent method called SSLPV (*Subsurface Scattering Light Propagation Volumes*) [Børlum et al., 2011] extends a technique originally developed by Kaplanyan and Dachsbacher [2010] to propagate light efficiently in a scene using a set of discretized directions on a 3D grid. The method allows real-time execution times and deformable meshes with no added pre-computation step, with the drawback of not being physically accurate. Moreover, the required memory space on the GPU is larger than the one required than our method, since the voxelization of the mesh must be stored.

Finally, for real-time critical applications (such as computer games), translucency is often estimated as a function of the thickness of the material, that is used to modify a lambertian term [Tomaszewska and Stefanowski, 2012, Green, 2004]. The thickness is usually evaluated by sampling a depth map. A method by Kosaka et al. [2012] uses an approach similar to the one we will describe in order to compute the thickness of the material using a different camera direction. While not physically accurate, this techniques allows to have a fast translucency effect that can be easily added to existing deferred pipelines. Compared to our method, this method requires no storage space and light computation. However, the translucency effects are not represented faithfully, and some artifacts may appear, as pointed out in Green [2004], and multi-sampling should be used in order to avoid artifacts.

As we can see, in the reviewed literature so far there is not a proper way to account for direction in subsurface scattering in real-time, or at least one that satisfies the requirements we have made in chapter 1. We will introduce in detail our method to handle directional subsurface scattering in real-time in chapter 4. In the next chapter, we are going to give a theoretical introduction to a mathematical description of light transport, as well as giving the proper formulas and definition of the standard dipole model by Jensen et al. [2001] and the directional dipole model presented by Frisvad et al. [2014].

Chapter 3

Theory

In this chapter, we give a theoretical introduction to the topic dealt with in this thesis. The ultimate goal of this chapter is to introduce and describe analytical models for subsurface scattering. First, we will give a brief introduction to the nature of light, and how we physically describe it. Secondly, we will introduce the basic radiometric quantities that will be used throughout the chapter. Then, we will describe how this quantities are related and can be used to describe light-material interaction, using reflectance functions, of which BSSRDF functions are a special case. Finally, we will introduce subsurface scattering and the diffusion approximation, concluding with a description of two BSSRDF models, by Jensen et al. [2001] and Frisvad et al. [2014].

Quantity	Description
$\mathbf{x}$	Point, not normalized vector
(x_x, x_y, x_z)	Components of a vector
$\vec{\omega}$	Normalized vector, direction
Q	Radiant Energy
Φ	Radiant Flux
E	Irradiance
I	Intensity
L	Radiance
$\vec{n} \cdot \vec{m}$	Dot product between two vectors
$\vec{n} \times \vec{m}$	Cross product between two vectors
A	Area
i (subscript)	Denotes incoming direction or incidence point
o (subscript)	Denotes outgoing direction or exitance point
M (capital letter)	Denotes matrix (see Appendix A)
$f(...)$	BRDF function
$S(...)$	BSSRDF function
$\eta = \frac{n_1}{n_2}$	Relative index of refraction
$R(\eta, \vec{\omega})$	Fresnel reflection term
$T(\eta, \vec{\omega})$	Fresnel transmission term
$\vec{\nabla} \cdot \vec{x}$	Directional derivative of vector $\vec{x}$
σ_a	Absorption coefficient
σ_s	Scattering coefficient
g	Mean cosine coefficient
$\sigma_t = \sigma_s + \sigma_a$	Extinction coefficient
$\sigma'_s = \sigma_s(1-g)$	Reduced scattering coefficient
$\sigma'_t = \sigma'_t + \sigma_a$	Reduced extinction coefficient
σ_{tr}	Transmission coefficient
α'	Reduced albedo
$C_\phi(\eta)$	Approximation of the scalar fluence Fresnel integral
$C_{\mathbf{E}}(\eta)$	Approximation of the vector irradiance Fresnel integral

Table 3.1: Table of the notation used in this thesis.

3.1 Light and Radiometry

Light is a form of electromagnetic radiation, that propagates through space as a sinusoidal wave. Usually by *light* we refer to *visible light*, the small part of the electromagnetic spectrum the human eye is sensible to (see Figure 3.1). This small window is between the 380 nm of infrared and 750 nm of ultraviolet light, but the precise boundaries may vary according to the environment and the observer. Instead explicitly noted, we will use the terms light and visible light interchangeably in this report.

The study of light is usually referred as *optics*. In computer aided image synthesis, we are interested in representing faithfully how visible light propagates how it interacts with the objects and the materials in a scene. In addition, we are interested in lighting effects that are noticeable at human scales (1 mm - 1 km), like subsurface scattering, absorption and emission phenomena. Optics studies more effects, like diffraction, interference and quantum effects, but we are not interested in representing them because for visible light they happen on a microscopic scale (1 nm - 1 µm).

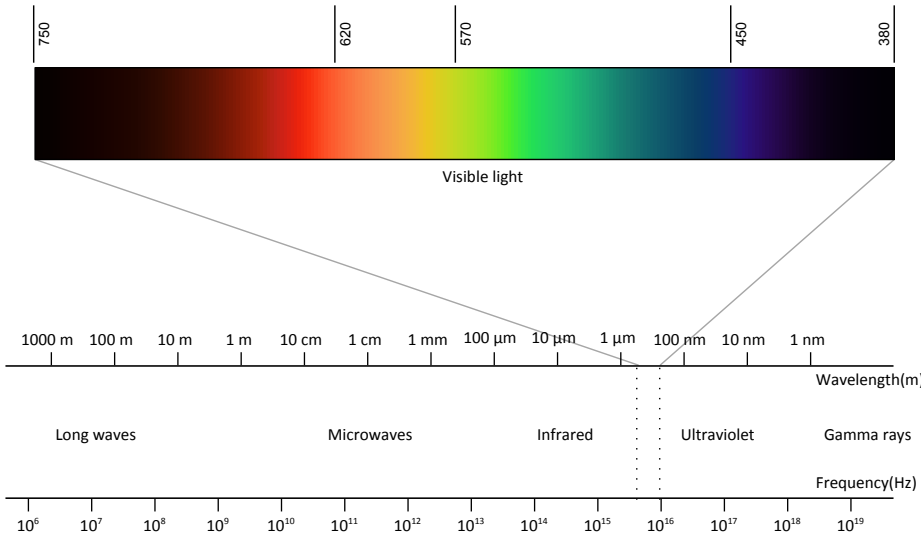


Figure 3.1: The electromagnetic spectrum.

The branch of physics that studies how to measure electromagnetic radiation is called *radiometry*. The energy of light, like all the others forms of energy, is measured in *Joules* [J = $\mathrm{kg\,m\,s^{-2}}$], and its power in *Watts* [W = $\mathrm{kg\,m\,s^{-3}}$]. *Photometry*, on the other hand, measures electromagnetic radiation as it is

perceived from the human eye, and limits itself only to the visible spectrum, while radiometry spans all of it. The corresponding names for energy and power in photometry are *radiant energy*, measured in *talbots* [cd s], and *radiant flux*, measured in *candelas* [cd].

In image synthesis it is more common to use radiometry, as its quantities directly derive from the electromagnetic theories, are universal, and can be easily converted to the photometric ones when necessary. The most important radiometric quantities used in computer graphics are *radiant flux*, *radiant energy*, *radiance*, *irradiance* and *intensity*.

3.2 Radiometric quantities

3.2.1 Radiant flux

The radiant flux, also known as radiant power, is the most basic quantity in radiometry. It is usually indicated with the letter Φ and it is measured in joules per seconds [$\mathrm{J\,s^{-1}}$] or Watts [W]. The quantity indicates how much power the light irradiates per unit time.

3.2.2 Radiant energy

Radiant energy, usually indicated as Q, is the energy that the light carries in a certain amount of time. Like all the other SI units for energy, it is measured in joules [J]. Radiant energy is obtained integrating the radiant flux along time for an interval ΔT:

$$Q = \int_{\Delta T} \Phi(t)\, dt$$

Due to the dual nature of the light, the energy carried by the light can be derived both considering light as made of particles, called *photons*, or considering it as a wave. We will not dig further into the topic, because for rendering purposes is not important if we characterize light as a flux of particles or as a sinusoidal wave.

3.2.3 Irradiance

Irradiance, usually defined as E, is the radiometric unit that measures the radiant flux per unit area *falling* on a surface. It is measured in Watts per square meter [$\mathrm{W\,m^{-2}}$]. It is defined as the flux per unit area:

$$E = \frac{d\Phi}{dA}$$

Irradiance is usually the term in literature used for the *incoming* power per unit area. The converse, i.e. the irradiance leaving a surface, it is usually referred as *radiant exitance* or *radiosity*, and indicated with the letter B.

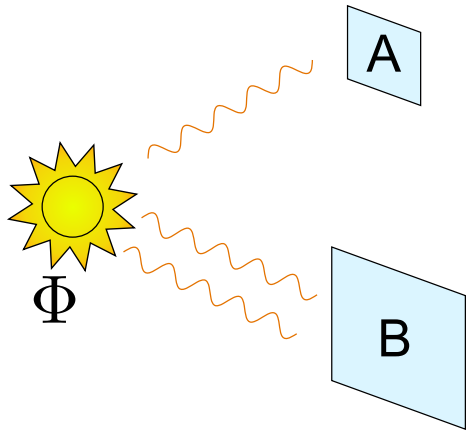


Figure 3.2: Irradiance versus power. For the two surfaces A and B, the received power Φ is the same, while the two irradiances E_A and E_B are different, as the area of B is twice as the one of A.

3.2.4 Intensity

Intensity is defined as the differential radiant flux per differential solid angle:

$$I(\vec{\omega}) = \frac{d\Phi}{d\omega} \tag{3.1}$$

It is measured in Watts per steradian [$\mathrm{W\,sr^{-1}}$] and it is indicated with the letter I. Intensity is often a misused term in the physics community, as it is used for many different quantities. Depending on the research community, intensity may refer to irradiance or even to radiance (see the following section). The definition given in 3.1 we use the most common definition given by the optics community.

3.2.5 Radiance

Radiance is arguably the most important quantity in image synthesis. It is defined precisely as the differential of the flux per solid angle per projected surface area, and it is measured in Watt per steradian per square meter [$\mathrm{W\,sr^{-1}\,m}$].

$$L(\vec{\omega}) = \frac{d^2\Phi}{d\omega dA\cos\theta}$$

Where θ is the angle between the surface normal and the incoming ray of light (so that $\cos\theta = \vec{n}\cdot\vec{\omega}_i$).

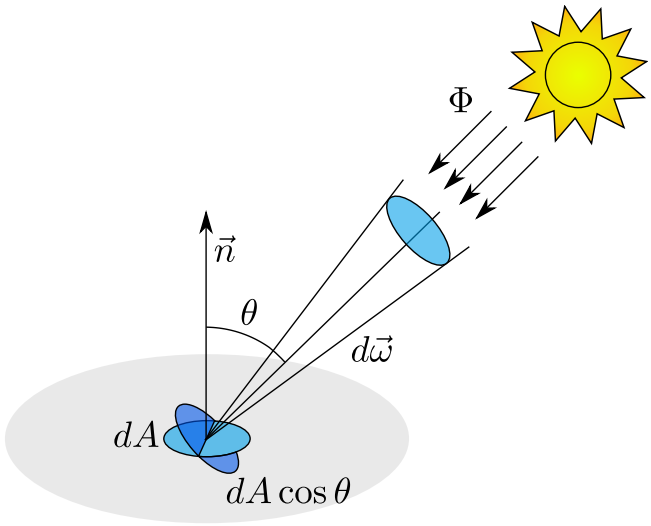


Figure 3.3: Radiance. The element of area dA gets projected according to the angle $\theta = \cos^{-1}\vec{n}\cdot\vec{\omega}$. Then the incoming flux Φ gets divided by the projected area and by the solid angle subtended by it.

Radiance has the important property of being constant along a ray of light. In addition, the sensibility of the human eye to light is directly proportional to the radiance. For a discussion on why radiance is related to the sensitivity of sensors and the human eye, see Cohen et al. [1993].

All the other radiometric quantities can be derived from radiance:

$$\begin{aligned} E &= \int_{2\pi} L_i(\vec{\omega}) \cos\theta \; d\omega \\ B &= \int_{2\pi} L_o(\vec{\omega}) \cos\theta \; d\omega \\ I(\vec{\omega}) &= \int_{A} L(\vec{\omega}) \cos\theta \; dA \\ \Phi &= \int_{A} \int_{2\pi} L(\vec{\omega}) \cos\theta \; d\omega dA \end{aligned} \tag{3.2}$$

For simplicity of notation, the dependence from the point of incidence $\mathbf{x}$ has been dropped in equations 3.2.

3.2.6 Radiometric quantities for simple lights

To help with the formulas used later in the report, we derive the standard radiometric quantities for the two simplest types of light, i.e. directional and point lights.

- *Directional lights* simulate very distant light sources, in which all the rays of light are parallel (e.g. sunlight). They are represented by a direction $\vec{\omega}_l$ and a constant radiance value, L.
- *Point lights* simulate lights closer to the observer. Isotropic point lights are represented by a position of the light $\mathbf{x}_l$ and a constant intensity I. Point lights have a falloff that depends on the inverse square law, i.e. the radiance diminishes with the square of the distance.

Table 3.2 shows different radiometric quantities evaluated for point and directional lights, for a surface point $\mathbf{x}$ with surface normal $\vec{n}$.

3.3 Reflectance Functions

After introducing the basic radiometric quantities, we still lack a way to describe light material interaction. More precisely, we need a way to relate the incoming and the outgoing radiance on a point of a chosen surface.

Quantity	Directional light	Point light
Cosine term	$\cos\theta = \vec{n} \cdot \vec{\omega}_l$	$\cos\theta = \frac{(\mathbf{x}-\mathbf{x}_l)\cdot\vec{n}}{\lvert\mathbf{x}-\mathbf{x}_l\rvert}$
$\Phi(\mathbf{x})$ Flux	$L\ \delta(\vec{\omega})$	$4\pi I$
$E(\mathbf{x})$ Irradiance	$L\cos\theta$	$I\frac{\cos\theta}{\lvert\mathbf{x}_l-\mathbf{x}\rvert^2}$
$I(\mathbf{x},\vec{\omega})$ Intensity	$L\ \delta(\vec{\omega})$	I
$L(\mathbf{x},\vec{\omega})$ Radiance	$L\ \delta(\vec{\omega})$	$\frac{I}{\lvert\mathbf{x}_l-\mathbf{x}\rvert^2}$

Table 3.2: Different radiometric values for simple light sources.

3.3.1 BRDF functions

One of the possible way to describe light-material interaction is by using a BDRF function [Nicodemus et al., 1992], acronym for *Bidirectional Reflectance Distribution Function*. The BRDF function $f(\mathbf{x},\vec{\omega}_i,\vec{\omega}_o)$ is defined on one point $\mathbf{x}$ of the surface as the differential ratio between the exiting radiance and the irradiance:

$$f(\mathbf{x},\vec{\omega}_i,\vec{\omega}_o) = \frac{dL_o(\mathbf{x},\vec{\omega}_o)}{dE_i(\mathbf{x},\vec{\omega}_i)} = \frac{dL_o(\mathbf{x},\vec{\omega}_o)}{L_i(\mathbf{x},\vec{\omega}_i)\cos\theta_i d\vec{\omega}_i} \tag{3.3}$$

The BRDF states that the incoming and the outgoing radiance are proportional, so that the energy hitting the material at the point $\mathbf{x}$ is proportional to the energy coming out from the point. BRDF functions have generally the following properties:

- *reciprocal*: for the Hemholtz reciprocity principle, a physics result that is also the basis of reverse path ray tracing [Desolneux et al., 2007]:

  $$f(\mathbf{x},\vec{\omega}_i,\vec{\omega}_o) = f(\mathbf{x},\vec{\omega}_o,\vec{\omega}_i)$$

- *anisotropic*: if the surface changes orientation and $\vec{\omega}_i$ and $\vec{\omega}_o$ stays the same, the resulting BRDFs are different. So generally

  $$f(\mathbf{x},\vec{\omega}_i,\vec{\omega}_o) \neq f(\mathbf{x},R\vec{\omega}_o,R\vec{\omega}_i)$$

  where R is a rotation matrix with arbitrary axis around the point $\mathbf{x}$.

- *positive*, since the BRDF regulates the transport between two positive quantities (radiance, irradiance).

  $$f(\mathbf{x},\vec{\omega}_o,\vec{\omega}_i) \geq 0$$

- *energy conserving*, so that the energy of the outgoing ray is no grater that the one of the incoming one

$$\int_{2\pi} f(\mathbf{x}, \vec{\omega}_o, \vec{\omega}_i) \cos\theta_o d\vec{\omega}_o \leq 1$$

By inverting equation 3.3, we obtain the so-called *reflectance equation*:

$$L_o(\mathbf{x}, \vec{\omega}_o) = \int_{2\pi} f(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i$$

Later we will use this equation as a starting point to obtain a formulation of the full rendering equation. The BRDF function has some limitations, being not able to account for all phenomena. For example, with a BRDF it is not possible to account for subsurface scattering, because it assumes the light enters and leaves the material at the same point. To model these phenomena, more complicated functions are needed, like the BSSRDF function described later in this chapter.

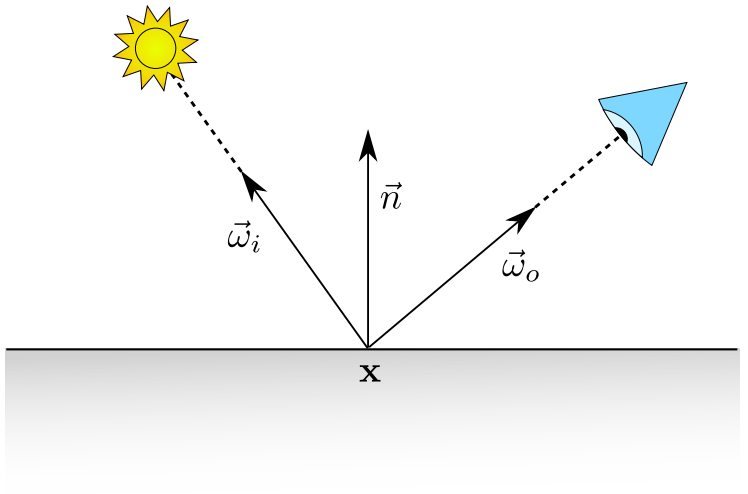
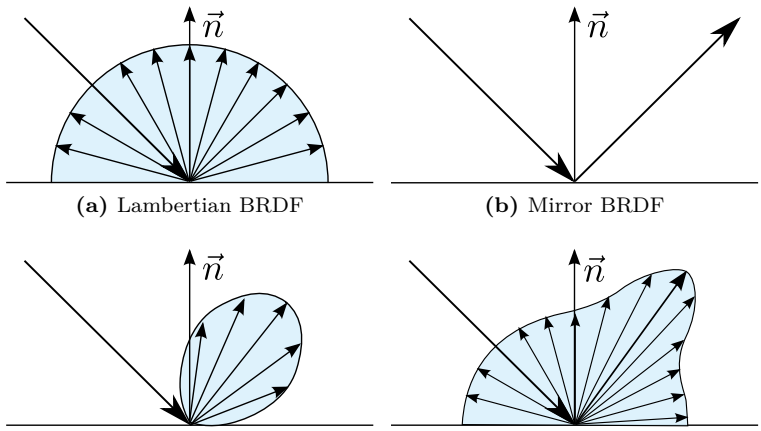


Figure 3.4: Setup for a BRDF. Note that the light enters and leaves the surface at the same point.

3.3.2 Examples of BRDF functions

There are many examples of BRDF functions in literature. In this section, in order to illustrate some examples, we will introduce three of them: the lambertian or diffuse BRDF, the specular or mirror BRDF and glossy BRDFs. For a detailed overview on BRDF functions, refer to [Akenine-Möller et al., 2008].



(a) Lambertian BRDF **(b)** Mirror BRDF

(c) Glossy BRDF **(d)** Combined BRDF

Figure 3.5: Examples of BRDF functions. In this particular example, the three simple BRDFs can be evaluated separately and then combined in the more complex BRDF of Figure 3.5d in order to represent multiple effects.

3.3.2.1 Lambertian BRDF

In the lambertian BRDF, the incoming radiance is distributed equally in all directions, regardless of the incoming direction. To do this, the BRDF must be constant:

$$f(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) = k_d$$

We can check that then the radiance is scattered equally in all directions by simple integration:

$$\begin{aligned} L_o(\mathbf{x}, \vec{\omega}_o) &= \int_{2\pi} f_d L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i \\ L_o(\mathbf{x}, \vec{\omega}_o) &= k_d \int_{2\pi} L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i \\ L_o(\mathbf{x}, \vec{\omega}_o) &= k_d \, E(\mathbf{x}) \end{aligned}$$

The lambertian model is an ideal model, so very few material exhibit a lambertian diffusion. Some of them are unfinished wood and *spectralon*, a synthetic material created in order to be as close as possible to a perfect lambertian material. Given its properties, spectralon is usually employed in calibrating radiance testing equipment.

3.3.2.2 Mirror BRDF

Another simple kind of BRDF is the perfectly specular BRDF, or mirror BRDF. In this function, all the incoming radiance from one direction $\vec{\omega}_i$ is transferred towards the reflected direction $\vec{\omega}_r$, defined as $\vec{\omega}_r = \vec{\omega}_i - 2(\vec{\omega}_i \cdot \vec{n})\vec{n}$. The resulting BRDF is defined as follows:

$$f(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) = \frac{\delta(\vec{\omega}_o - \vec{\omega}_r)}{\cos\theta_i}$$

The function $\delta(\vec{\omega})$ is a hemispheric delta function. Once integrated over a hemisphere, the function evaluates to one only for the vector $\vec{\omega} = \mathbf{0}$. Putting the BRDF into the reflectance equation gives the following outgoing radiance:

$$L_o(\mathbf{x}, \vec{\omega}_o) = \begin{cases} L_i(\mathbf{x}, \vec{\omega}_i) & \text{if } \vec{\omega}_o = \vec{\omega}_r \\ 0 & \text{otherwise} \end{cases}$$

that is the expected result, as all the radiance is reflected into the direction $\vec{\omega}_r$.

3.3.2.3 Glossy BRDFs

As we can see from real life experience, rarely objects are completely diffuse or completely specular. These two models are idealized models, that represent an ideal case. So, to create a realistic BRDF model, we often need to combine the two terms and add an additional one, called glossy reflection.

The most used BRDF model used to model glossy reflections is based on microfacet theory [Torrance and Sparrow, 1992, Ashikmin et al., 2000] and was first introduced by [Blinn, 1977]. In this theory, the surface of an object is modeled as composed of small mirrors. In one of its classical formulations, the BRDF is represented as:

$$f(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) = \frac{DGR}{4\cos\theta_r \cos\theta_i} = \frac{GR}{4}\frac{(\vec{n} \cdot \vec{h})^s}{(\vec{n} \cdot \vec{r})(\vec{n} \cdot \vec{\omega}_i)}$$

D regulates how microfacets are distributed, and it is often modeled as $(\vec{n} \cdot \vec{h})^s$, where $\vec{h}$ is the half vector between the eye and the light, and s is an attenuation parameter. $\vec{h}$ is defined as:

$$\vec{h} = \frac{\vec{\omega}_o + \vec{\omega}_i}{\|\vec{\omega}_o + \vec{\omega}_i\|}$$

G accounts for the object self shadowing, while R is the Fresnel reflection term (more details in Section 3.3.4). $\vec{r}$ is the reflection vector as defined in the previous

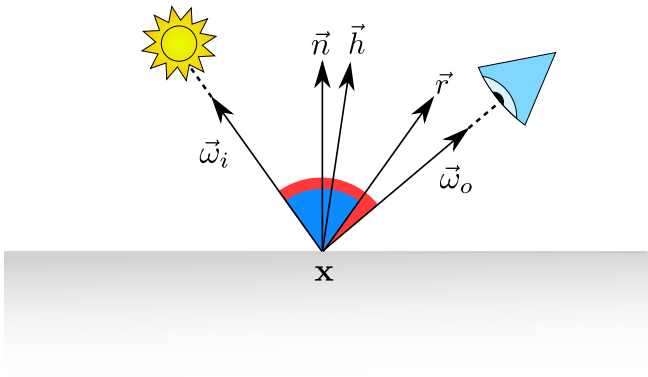


Figure 3.6: Glossy vectors for microfacet theory. The blue angles are the same for the reflection vector, the red ones are the same for the half vector.

section. See Figure 3.6 on how the vectors for the glossy reflection - $\vec{n}$, $\vec{h}$ and $\vec{r}$ - are defined.

Various alternative definitions exist for the D and G function, varying among the literature. Other glossy models not based on microfacet theory do exist as well [Akenine-Möller et al., 2008].

3.3.3 The rendering equation

Given the reflectance equation, it is possible to generalize it in order to model all the lighting in an environment (global illumination). In fact, the described reflectance equation is a suitable candidate to represent a full global illumination equation, but it does not account for two important factors.

The first factor are emissive surfaces. We need to add an emissive radiance term $L_e(\mathbf{x}, \vec{\omega})$ that models the amount of radiance that a point is emitting in a certain direction. This is useful to model light sources, without introducing a separate equation. We note that point lights have a singularity: they emit infinite radiance on the point where they are placed.

The second factor is that the reflectance equation accounts only for direct illumination. In general, we want to include also light that bounced onto another surface before reaching the current surface. To model this, we can replace the L_i term in the reflectance equation with another term L_r that accounts for light

coming from another surface. This term can be usually modeled as the product of the radiance of the light plus a visibility function $V(\mathbf{x})$.

Accounting for all the described factors, we reach one formulation of the rendering equation [Kajiya, 1986]:

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}) + \int_{2\pi} f(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\mathbf{x}, \vec{\omega}_i) V(\mathbf{x}) \cos\theta_i d\vec{\omega}_i$$

This form of the rendering equation is still not completely general, since it is based on a BRDF, to it comes with the same limitations (no subsurface scattering effects or wavelength-changing effects like iridescence). We will extend the rendering equation in order to account for these phenomena later on in this chapter.

3.3.4 Fresnel equations

Until now, on the described BRDF models, we did consider only the reflected part of the radiance. When a beam of light coming from direction $\vec{\omega}_i$ hits a surface, only part of the incoming radiance gets reflected, while another part gets refracted into the material. As we can see from Figure 3.7, we obtain the two vectors $\vec{\omega}_r$ and $\vec{\omega}_t$, the reflected and refracted vector, defined as follows [Kay and Greenberg, 1979]:

$$\vec{\omega}_r = \vec{\omega}_i - 2(\vec{\omega}_i \cdot \vec{n})\vec{n}$$
$$\vec{\omega}_t = \eta((\vec{\omega}_i \cdot \vec{n})\vec{n} - \vec{\omega}_i) - \vec{n}\sqrt{1 - \eta^2(1 - (\vec{\omega}_i \cdot \vec{n})^2)}$$

Where $\eta = \frac{n_1}{n_2}$ is the relative index of refraction between the two materials. With this setup, illustrated in Figure 3.7, we can use a solution to Maxwell's equations for wave propagation to describe the radiant flux. In particular, we can tell which part of the power propagates in the reflected and refracted direction respectively. The coefficients that describe this subdivision of the power are called *Fresnel coefficients* [Born and Emil, 1999]. The coefficients are different according to the polarization of the incoming light (parallel or perpendicular),

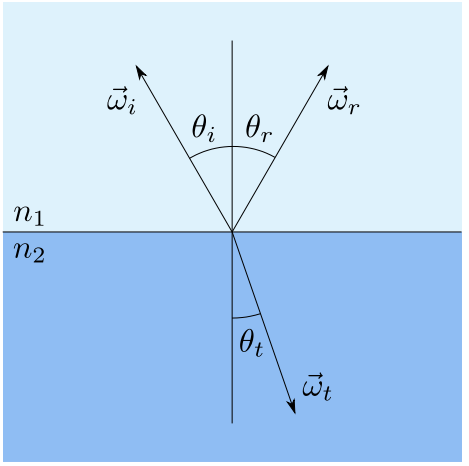


Figure 3.7: Reflected and refracted vector on mismatching indices of refraction.

so there are two for reflection (R_s, R_p) and two for transmission (T_s, T_p).

$$R_s(\eta, \vec{\omega}_i) = \left| \frac{\eta \cos\theta_i - \cos\theta_t}{\eta \cos\theta_i + \cos\theta_t} \right|^2$$

$$R_p(\eta, \vec{\omega}_i) = \left| \frac{\eta \cos\theta_t - \cos\theta_i}{\eta \cos\theta_t + \cos\theta_i} \right|^2$$

$$T_s(\eta, \vec{\omega}_i) = \eta \, \frac{\cos\theta_t}{\cos\theta_i} \left| \frac{2\cos\theta_i}{\eta \cos\theta_i + \cos\theta_t} \right|^2$$

$$T_p(\eta, \vec{\omega}_i) = \eta \, \frac{\cos\theta_t}{\cos\theta_i} \left| \frac{2\cos\theta_i}{\eta \cos\theta_t + \cos\theta_i} \right|^2$$

In most computer graphics applications (and this is reasonable for most of the real-world lights), we assume that the two polarizations are equally mixed. So, we will use the coefficient $R = \frac{R_s + R_p}{2}$ and $T = \frac{T_s + T_p}{2}$ in our calculations. Note that $R + T = 1$, so the overall energy is conserved.

3.3.5 BSSRDF functions and generalized rendering equation

As we anticipated in Section 3.3.1, the BRDF theory that was introduced before is not accurate in predicting the behavior for all materials, since BRDF models assume that the light enters and leaves the material in the same point. While this assumption holds true for a wide range of material, like metal or plastic, it

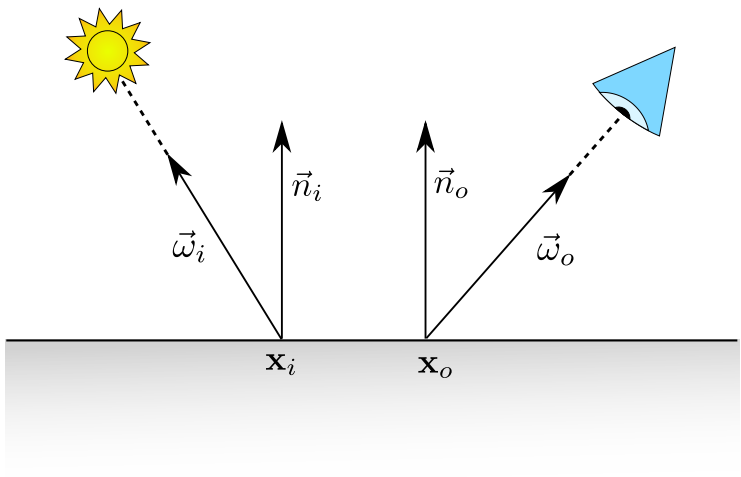


Figure 3.8: BSSRDF setup. As we compare it to the one of Figure 3.4, we can see that the light enters and leaves the surface at two different points.

poorly describes translucent materials, that exhibit a consistent amount of light transport under the surface.

In order to describe light transport in this material, we introduce a function, called BSSRDF [Nicodemus et al., 1992], acronym for *Bidirectional Subsurface Scattering Reflectance Distribution Function*. This function extends the concept of BRDF to account for two separate points. The BSSRDF is usually indicated with a capital S. We define the BRDF as the ratio between the incoming flux in a point $\mathbf{x}_i$ from the direction $\vec{\omega}_i$ and the outgoing radiance in *another* point $\mathbf{x}_o$ on direction $\vec{\omega}_o$:

$$S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) = \frac{dL_o(\mathbf{x}_o, \vec{\omega}_o)}{d\Phi_i(\mathbf{x}_i, \vec{\omega}_i)} = \frac{dL_o(\mathbf{x}_o, \vec{\omega}_o)}{dE_i(\mathbf{x}_i, \vec{\omega}_i)dA_i} = \frac{dL_o(\mathbf{x}_o, \vec{\omega}_o)}{L_i(\mathbf{x}_i, \vec{\omega}_i)\cos\theta_i d\vec{\omega}_i dA_i}$$

As we can see, the BSSRDF is similar to the BRDF, apart from a additional derivation in the area domain. Once we rearrange this equation, we can obtain an updated reflectance equation for the BSSRDF:

$$L_o(\mathbf{x}_o, \vec{\omega}_o) = \int_A \int_{2\pi} S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) L_i(\mathbf{x}_i, \vec{\omega}_i) \cos\theta_i d\vec{\omega}_i dA_i$$

We can immediately see that the new reflectance equation accounts for light scattering between two points, but this generality comes with a price. In fact, it adds a order of magnitude of complexity, since now the BSSRDF needs to be integrated twice, once on the whole surface and once on the normal hemisphere.

As we did for the BRDF, we can further extend the reflectance equation to further include visibility and emission, giving an extended form of the rendering equation [Jensen et al., 2001].

$$L_o(\mathbf{x}_o, \vec{\omega}_o) = L_e(\mathbf{x}_i, \vec{\omega}_i) + \int_A \int_{2\pi} S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) L_i(\mathbf{x}_i, \vec{\omega}_i) V(\mathbf{x}) (\vec{n} \cdot \vec{\omega}_i) d\vec{\omega}_i dA_i \tag{3.4}$$

From now on, by 'rendering equation' in this report we will mean the one in equation 3.4.

3.4 Light transport and subsurface scattering

When we derive our models for lighting, in general we assume that the light is traveling in vacuum. This assumption holds for light that is propagating though the air (which is assimilable to vacuum), but once we relax it, more variables should be taken into consideration. Objects through which light travels are referred as *participating media*. In this chapter, we will derive and consider an alternative formulation of the rendering equation for light traveling into participating media, called *radiative transfer equation* [Chandrasekar, 1950].

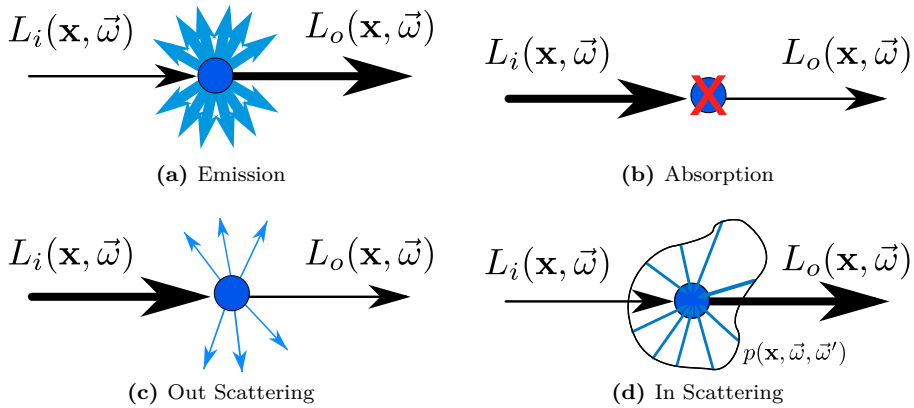
When a beam of light travels through an object, various phenomena occur. A photon on the beam can be either being absorbed (disappear), scattered (change direction) or emitted (appear). These phenomena can be uniform throughout the material (homogeneous materials), as in solid materials like wax or leaves, or be not uniform (heterogeneous materials), like in smoke or clouds.

We will briefly describe all three mentioned effects, then combine them to compose the radiative transfer equation. The purpose is to describe how radiance varies along a beam of light with direction $\vec{\omega}$. This directional derivative is indicated as:

$$(\vec{\nabla} \cdot \vec{\omega}) L(\mathbf{x}, \vec{\omega}) = \frac{\partial L_x}{\partial x} \vec{\omega}_x + \frac{\partial L_y}{\partial y} \vec{\omega}_y + \frac{\partial L_z}{\partial z} \vec{\omega}_z$$

3.4.1 Emission

Emission is the natural property of the materials to emit light, i.e. to generate photons that add to the existing ones passing through the material. The effect is generally generated by chemical processes emitting photons (as in fireflies), by natural black-body radiation emission in the visible spectrum (such as in a



(a) Emission **(b)** Absorption

(c) Out Scattering **(d)** In Scattering

star like the sun or in incandescent bulbs), or by other radiation that changes its wavelength into the visible spectrum.

In the directional 3D derivative, the variance in emission is modeled as a constant depending only on the current position and direction:

$$(\vec{\nabla} \cdot \vec{\omega}) L(\mathbf{x}, \vec{\omega}) = \epsilon(\mathbf{x}, \vec{\omega})$$

This means that emission increases linearly along the body: if the beam travels a distance d within the medium, $d \cdot k$ photons are emitted. Emission is generally isotropic, not depending on the direction ($\epsilon(\mathbf{x}, \vec{\omega}) = \epsilon(\mathbf{x})$).

3.4.2 Absorption

Absorption is a property of materials that describes a simple physical phenomenon: a photon, traveling though the material, hits one atom of the material. The energy carried by the photon is then absorbed by the atom, augmenting its kinetic energy. This directly translates in an increase of heat in the material. Usually, a certain percentage of the photons that hit the atoms is absorbed per unit length. Then, if k is the percentage of the photons absorbed in a meter, after one meter the original radiance will become $k \cdot L_i$, then $k^2 \cdot L_i$, etc.

If we write this phenomena as a differential equation, we get after a distance d a radiance reduction of $k^d = e^{-\sigma_a d}$, that leads to the following 3D directional derivative:

$$(\vec{\nabla} \cdot \vec{\omega}) L(\mathbf{x}, \vec{\omega}) = -\sigma_a(\mathbf{x}, \vec{\omega}) L(\mathbf{x}, \vec{\omega})$$

σ_a is referred as the *absorption coefficient*. Also this coefficient is generically isotropic, and constant for homogenous materials.

3.4.3 Out-scattering

Out scattering is the radiance lost due to scattering. The scattering phenomenon happens when photons are deflected away from the current direction $\vec{\omega}$. As in the previous case, the phenomena is modeled as a percentage of the radiance lost per unit length. So the loss due to out-scattering is modeled as:

$$(\vec{\nabla} \cdot \vec{\omega})L(\mathbf{x}, \vec{\omega}) = -\sigma_s(\mathbf{x}, \vec{\omega})L(\mathbf{x}, \vec{\omega})$$

σ_s is referred as the *scattering coefficient*. We note that in this case we are not interested in which direction the photons are actually going. That will be accounted in the in-scattering term of another point in the material.

3.4.4 In-scattering

Given some loss due to some of the photons changing direction, there will be some of them that from other scattering events will change to the $\vec{\omega}$ direction. We need then to discover the number of photons that comes from all the other directions. To do this, we integrate the incoming radiance from all directions in the point $\mathbf{x}$. This quantity, similar to irradiance, in an infinite medium is called *fluence*, and indicated as ϕ:

$$\phi(\mathbf{x}) = \int_{4\pi} L(\mathbf{x}, \vec{\omega}')d\omega'$$

Fluence should be then averaged over the entire sphere, yielding $\frac{\phi}{4\pi}$ as a normalization factor. This quantity then is then multiplied by the scattering coefficient, because only some photons on average scatter towards the current point. This results in:

$$(\vec{\nabla} \cdot \vec{\omega})L(\mathbf{x}, \vec{\omega}) = \sigma_s(\mathbf{x})\frac{1}{4\pi}\int_{4\pi} L(\mathbf{x}, \vec{\omega}')d\omega' \tag{3.5}$$

However, equation 3.5 assumes that radiance scatters equally in all directions. This is not usually the case, and the $\frac{1}{4\pi}$ term needs to be replaced by a probability distribution function that describes how the photons scatter in the medium. This function is called *phase function*, and indicated as $p(\mathbf{x}, \vec{\omega}, \vec{\omega}')$. In the actual models its integral on the hemisphere is often used as a parameter, called *mean cosine* (g):

$$g(\mathbf{x}) = \int_{4\pi} p(\mathbf{x}, \vec{\omega}, \vec{\omega}')\vec{\omega} \cdot \vec{\omega}' d\omega'$$

This term indicates the general direction of the scattering in the material. If positive, the scattering is prevalent along the beam (forward scattering), if negative is prevalent in the opposite direction (backward scattering). If zero, the scattering is isotropic, i.e. equal in all directions.

So, the final 3D equation for in-scattering, accounting for the phase function, is as follows:

$$(\vec{\nabla} \cdot \vec{\omega})L(\mathbf{x}, \vec{\omega}) = \sigma_s(\mathbf{x}) \int_{4\pi} p(\mathbf{x}, \vec{\omega}, \vec{\omega}')L(\mathbf{x}, \vec{\omega}')d\omega'$$

3.4.5 Final formulation of the radiative transfer equation

Combining emission, absorption, scattering described in the previous sections, we reach the final formulation of the radiative transfer equation (RTE):

$$(\vec{\nabla} \cdot \vec{\omega})L(\mathbf{x}, \vec{\omega}) = -\sigma_t(\mathbf{x})L(\mathbf{x}, \vec{\omega}) + \epsilon(\mathbf{x}) + \sigma_s(\mathbf{x}) \int_{4\pi} p(\mathbf{x}, \vec{\omega}, \vec{\omega}')L(\mathbf{x}, \vec{\omega}')d\omega' \quad (3.6)$$

Where the two reducing term, scattering and absorption, have been combined together in $\sigma_t = \sigma_a + \sigma_s$, called the *extinction coefficient*.

3.4.6 The diffusion approximation

The radiative transfer equation 3.6 is a integro-differential equation with many degrees of freedom. As we stated in Chapter 2, there are rendering techniques that numerically solve the equation in order to obtain a realistic result. However, analytical methods tend to use some approximations of the RTE, that hold well given specific conditions. The *diffusion approximation* [Ishimaru, 1997] is one of these approximations, and it is still widely used today since its introduction in the computer graphics community by [Stam, 1995].

The assumption under the diffusion approximation is that given a physical medium, the number of scattering events is so high that the beam of light quickly becomes isotropic. Each one of the scattering events blurs the light distribution, and as a result the distribution becomes more uniform as the number of scattering events increases. This has been proven to be a reasonable assumption even for highly anisotropic light sources (e.g. a focused laser beam) and phase functions.

When using the diffusion approximation, instead of using the extinction coefficient σ_t, we account for the contribution from the phase function by using the

so-called *reduced extinction coefficient* σ_t'. It is defined as $\sigma_t' = \sigma_a + \sigma_s'$, with $\sigma_s' = \sigma_s(1-g)$. σ_s' Is called *reduced scattering coefficient*. The converse of the reduced extinction coefficient is called *mean free path* and represents the average distance that light travels in the medium before being absorbed or scattered.

The rationale behind this reduced coefficient is that a highly forward scattering material is virtually indistinguishable from a not-scattering material. So, for highly forward scattering materials ($g \approx 1$) the scattering coefficients reduce to zero. For highly backward scattering materials ($g \approx -1$), the scattering is accounted for twice as for an isotropic material (see table 3.3).

Coefficient	Backward Scattering ($g \approx -1$)	Isotropic ($g \approx 0$)	Forward Scattering ($g \approx 1$)
σ_s'	$2\sigma_s$	σ_s	0
σ_t'	$\sigma_a + 2\sigma_s$	$\sigma_a + \sigma_s$	σ_a

Table 3.3: Explicit scattering coefficients for different kinds of materials.

We leave to Ishimaru [1997] and Jensen et al. [2001] for the algebraic details of the calculation. Once we solve the diffusion equation, we obtain the following formula for $\phi(\mathbf{x})$, the fluence of light in an infinite scattering medium.

$$\phi(\mathbf{x}) = \frac{\Phi}{4\pi D} \frac{e^{\sigma_{tr} r}}{r} \tag{3.7}$$

We recall that $\phi(\mathbf{x}) = \int_{4\pi} L(\mathbf{x}, \vec{\omega}) d\vec{\omega}$. $r = \|\mathbf{x}\|$ is the distance from the point to the light source. The two coefficients D and σ_{tr} are called *diffusion coefficient* and *transmission coefficient* respectively. The two coefficients are defined as follows:

$$D = \frac{1}{3\sigma_t'}$$

$$\sigma_{tr} = \sqrt{3\sigma_a \sigma_t'} = \sqrt{\frac{\sigma_a}{D}}$$

This is the equation describe light propagation in an infinite medium, i.e. no surface interaction is considered. In order to derive an actual BSSRDF model from the diffusion approximation, *boundary conditions* must be considered. Jensen Jensen et al. [2001] derived an analytical model starting from this approximation of the RTE, while Frisvad Frisvad et al. [2014] uses a higher order diffusion approximation of the RTE. The two models are explained in the following sections.

3.4.7 Standard dipole model

The first model we describe is due to Jensen et al. [2001]. It it usually referred in literature as *Jensen dipole model* or *Standard dipole model*. In their original paper, the authors used the diffusion approximation for light in an infinite medium. Starting from that, they derive an approximation that holds for light in a semi-infinite medium, i.e. light traveling in void hitting a planar slab of a translucent material.

As a boundary condition, we take the light coming *out* of the material. Light coming out of the material has a initial fluence ϕ_0. We assume then that the fluence decays linearly until a distance $z = 2AD$ from the surface, where it becomes zero. See Donner [2006] for the full derivation. D is the diffusion coefficient, while A is a corrective term that accounts for mismatching indices of refraction:

$$\begin{aligned} A &= \frac{1 + F_{dr}}{1 - F_{dr}} \\ F_{dr} &= \int_{2\pi} R(\eta, \vec{n} \cdot \vec{\omega})(\vec{n} \cdot \vec{\omega}) d\vec{\omega} \end{aligned} \tag{3.8}$$

Where R is the Fresnel reflection term as defined in Section 3.3.4, and $\eta = n_1/n_2$ is the relative refraction index. The Fresnel reflectance integral F_{dr} is usually approximated with an analytical expression:

$$F_{dr} = -\frac{1.440}{\eta^2} + \frac{0.710}{\eta} + 0.668 + 0.0036\eta$$

Given the boundary condition, we can then model the subsurface scattering in a point $\mathbf{x}_o$ with two small sources on the point, a configuration called a *dipole*. One source is placed beneath the surface, called the *real source*, while the other one is mirrored above the surface, called *virtual source*. The first source actually models the subsurface scattering effect, while the second one reduces the first in order to account for the boundary conditions and the extrapolation boundary. Refer to Figure 3.9 for a visual detail on the setup.

The real source is placed one mean free path beneath the surface, at $z_r = 1/\sigma_t'$, while the virtual one is placed symmetrically according to the boundary conditions, ad a distance $z_v = z_r + 4AD$. From z_r and z_v we can calculate the distances d_r and d_v from the entrance point $\mathbf{x}_i$. Given $r = \|\mathbf{x}_o - \mathbf{x}_i\|$, we obtain:

$$\begin{aligned} d_r &= \sqrt{z_r^2 + r^2} \\ d_v &= \sqrt{z_v^2 + r^2} \end{aligned}$$

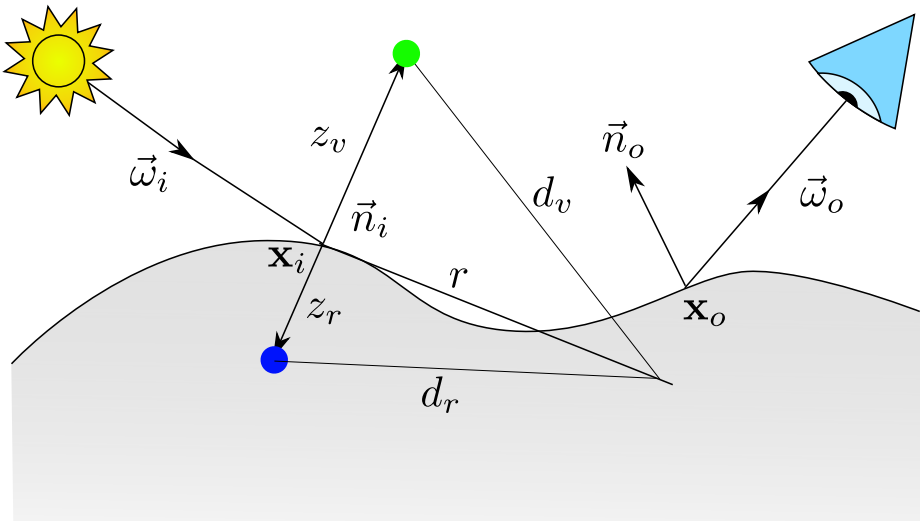


Figure 3.9: Setup for the standard dipole model.

Given these constraints, we obtain an equation for the BSSRDF in a semi infinite medium:

$$S_d(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_i, \vec{\omega}_o) = \frac{\alpha'}{4\pi^2}\left[\frac{z_r(1+\sigma_{tr}d_r)\ e^{-\sigma_{tr}d_r}}{d_r^3} + \frac{z_v(1+\sigma_{tr}d_v)\ e^{-\sigma_{tr}d_v}}{d_v^3}\right]$$

Where $\alpha' = \sigma_s'/\sigma_t'$ is called *reduced albedo*.

The model so far described was intended to model only the multiple scattering BSSRDF term, S_d. In order to obtain the full BSSRDF S, a single scattering term $S^{(1)}$ must be added. Moreover, we need to add as well the two Fresnel transmission terms, one for the incoming and one for the outgoing radiance. There are in literature many approaches to model single scattering, that are out of the scope of this report. The final BSSRDF equation for the standard dipole model then becomes:

$$S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) = T(\eta, \vec{\omega}_i) S_d(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) T(\eta, \vec{\omega}_o) + S^{(1)}(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o)$$

Jensen et al. [2001] in their original paper describes some corrections that need to be done to the model in order to make it work with generic surfaces, and on how to account for extensions like texture support. We will not describe these extensions here, remanding to the original paper for a detailed description.

3.4.8 Directional dipole model

Various evolutions to the standard dipole model have been proposed throughout the years. In this chapter, we will introduce the BSSRDF approximation called *directional dipole*, proposed by Frisvad et al. [2014]. In the standard dipole model, in fact, the diffusive part of the BSSRDF depends only on the distance between the point of incidence and the point of emergence, that is $S_d(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) = S_d(\|\mathbf{x}_o - \mathbf{x}_i\|)$.

The directional dipole model, based on the diffusion approximation, accounts for the direction of the incoming light in its calculations, in order to model the scattering effects more precisely. Moreover, the model, instead of splitting the BSSRDF in a multiple and single scattering term, splits the BSSRDF into a diffusive term S_d and a term $S_{\delta E}$, called *reduced intensity*, that can be computed using the delta-Eddington approximationJoseph et al. [1976]. The final BSSRDF thus becomes:

$$S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) = T(\eta, \vec{\omega}_i)(S_d(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o) + S_{\delta E}(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o))T(\eta, \vec{\omega}_o) \quad (3.9)$$

Where T are the Fresnel transmission coefficients for the incoming and outgoing directions. We note also that the diffusive part of the BSSRDF does not depend on the outgoing direction $\vec{\omega}_o$.

Diffusive BSSRDF
The diffusive part of the directional dipole model uses a first-order approximation of the RTE, that for a point light in an infinite medium gives the following fluence:

$$\phi(\mathbf{x}_o, \theta) = \frac{\Phi}{4\pi D} \frac{e^{\sigma_{tr} r}}{r} \left(1 + 3D \frac{1 + \sigma_{tr} r}{r} \cos\theta\right) \quad (3.10)$$

Where D and σ_{tr} are the two scattering coefficients defined beforehand, $r = \|\mathbf{x}_o\|$ and

$$\cos\theta = \frac{\mathbf{x} \cdot \vec{\omega}_{12}}{r}$$

Where $\vec{\omega}_{12}$ is the refracted vector as defined in Section 3.3.4. Comparing 3.10 with equation 3.7, we can see that we introduced a new term that depends on the angle θ between the refracted incoming light vector and the vector connecting incidence and emergence.

Using the diffusion approximation, we can first establish a relationship between the radiant exitance $M(\mathbf{x}_o)$ and the diffusive BSSRDF S'_d in an infinite medium:

$$\frac{dM(\mathbf{x}_o)}{d\Phi_i(\mathbf{x}, \vec{\omega}_i)} = T(\eta, \vec{\omega}_i) S'_d(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o)\ 4\pi C_\phi(1/\eta) \quad (3.11)$$

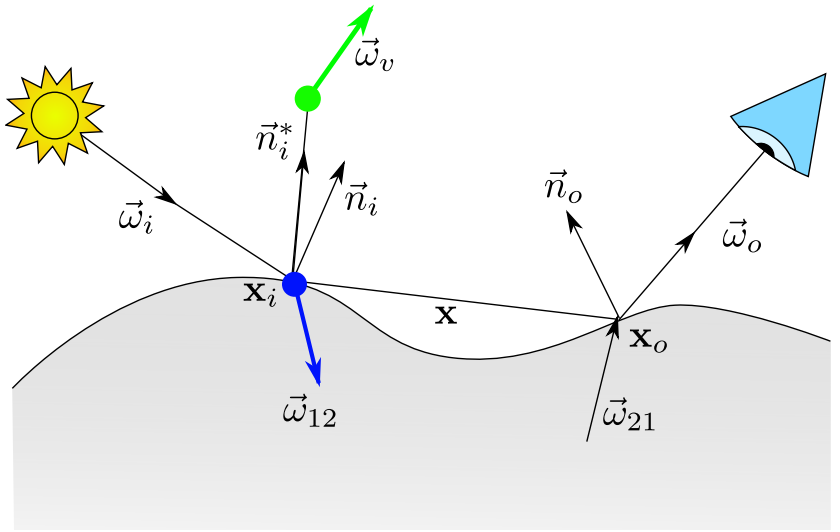


Figure 3.10: Setup for the directional dipole model.

Where $C_\phi(1/\eta)$ is related to the integral on the hemisphere of the fresnel coefficients. Using the definition of radiant exitance and inserting inside the classical diffusion approximation, we reach the diffusion formulation of the radiant exitance:

$$M(\mathbf{x}_o) = C_\phi(\eta)\phi(\mathbf{x}_o) + C_{\mathbf{E}}(\eta)D\vec{n}_o \cdot \nabla\phi(\mathbf{x}_o) \tag{3.12}$$

Again, $C_\phi(\eta)$ and $C_{\mathbf{E}}(\eta)$ are two terms that are related to the integration of the fresnel coefficients. Combining the three equations 3.10, 3.11 and 3.12, we reach the final form for our diffusive BSSRDF in an infinite medium:

$$\begin{aligned} S'_d(\mathbf{x}, \vec{\omega}_{12}, r) = & \frac{1}{4C_\phi(1/\eta)} \frac{1}{4\pi^2} \frac{e^{-\sigma_{tr} r}}{r^3} \\ & \left[C_\phi(\eta) \left(\frac{r^2}{D} + 3(1+\sigma_{tr} r)\mathbf{x} \cdot \vec{\omega}_{12} \right) - \right. \\ & - C_{\mathbf{E}}(\eta) \Big(3D(1+\sigma_{tr} r)\ \vec{\omega}_{12} \cdot \vec{n}_o - \\ & \left. - \left((1+\sigma_{tr} r) + 3D \frac{3(1+\sigma_{tr} r) + (\sigma_{tr} r)^2}{r^2} \mathbf{x} \cdot \vec{\omega}_{12} \right) \mathbf{x} \cdot \vec{n}_o \Big) \right] \end{aligned} \tag{3.13}$$

Fresnel integrals

The two terms $C_\phi(\eta)$ and $C_{\mathbf{E}}(\eta)$ originally come from integrating the outgoing Fresnel transmittance over the whole outgoing hemisphere, weighted with a cosine term. The two functions are defined as follows:

$$\begin{aligned} C_\phi(\eta) &= \frac{1}{4\pi}\int_{2\pi} T(\eta,\vec{\omega})(\vec{n}_o\cdot\vec{\omega})d\vec{\omega} \\ C_{\mathbf{E}}(\eta) &= \frac{3}{4\pi}\int_{2\pi} T(\eta,\vec{\omega})(\vec{n}_o\cdot\vec{\omega})^2 d\vec{\omega} \end{aligned} \tag{3.14}$$

These two integrals can be rearranged in order to express them in terms of reflectance instead of transmittance, recalling $R = 1 - T$.

$$\begin{aligned} C_\phi(\eta) &= \frac{1}{4\pi}\left(\pi - \int_{2\pi} R(\eta,\vec{\omega})(\vec{n}_o\cdot\vec{\omega})d\vec{\omega}\right) = \frac{1}{4}(1-2C_1) \\ C_{\mathbf{E}}(\eta) &= \frac{3}{4\pi}\left(\frac{2\pi}{3} - \int_{2\pi} R(\eta,\vec{\omega})(\vec{n}_o\cdot\vec{\omega})d\vec{\omega}\right) = \frac{1}{2}(1-3C_2) \end{aligned} \tag{3.15}$$

Even with this rearrangement the integrals cannot be expressed in closed form. D'Eon and Irving [2011] use a convenient polynomial approximation for the two coefficients C_1 and C_2, expressed as:

$$2C_1 \approx \begin{cases} +0.919317 - 3.4793\eta + 6.75335\eta^2 - 7.80989\eta^3 \\ \qquad +4.98554\eta^4 - 1.36881\eta^5 & \eta < 1 \\ -9.23372 + 22.2272\eta - 20.9292\eta^2 + 10.2291\eta^3 \\ \qquad -2.54396\eta^4 + 0.254913\eta^5 & \eta \geq 1 \end{cases}$$

$$3C_2 \approx \begin{cases} 0.828421 - 2.62051\eta + 3.36231\eta^2 - 1.95284\eta^3 \\ \qquad +0.236494\eta^4 + 0.145787\eta^5 & \eta < 1 \\ -1641.1 + \frac{135.926}{\eta^3} - \frac{656.175}{\eta^2} + \frac{1376.53}{\eta} + 1213.67\eta \\ \qquad -568.556\eta^2 + 164.798\eta^3 \\ \qquad -27.0181\eta^4 + 1.91826\eta^5 & \eta \geq 1. \end{cases}$$

Boundary conditions

As the name implies, also for the directional dipole we model the boundary conditions on the material interface using a dipole. In this case, however, instead of using two point light sources, we use two ray sources, a real and a virtual one. As in the standard dipole, the source is displaced towards the normal of a

distance d_e. In the case of the standard dipole, we use $2D$, that becomes $2AD$ in the case of mismatching indices of refraction on the interface. In the case of the directional dipole, we use

$$d_e = \frac{2.131D}{\sqrt{\alpha'}}$$

Where we recall $\alpha' = \sigma'_s/\sigma'_t$ as the reduced albedo. This result have been proven [Davison and Sykes, 1958] to be consistent with numerical simulations of the RTE. In addition, the A term is modified using the hemispheric Fresnel integrals:

$$A(\eta) = \frac{1 - C_{\mathbf{E}}(\eta)}{2C_\phi(\eta)}$$

As the standard dipole, the directional dipole assumes a semi-infinite medium given the previous boundary conditions. In order to relax this assumptions, we need to further extend the model in order to reduce undesired effects. One first modification proposed by Frisvad et al. [2014] is to use a modified tangent plane defined by the normal $\vec{n}_i^*$ to mirror the real source towards the mirror light source, instead of the obvious one defined by $\vec{n}_i$. We define the modified normal as follows:

$$\vec{n}_i^* = \begin{cases} \vec{n}_i & \text{for } \mathbf{x}_o = \mathbf{x}_i \\ \dfrac{\mathbf{x}_o - \mathbf{x}_i}{\|\mathbf{x}_o - \mathbf{x}_i\|} \times \dfrac{\vec{n}_i \times (\mathbf{x}_o - \mathbf{x}_i)}{\|\vec{n}_i \times (\mathbf{x}_o - \mathbf{x}_i)\|} & \text{otherwise} \end{cases}$$

Another important modification is the distance to the real source. In the standard dipole, we used $d_r = \sqrt{z_r^2 + r^2}$, with $z_r = 1/\sigma'_t$, which is the average distance a photon travels within the material before being absorbed or scattered. The problem of this definition is that it introduces a singularity in $r = 0$. Moreover, the standard dipole becomes fairly imprecise when r is small, overestimating the overall effect. In order to avoid these problems, Frisvad et al. [2014] proposed a more complicated definition of d_r that matches simulations of transport theory more closely. For the details, see Appendix B in the original paper. d_r is defined as follows, recalling $\sigma_t = \sigma_s + \sigma_a$:

$$d_r^2 = \begin{cases} r^2 + D\mu_0(D\mu_0 - 2d_e \cos\beta) & \mu_0 \geq 0 \text{ (frontlit)} \\ r^2 + \dfrac{1}{(3\sigma_t)^2} & \mu_0 < 0 \text{ (backlit)} \end{cases}$$

Where $\mu_0 = -\vec{\omega}_{12} \cdot \vec{n}_o$ is an indicator if the point $\mathbf{x}_o$ is frontlit or backlit. β is a geometry term that is evaluated as:

$$\cos\beta = -\sqrt{\frac{r^2 - (\mathbf{x} \cdot \vec{\omega}_{12})^2}{r^2 + d_e^2}}$$

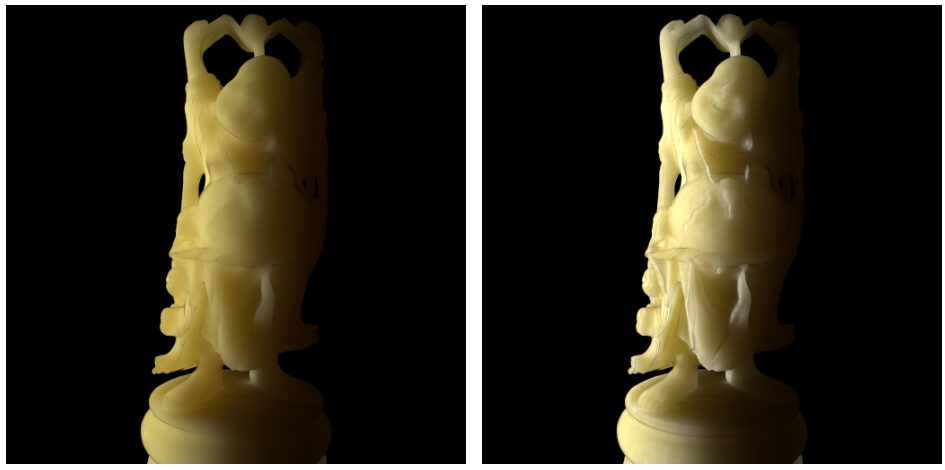
Combining all the corrections seen so far, we can write the final form of our BSSRDF model, that is a combination of the real source term minus the virtual source term:

$$S_d(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o) = S'_d(\mathbf{x}_o - \mathbf{x}_i, \vec{\omega}_{12}, d_r) - S'_d(\mathbf{x}_o - \mathbf{x}_v, \vec{\omega}_v, d_v)$$

Where the extra coefficients for the virtual source are defined as follows:

$$\begin{aligned} x_v &= x_i + 2Ad_e\vec{n}_i^* \\ \vec{\omega}_v &= \vec{\omega}_{12} - 2(\vec{\omega}_{12} \cdot \vec{n}_i^*)\vec{n}_i^* \\ d_v &= \|x_o - x_v\| \end{aligned}$$

The directional dipole model described in this chapter, gives a better result that the standard dipole, at the extra price of additional calculations. In particular, the model improves the previous one for highly forward scattering materials, where it is sensibly closer to the path traced result. The final goal of this thesis is to provide a real-time implementation of it. Given this theoretical introduction, in the next section we will describe our contribution in order to breakdown the problems and the issues of a real-time implementation.



(a) Standard dipole **(b)** Directional Dipole

Figure 3.11: Comparison of the standard and the directional dipole model, for a Stanford Happy Buddha made of potato, with a side directional light. We can see that the directional dipole is able to capture finer details and provide a generally less flat appearance than the standard one.

Chapter 4

Method

In this chapter, after giving the theoretical foundations in Chapter 3, we introduce our method to render translucent materials efficiently using the directional dipole model. This chapter connects the theory with the implementation in Chapter 5. First of all, we will give a theoretical justification of our method, deriving a discretization of the rendering equation that can be actually solved and implemented in a GPU environment. Then, we will discuss some possible sampling patterns and how they could possibly improve the results of the final rendering. Then, we will introduce some details on the acquisition of scattering parameters in an experimental environment. Finally, we will describe a method to approximate environment lighting using an arbitrary number of directional sources.

4.1 Method overview

First of all, we recall the general form of the rendering equation for participating media using a BSSRDF (equation 3.4):

$$L_o(\mathbf{x}_o, \vec{\omega}_o) = L_e(\mathbf{x}_o, \vec{\omega}_o) + \int_A \int_{2\pi} S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) L_i(\mathbf{x}_i, \vec{\omega}_i) V(\mathbf{x}_i)(\vec{n}_i \cdot \vec{\omega}_i) d\vec{\omega}_i dA_i$$

In the usual approach to offline path traced rendering, we need to integrate the radiance from all the possible sources on the surface point seen by each pixel of the final image. For this surface element subtended by one pixel, all the incoming radiance contributions from the other points in the scene must be accounted an then multiplied by the BSSRDF function in the direction of the camera. In a way, we are basically performing the integral in equation 3.4 numerically. If we use a BSSRDF function, the contribution from all the points from the other surfaces must be employed, while in the case of a BRDF some contributions may be excluded. Given its natural exponential explosion, path tracing is not generally suitable for real-time rendering.

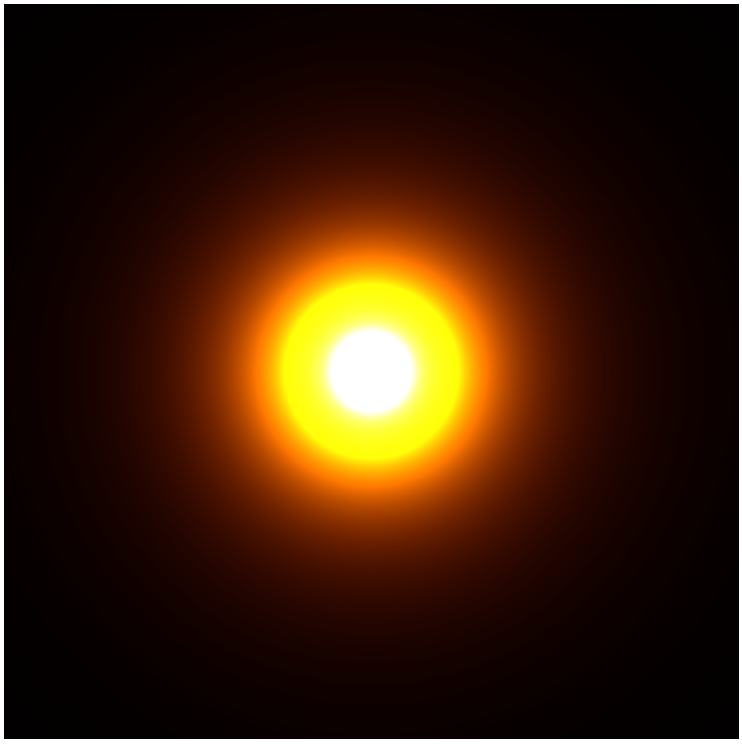


Figure 4.1: Simulation of the directional dipole BSSRDF of a laser hitting a slab of 2x2 cm of potato material. We note the exponential decay of subsurface scattering phenomena.

In our method, in its final goal to be real time, we perform the same integral as equation 3.4, but under some assumptions and restrictions that allow us to perform it more efficiently. In addition, since our method approximates the *integral* and not the BSSRDF function, it is applicable to any BSSRDF function, given that it has limited or no dependence on the outgoing direction $\vec{\omega}_o$, like the

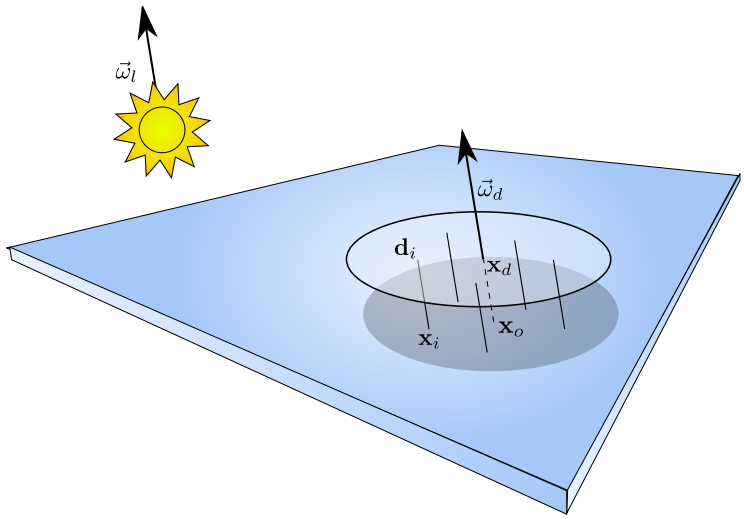


Figure 4.2: Setup for our method: the disk is placed on the point $\mathbf{x}_o$, displaced along the disc direction $\vec{\omega}_d$ and then the sample points $\mathbf{d}_i$ are reprojected back to find the samples $\mathbf{x}_i$.

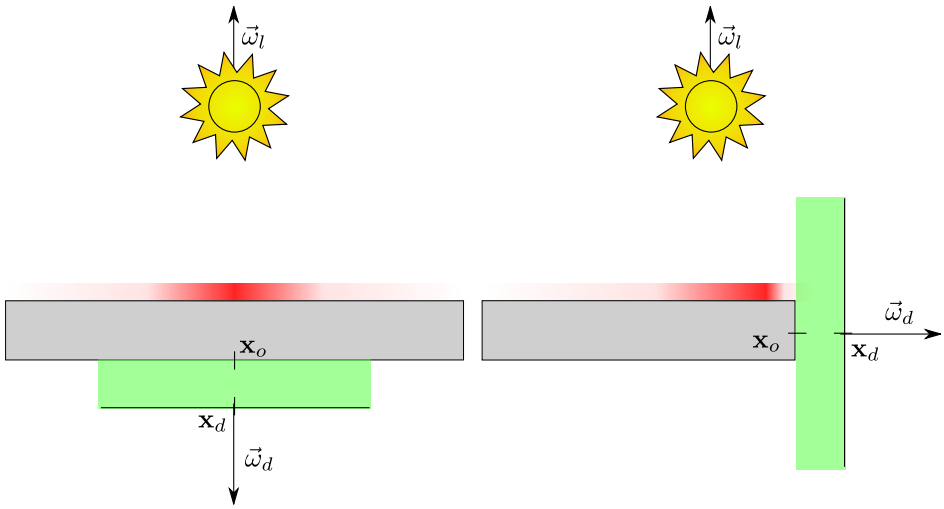
directional dipole model.

The idea on approximating the integral comes from the fact that the directional dipole magnitude (and subsurface scattering effects) decays exponentially from the point of incidence, as we can see from figure 4.1. So, the subsurface scattering contribution for points that are far apart becomes quickly negligible. The distance to which this happens is related to the transmission coefficient σ_{tr}. We will investigate this relation better in the result section.

So, given this exponential decay, we place a disc on the surface for a position $\mathbf{x}_o$. We define a sampling disc as a point $\mathbf{x}_d$, a radius r_d and a direction $\vec{\omega}_d$ From this disc, we chose a subset of points, that are then projected on the surface and used to calculate the BSSRDF contribution. This set of points $\mathbf{x}_i$ is called *sample points*. To obtain a sample point from a disc point $\mathbf{d}_i$, we take the point that intersects the surface along the line with direction $\vec{\omega}_d$ passing from the point $\mathbf{d_i}$ (see figure 4.2 for an illustration of the process). In formulas, we do:

$$\mathbf{x}_i = \mathbf{d}_i - s\vec{\omega}_d, s \in \mathbb{R}, \mathbf{x}_i \in M$$

We note that we can have multiple s solving this equation, or none at all. If at least one s exists, we take the one that makes the distance between $\mathbf{d}_i$ and $\mathbf{x}_i$ the smallest. Then, we calculate and average the BSSRDF contribution from these points $\mathbf{x}_i$ on the point x_o. The process can be repeated more times for multiple lights, using the same set of sampling points.

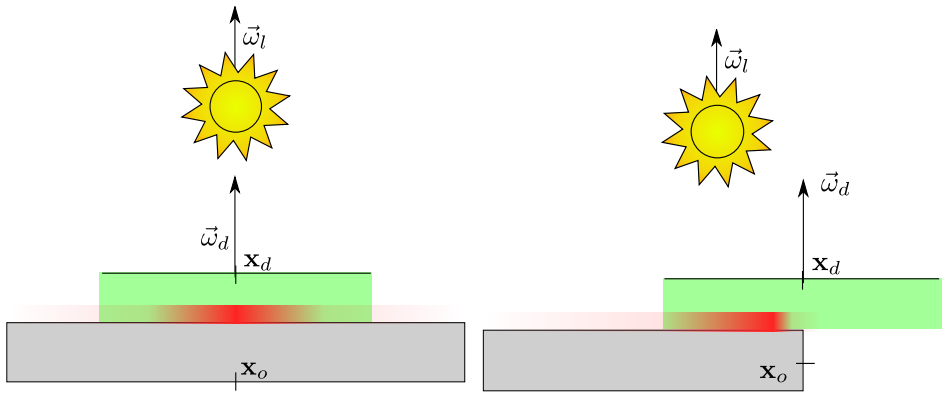


(a) Bottom sample, naïve approach **(b)** Side sample, naïve approach

Figure 4.3: Counter examples of the naïve choice for $\mathbf{x}_d$ and $\vec{\omega}_d$. The green area represent the sampling area of the disc (see from the side). The red area shows where on the surface the contribution is higher for the point $\mathbf{x}_o$. We can see that with this approach all the contribution is actually missed.

Given this setup, we need to find a way of efficiently placing the disc in order to get a good approximation of the BSSRDF. In fact, if the disc is placed in the wrong position, the accounted contribution from the sampling points will not be correct. Moreover, also the orientation of the disc is important, in order to not undersample light in certain regions of the model. A naïve approach would suggest to pick $\mathbf{x}_d = \mathbf{x}_o$ and $\vec{\omega}_d = \vec{n}_o$, but we can see that this approach is not correct. The most obvious counter examples are displayed in Figure 4.3: in the first example 4.3a, the contribution from the surrounding points will be zero, because none of the sampling points is directly illuminated by the light, so the visibility term on the rendering equation will evaluate to zero. In the second example (figure 4.3b), the wrong direction $\vec{n}_o$ prevents the most of the points from being sampled. A more careful choice for placing the points in the disc is to place the disc in a way that the obtained surface point is always the closest one to the light. To ensure that the points $\mathbf{x}_i$ always have this property, we can place $\mathbf{x}_d$ far enough from the surface in a way that all the sampling points have to be the closest to the light. If we define a *bounding box vector* in the same coordinate frames, we have a simple formulation for $\mathbf{x}_d$:

$$\mathbf{x}_d = \mathbf{x}_o + (\mathbf{b} \cdot \vec{\omega}_l)\vec{\omega}_l$$



(a) Bottom sample, our method **(b)** Side sample, our method

Figure 4.4: The counter examples of figure 4.3 updated using our method of choice for $\mathbf{x}_d$. We can see that now most of the radiance distribution is accounted for.

The bounding box vector is a vector where its components are the maximum extension of the mesh in the coordinate reference system. So,

$$\mathbf{b} = (\max(\mathbf{x}_i^x) - \min(\mathbf{x}_i^x), \max(\mathbf{x}_i^y) - \min(\mathbf{x}_i^y), \max(\mathbf{x}_i^z) - \min(\mathbf{x}_i^z)), \quad \forall \mathbf{x}_i$$

To solve the problem in the second example in figure 4.3b, we chose to always orient the circle towards the light, that is $\vec{\omega}_d = \vec{\omega}_l$ for directional lights and $\vec{\omega} = \frac{\mathbf{x}_l - \mathbf{x}_d}{\|\mathbf{x}_l - \mathbf{x}_d\|}$ for point lights. As we can see, for the new choices of disc placement and orientation we are able to catch the points from where the contribution is stronger. This new setup seems much more complicated that the naïve approach, but as we will see in the implementation section the Z-buffer of the GPU will permit us to get the desired points without any additional computation, using shadow mapping. For environment lights, we will see how to transform them in directional lights at the end of this chapter, so the setup of the equations will be the same for directional lights.

4.1.1 Approximation of the rendering equation

Going into the mathematical details the idea is to take the integral form of the rendering equation (equation 3.4):

$$L_o(\mathbf{x}_o, \vec{\omega}_o) = L_e(\mathbf{x}_o, \vec{\omega}_o) + \int_A \int_{2\pi} S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) L_i(\mathbf{x}_i, \vec{\omega}_i) V(\mathbf{x}_i) (\vec{n}_i \cdot \vec{\omega}_i) d\vec{\omega}_i dA_i$$

First of all, we make the assumption of a body that is not emitting light: all the radiance from the body comes from an external source. This assumption

can be trivially relaxed and implemented, but to simplify the equation in this chapter we will exclude it from the calculations. Secondly, we limit ourselves to the case of one directional light, treating the case of a point light later as an extension. The directional light direction $\vec{\omega}_l$ and radiance $L_d\ \delta(\vec{\omega}_d)$.

Under the first assumption, equation 3.4 becomes:

$$L_o^D(\mathbf{x}_o, \vec{\omega}_o) = \int_A \int_{2\pi} S(\mathbf{x}_i, \vec{\omega}_i, \mathbf{x}_o, \vec{\omega}_o) L_d\ \delta(\vec{\omega}_l)\ V(\mathbf{x}_i)(\vec{n}_i \cdot \vec{\omega}_i) d\vec{\omega}_i dA_i$$
$$L_o^D(\mathbf{x}_o, \vec{\omega}_o) = \int_A S(\mathbf{x}_i, \vec{\omega}_l, \mathbf{x}_o, \vec{\omega}_o) L_d V(\mathbf{x}_i)(\vec{n}_i \cdot \vec{\omega}_l) dA_i$$

In this way, we remove the internal integral. Then, we need to discretize the remaining integral. We imagine to have a set of N points on the surface. We assume that each one of these points is visible from the light source (so we can get rid of the $V(\mathbf{x}_i)$ term). We will discuss in the implementation section how to make sure that all these points are visible. Each one of these points has an associated area A_i, so that we can write:

$$L_o^D(\mathbf{x}_o, \vec{\omega}_o) = L_d \sum_{i=1}^{N} S(\mathbf{x}_i, \vec{\omega}_l, \mathbf{x}_o, \vec{\omega}_o)(\vec{n}_i \cdot \vec{\omega}_l) A_i \tag{4.1}$$

Now, instead of using all the points on the surface, we consider only the points within a certain radius r^* from the point $\mathbf{x}_o$, i.e. the disk we discussed in Section 4.1. Assuming the points are distributed uniformly on the disk, we obtain the following area for a point:

$$A_i = \frac{A_c}{N\ (\vec{n}_i \cdot \vec{\omega}_l)}$$

Where $A_c = \pi(r^*)^2$ is the area of the circle. The $\vec{n}_i \cdot \vec{\omega}_l$ accounts for the fact that the area is projected on the surface. Inserting into equation 4.1, we obtain:

$$L_o^D(\mathbf{x}_o, \vec{\omega}_o) = L_d \frac{A_c}{N} \sum_{i=1}^{N} S(\mathbf{x}_i, \vec{\omega}_l, \mathbf{x}_o, \vec{\omega}_o) \tag{4.2}$$

That is our final approximation for a directional light. For a point light, following the exact same steps, we reach a similar solution. We recall that a point light is defined by an intensity I_p and a source point $\mathbf{x}_p$:

$$L_o^P(\mathbf{x}_o, \vec{\omega}_o) = I_p \frac{A_c}{N} \sum_{i=1}^{N} \frac{S(\mathbf{x}_i, \frac{\mathbf{x}_p - \mathbf{x}_i}{\|\mathbf{x}_p - \mathbf{x}_i\|}, \mathbf{x}_o, \vec{\omega}_o)}{\|\mathbf{x}_p - \mathbf{x}_i\|^2} \tag{4.3}$$

And, since the radiance is linearly summable, we can combine the contribution from an arbitrary number of $P_1, P_2 ... P_p$ point sources and $D_1, D_2 ... D_d$ direc-

tional sources:

$$\begin{aligned} & L_o(\mathbf{x}_o, \vec{\omega}_o) = \\ & = \sum_{k=1}^{p} L_o^{P_k}(\mathbf{x}_o, \vec{\omega}_o) + \sum_{k=1}^{d} L_o^{D_k}(\mathbf{x}_o, \vec{\omega}_o) \\ & = \frac{A_c}{N} \left[\sum_{k=1}^{p} I_p^k \sum_{i=1}^{N} \frac{S(\mathbf{x}_i, \frac{\mathbf{x}_p^k - \mathbf{x}_i}{\|\mathbf{x}_p^k - \mathbf{x}_i\|}, \mathbf{x}_o, \vec{\omega}_o)}{\|\mathbf{x}_p^k - \mathbf{x}_i\|^2} + \sum_{k=1}^{d} L_d^k \sum_{i=1}^{N} S(\mathbf{x}_i, \vec{\omega}_l^k, \mathbf{x}_o, \vec{\omega}_o) \right] \end{aligned} \tag{4.4}$$

4.2 Sampling patterns

As we discussed, the BSSRDF function for the directional dipole is dominated by an exponential decay. So, it is more probable to find points that contribute more to the BSSRDF if we take points closer to the evaluation point $\mathbf{x}_o$. However, our assumption of uniform areas in the previous calculations does not hold anymore, so we need to modify the previous equations in order to account for the non-linear sampling.

Assuming to have number generator that can generate numbers on a disc, we can create an exponentially distributed disc with the following exponential probability distribution function (PDF):

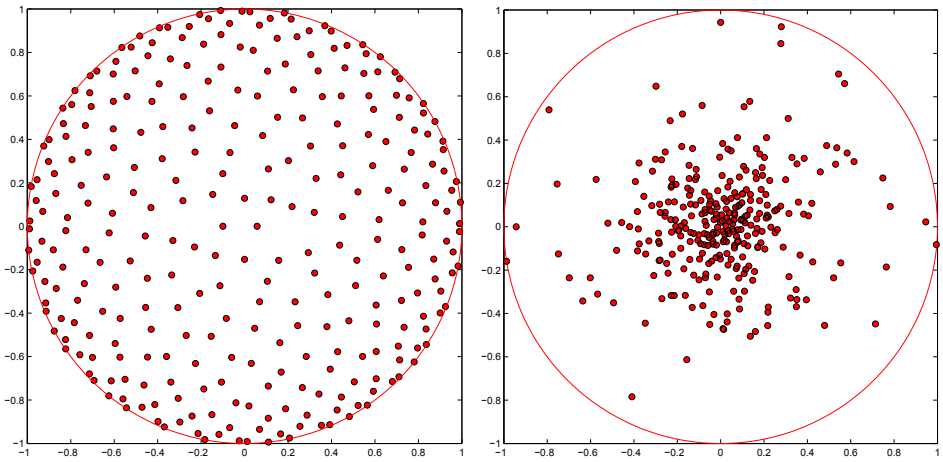
$$pdf(x) = \sigma_{tr} e^{-\sigma_{tr} x}$$

The difference between a disc sampled using this distribution and an uniform one is shown in figure 4.5. The process to create this distribution using rejection sampling will be illustrated in the implementation section. We take the radius of a point $\mathbf{x}_i$ as $r_i = \|\mathbf{d_i}\|$. So now we have a new normalization term to include in order to scale back the result. So, we need now to divide by a $exp(-\sigma_{tr} r_i)$ term each sample. The new equation for a directional light instead of 4.2 then becomes:

$$\hat{L}_o^D(\mathbf{x}_o, \vec{\omega}_o) = L_d \frac{A_c}{N} \sum_{i=1}^{N} S(\mathbf{x}_i, \vec{\omega}_l, \mathbf{x}_o, \vec{\omega}_o) e^{\sigma_{tr} r_i}$$

The other two equations 4.3 and 4.4 then change accordingly:

$$\hat{L}_o^P(\mathbf{x}_o, \vec{\omega}_o) = I_p \frac{A_c}{N} \sum_{i=1}^{N} \frac{S(\mathbf{x}_i, \frac{\mathbf{x}_p - \mathbf{x}_i}{\|\mathbf{x}_p - \mathbf{x}_i\|}, \mathbf{x}_o, \vec{\omega}_o)}{\|\mathbf{x}_p - \mathbf{x}_i\|^2} e^{\sigma_{tr} r_i}$$



(a) Uniform sampling ($\sigma_{tr} = 1$) **(b)** Exponential sampling ($\sigma_{tr} = 70$)

Figure 4.5: Uniform versus exponentially-weighted sampling of 300 points.

$$\hat{L}_o(\mathbf{x}_o, \vec{\omega}_o) =$$

$$= \sum_{k=1}^{p} \hat{L}_o^{P_k}(\mathbf{x}_o, \vec{\omega}_o) + \sum_{k=1}^{d} \hat{L}_o^{D_k}(\mathbf{x}_o, \vec{\omega}_o)$$

$$= \frac{A_c}{N} \left[\sum_{k=1}^{p} I_p^k \sum_{i=1}^{N} \frac{S(\mathbf{x}_i, \frac{\mathbf{x}_p^k - \mathbf{x}_i}{\|\mathbf{x}_p^k - \mathbf{x}_i\|}, \mathbf{x}_o, \vec{\omega}_o)}{\|\mathbf{x}_p^k - \mathbf{x}_i\|^2} e^{\sigma_{tr} r_i} + \sum_{k=1}^{d} L_d^k \sum_{i=1}^{N} S(\mathbf{x}_i, \vec{\omega}_l^k, \mathbf{x}_o, \vec{\omega}_o) e^{\sigma_{tr} r_i} \right]$$

4.3 Parameter acquisition

When rendering translucent materials, it is important that we have the right scattering properties, in order to match the appearance of real world objects. The scattering parameters may be tweaked by the artist and set up manually, but this is a long process since the the scattering properties are not directly related to material appearance. In order to avoid this problems, the scattering parameters are measured from samples taken from real world objects. In this section, we will give an overview of two methods used to estimate the scattering parameters.

The first method was presented alongside the standard dipole model by Jensen et al. [2001]. The measurement apparatus consists of a series of lenses that

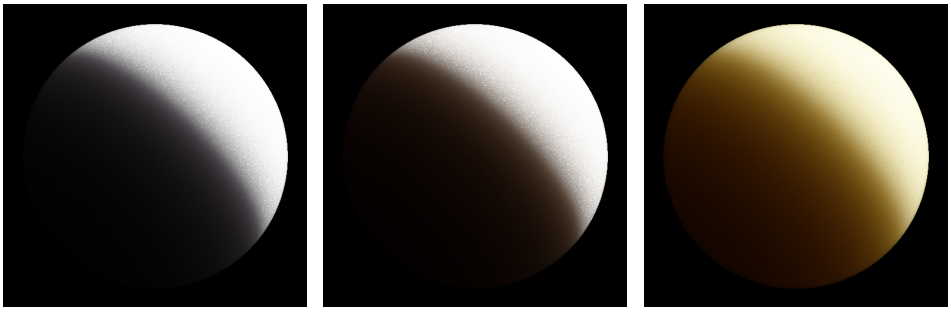
focus the light on the sample. The light power Φ is measured by calibrating the sensor with a spectralon sample. A picture of the sample is then acquired at different exposure, in order to build an high dynamic range image. This is necessary since the scattering decays exponentially, so a high range is needed to have meaningful measurements. The measured data are then fitted to diffusion theory in order to obtain the scattering coefficients. Due to the nature of the measurement, it is not possible to measure the mean cosine g of the material, but only the reduced scattering coefficient $\sigma'_s = \sigma_s(1 - g)$ and the absorption coefficient σ_a. This measurement model uses the diffusion approximation to work, so it shares the same limitations: it is valid only for materials where $\sigma_a \ll \sigma_s$, or highly scattering materials.

The second method, proposed by Narasimhan et al. [2006] proposes a method to measure the scattering coefficients by dilution. The assumption is that water does not interfere with the scattering properties of the materials dissolved within it for small distances (less than 50 cm). Naturally, the material needs then to be already in a liquid form, or to be a powder that can be easily dissolved in water. The setup of the experiment is a box full of water with a camera and an area light. High dynamic range picture of the material dissolved in water are then taken, and the scattering coefficients can be measured analyzing the resulting images. Various measurements at different concentrations are needed in order to get an effective measurement of the coefficients, but then the coefficients can be extrapolated for any concentration.

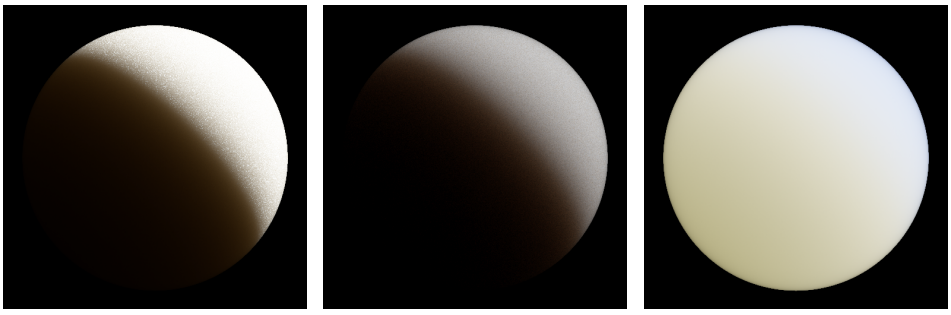
Some of the scattering properties measured thanks to this method are reported in table 4.1. This coefficients will be used throughout the report when referencing to a specific material.

Material	Absorption, σ_a			Scattering, σ_s			Mean cosine, g			η	Source
	R	G	B	R	G	B	R	G	B		
Apple	0.0030	0.0034	0.0046	2.29	2.39	1.97	-	-	-	1.3	J
Ketchup	0.061	0.97	1.45	0.18	0.07	0.03	-	-	-	1.3	J
Marble	0.0021	0.0041	0.0071	2.19	2.62	3.00	-	-	-	1.5	J
Potato	0.0024	0.0090	0.12	0.68	0.70	0.55	-	-	-	1.3	J
Whole milk	0.0011	0.0024	0.014	2.55	3.21	3.77	-	-	-	1.3	J
Coffee	0.1669	0.2287	0.3078	0.2707	0.2828	0.297	0.907	0.896	0.88	1.3	N
Soy milk	0.0001	0.0005	0.0034	0.2433	0.2714	0.4563	0.873	0.858	0.832	1.3	N
Wine (merlot)	0.7586	1.6429	1.9196	0.0053	0	0	0.974	0	0	1.3	N
Beer (Budweiser)	0.1449	0.3141	0.7286	0.0037	0.0069	0.0074	0.917	0.956	0.982	1.3	N
White grapefruit juice	0.0096	0.0131	0.0395	0.3513	0.3669	0.5237	0.548	0.545	0.565	1.3	N

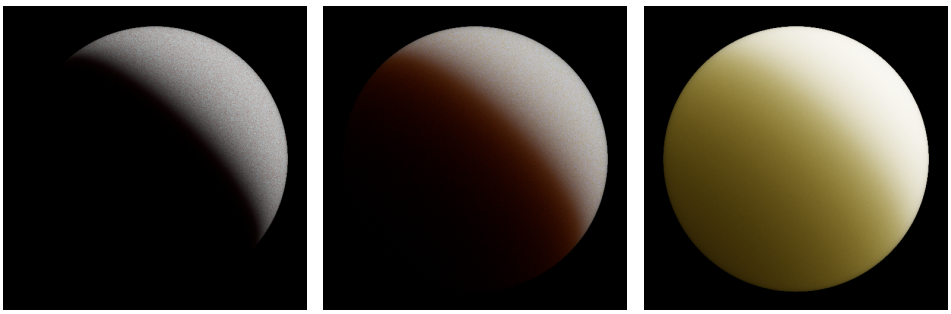
Table 4.1: Scattering material parameters estimated using different methods. For the source field, `J` materials come from Jensen et al. [2001], while `N` materials come from Narasimhan et al. [2006]. Note that the materials measured with the technique proposed in Jensen et al. [2001] are without the g coefficient.



(a) Apple **(b)** Marble **(c)** Potato



(d) Whole Milk **(e)** Coffee **(f)** Soy Milk



(g) Wine (merlot) **(h)** Beer (Budweiser) **(i)** White Grapefruit Juice

Figure 4.6: Path traced renderings on a sphere of the materials reported in table 4.1.

4.4 Environment lights

Environment lighting is a omni-directional type of lighting that represents light coming from an environment. Instead of defining a light as a direction or a point in space, we define directly the radiance distribution on a map. The map is usually provided in a HDR format in order to cover various range of radiance values, and it is usually given as a set of six cube faces (cubemap) or as a latitude-longitude map, as the one in Figure 4.7. The maps are usually given in equirectangular projection.

In the game development community, spherical harmonics (SH) [Green, 2003, Sloan, 2008] are usually employed. This technique, given an heavy pre-computation step, transforms the radiance map into a set of coefficients in the spherical harmonics basis. This coefficients can be used easily to represent the low frequency component of the radiance map Ramamoorthi and Hanrahan [2001].

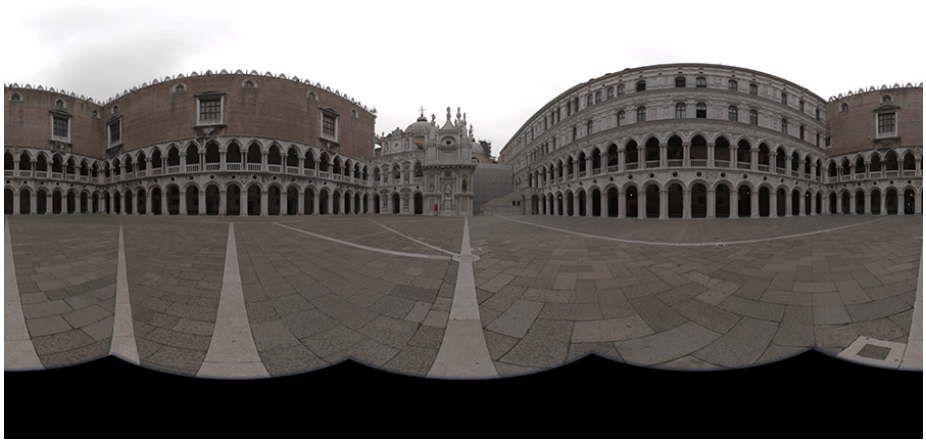


Figure 4.7: Latitude - longitude environment map of the inner courtyard of the Doge's palace in Venice (Doge map). The map has been converted to a RGB format from the original HDR format. This and all the maps used in this report are courtesy of `http://gl.ict.usc.edu/Data/HighResProbes/`.

In this chapter, we introduce a technique presented in Pharr and Humphreys [2004] to convert a environment map into a set of directional light sources of arbitrary size. The general idea is to generate a set of random points and them transform them according to a pre-computed probability distribution. This distribution make the random point concentrate in areas where the radiance is higher, so that it is possible to get the most representative points on the radiance

map. The found points are then transformed into a spherical coordinate basis in order to get the light direction.

We start defining our image as an array of n rows and m columns. We need to define a function that of each pixel of the image gives us a single radiance value. Instead of using radiance, we use luminance. The ITU-R recommendation standard BT.709 [ITU, 2001] gives us a formula to obtain luminance from spectral radiance values:

$$f(u,v) = 0.2126\ R(u,v) + 0.7152\ G(u,v) + 0.0722\ B(u,v)$$

Where $[u,v] \in [0,n) \times [0,m)$. R, G and B represent the spectral coordinates of the radiance map. We would like to define now a probability distribution function based on the $f(u,v)$ function. We can define it simply by normalizing f with the integral of the function over the domain:

$$p(u,v) = \frac{f(u,v)}{\iint f(u,v)dudv} = \frac{f(u,v)}{\sum_u \sum_v f(u,v)}$$

However, in order to be able to sample from the distribution $p(u,v)$, there are some things to take care about. We would like now to separate the two variables, in order to sample from two one-dimensional distributions, instead of one two-dimensional distribution. To do this, we use the conditional probability formula:

$$p(u,v) = p_v(v|u)p_u(u) \tag{4.5}$$

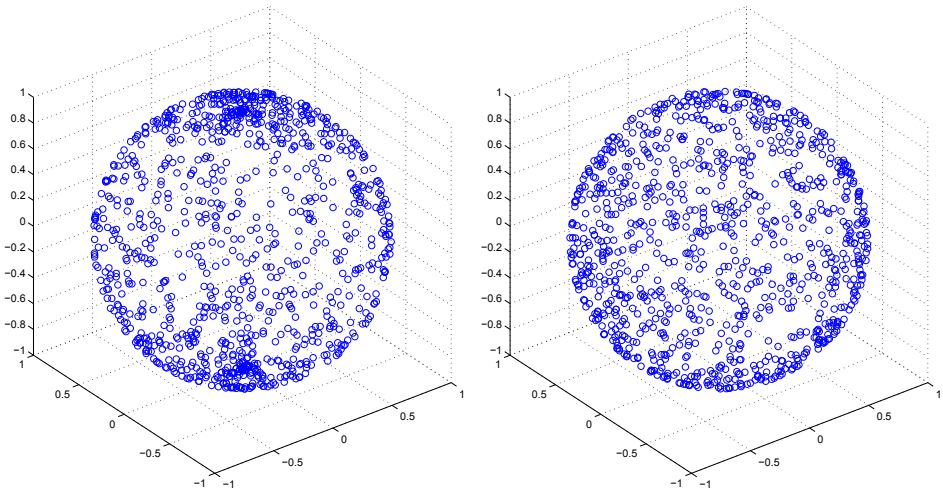
So that we first choose u, sample its probability $p_u(u)$ and then compute the conditional density $p_v(v|u)$ using the found value for u. Using the marginal formulas, the first probability is easily found:

$$p_u(u) = \int p(u,v)dv = \frac{\sum_v f(u,v)}{\sum_u \sum_v f(u,v)}$$

And, from equation 4.5, we find the conditional probability:

$$p_v(v|u) = \frac{p(u,v)}{p_u(u)} = \frac{f(u,v)}{\sum_v f(u,v)} \sin\theta$$

The $\sin\theta$ comes from the fact that the latitude-longitude map with a equirectangular projection is not area preserving. So, the sampling must take into account the distortion of the map, otherwise the samples will be more concentrated at the poles, rather than distributed uniformly on the sphere. We can appreciate



(a) Without $\sin\theta$ correction **(b)** With $\sin\theta$ correction

Figure 4.8: Effect of $\sin\theta$ on a random point distribution on a sphere.

the difference for a random sampling on a sphere in Figure 4.8. Now we are ready to sample the function. In order to bias our sample according to the radiance distribution, we need to calculate the cumulative distribution function (CDF) for a one-dimensional probability distribution function, then inverse sample it. The CDF is defined as:

$$c_u(u) = \int_{-\infty}^{u} p_u(u)du = \sum_{i=0}^{u} p_u(u)$$

That is the discrete integral of the function up to the point u. Figure 4.9 explains why we need to inverse sample the CDF. If, as the figure, all the radiance is distributed towards the right side of the picture (i.e. the CDF rises slowly), if we pick a set of random points (on the y axis) and inverse sample the CDF on them (on the x axis), we obtain a new set of points that is biased towards the highest concentration of radiance. Let us now pick a couple of random points $(\zeta_1, \zeta_2) \in [0, 1)^2$. We then convert these points to a pair of coordinates $(u_1, u_2) \in [0, n) \times [0, m)$ by inverse sampling of the CDF. We will give the details of how to discretize this process in the implementation section. Then, we obtain the spherical coordinates using the standard formula:

$$(\theta, \phi) = \left(\frac{u_1}{\pi n}, \frac{u_2}{2\pi m}\right)$$

And, from the spherical coordinates, we use the equirectangular projection formula to transform them into a vector in the 3D space, that is the final direction

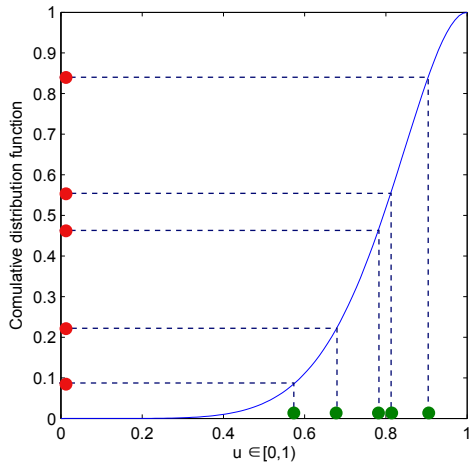


Figure 4.9: Effect of weighting values with the CDF. We note that the random values on the y axis are transformed and accumulated where the CDF is steeper.

for our light.

$$\vec{\omega}_l = (\cos\phi \ \sin\theta, \sin\phi \ \sin\theta, \cos\theta)$$

And, by varying the random values (ζ_1, ζ_2), obtain a set of directions that we can use for rendering. The generated points for the Doge map can be seen in Figure 4.10.

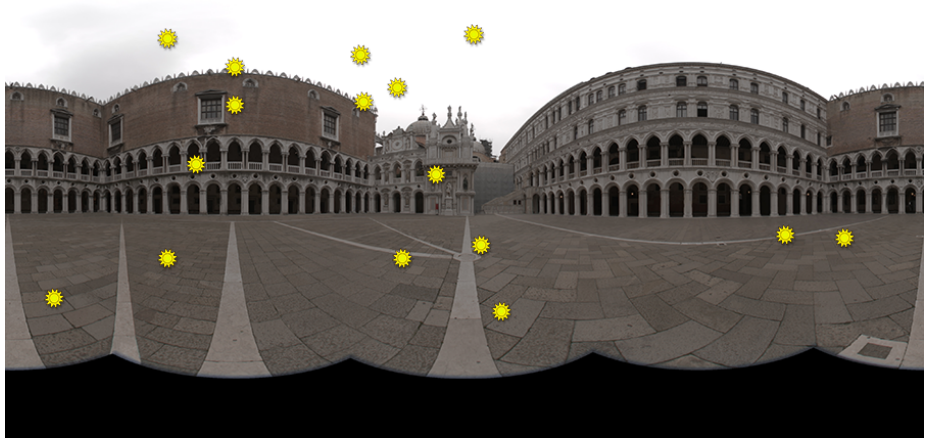


Figure 4.10: Position of calculated lights for the doge map.

Chapter 5

Implementation

In this chapter, we will introduce the implementation of our method. We will focus on a GPU-oriented implementation of the method presented in Chapter 4. First, we will briefly introduce the environment in which we are operating. Then, we will give a general outline of our algorithm. Then, we will discuss in more detail some of the ideas introduced in the outline. After this discussion, we will discuss some defects and artifacts in our algorithm that arose during the implementation, and how we have solved them. Finally, we will extend our implementation to different kind of lights. Finally, we will discuss our implementation.

5.1 Environment

In order to justify some of the choices and the code parts that will be introduced in this section, we will first introduce the environment of our implementation. The method we are going to discuss was made using the OpenGL API, version 4.3 (released in August 2012), a multi-platform API used for rendering 2D and 3D accelerated graphics. With the OpenGL API comes GLSL, the *OpenGL Shading Language*, used for writing pieces of code to be run on the GPU, called *shaders*. Our method uses some advanced features of OpenGL 4.3, so it is not

immediately portable to previous generation hardware, and runs only on high-end modern GPUs. On the CPU side, we use an extended framework based on Qt, a C++ library that allows to create a OpenGL context and graphical interfaces in an easy way.

5.2 Algorithm overview

By keeping the limitations presented in Chapter 1 in mind, we introduce our algorithm. The algorithm is inspired by *translucent shadow maps* [Dachsbacher and Stamminger, 2003], that we presented in Chapter 2. The general idea is to first render the scene from the light point of view, then place the disk we discussed in the previous Chapter (Section 4.1) directly on the generated texture, storing the result in a radiance map. We use many directions in order to capture all the sides of the object. Finally, we sample from the obtained radiance map for the final rendering.

In this section, we will assume to only have one directional light with radiance L_d and direction $\vec{\omega}_d$, with one not-deformable object in the scene. We will discuss later how to extend the method to multiple light sources.

Step 1 - Light buffer
In the first step, positions and normals of the object are rendered into a texture from the light point of view. We will refer to the resulting texture as the *light map*. As in standard shadow mapping we create and store a matrix to convert between world space and texture light space. Depth testing in this step is enabled. More details on the implementation of a render to texture are presented in Section 5.3.1.

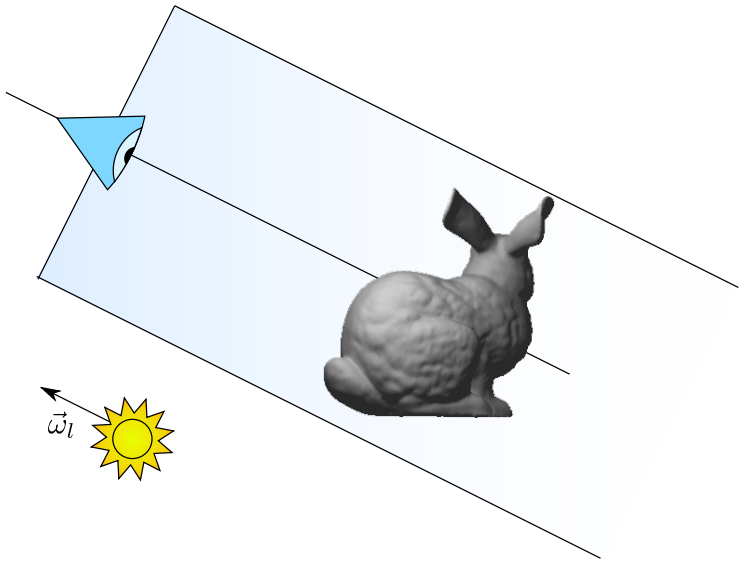
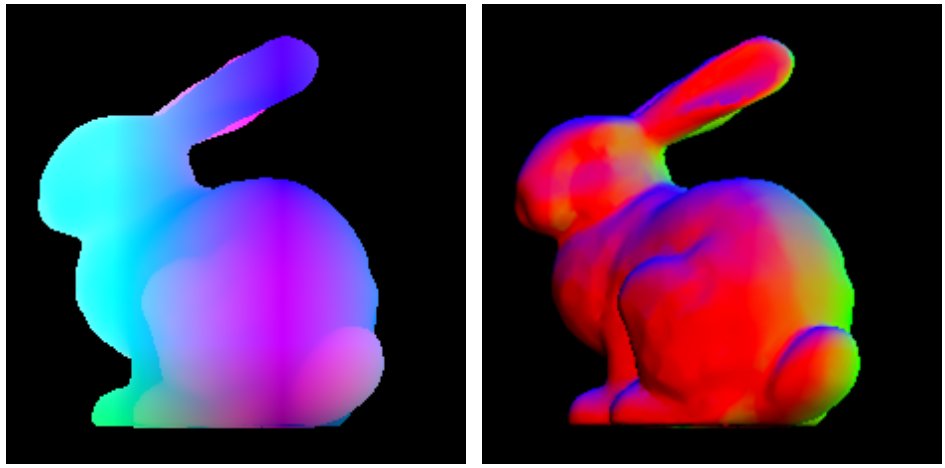


Figure 5.1: Render to G-buffer. Note that the frustum and the light direction are aligned.



(a) Vertex buffer **(b)** Normal buffer

Figure 5.2: State of the vertex and normal buffer after rendering from a directional light. The model used was the Stanford bunny from the Standford 3D Scanning repository.

Step 2 - Render to radiance map
In the second step, we render the object from K different directions into another texture, called the *radiance map*. The radiance map is organized as a layered texture, where each layer represents a direction. More details on render to texture and layered textures are presented in sections 5.3.1 and 5.3.2. The points on which to place the cameras are chosen randomly, distributed on a sphere (see Section 5.3.4 for the details). The cameras point towards the center of the object bounding box.

On this step, for each pixel that corresponds to an exiting point $\mathbf{x}_o$ the shader samples N points from the texture rendered in the previous step. If the sampled point is valid, it is then used to calculate the BSSRDF and accumulate it in the resulting radiance map. So, this step calculates the following:

$$R^k(\mathbf{x}_o) = L_d \sum_{i=1}^{N} S(\mathbf{x}_i^k, \vec{\omega}_l, \mathbf{x}_o, \vec{\omega}_o) \exp\left(\sigma_{tr} r_i^k\right), \quad k \in [0, K-1] \tag{5.1}$$

Where we recall that we have introduced an exponential term in order to compensate for the exponential displacement of the sampling pattern. The generation of the sampling pattern is described in Section 5.3.4.1 in more detail. We introduced a k parameter, that represents the current direction we are rendering to. We can see how we are rendering a point from one of the considered directions in Figure 5.4. Also in this case, the texture space - world space conversion matrices are stored and prepared to be reused in step 3, where we will combine the results.

We can appreciate that rendering the light from the camera point of view comes with two important advantages:

- If the disk is placed in texture space, it is automatically oriented towards the light direction, that is $\vec{\omega}_d = \vec{\omega}_l$.
- The light renders in the texture only the points directly visible from it, that are also the only points directly lit. In addition, if we sample the light map on any point, we get the corresponding vertex that is closest to the light.

This two factors allows us to sample the most optimal point and direction in where to place the disk, as we described it in Section 4.1.

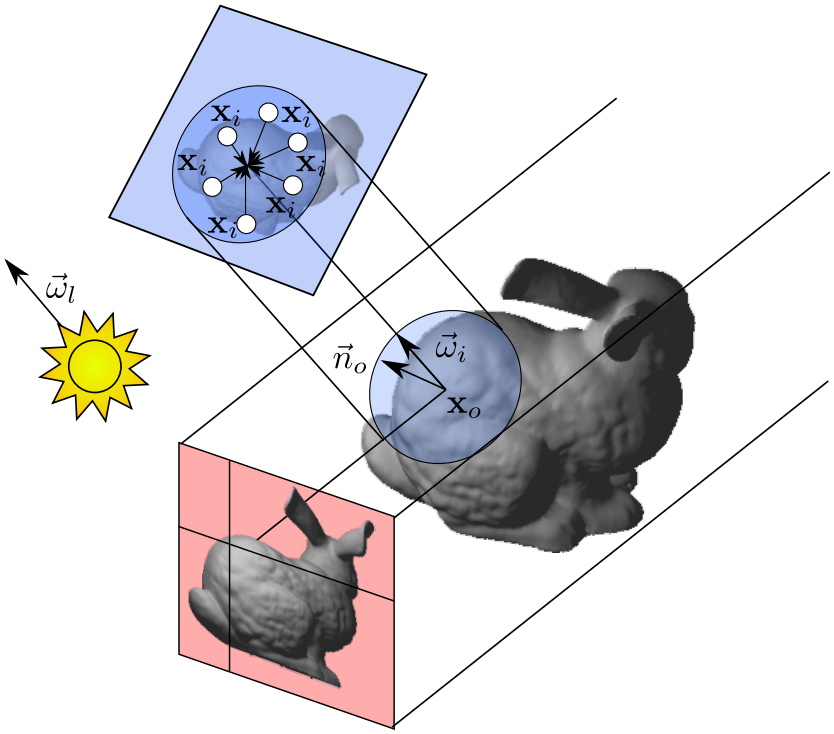


Figure 5.3: Render to the radiance map. When we render the point $\mathbf{x}_o$, the position in the light map is calculated and the values $\mathbf{x}_i$ in the samples are calculated and summed over.

In addition, on each layer, we accumulate the result over different frames:

$$\tilde{R}^{t,k} = \sum_{i=0}^{t-1} R^{i,k} + R^{t,k} = \tilde{R}^{t-1,k} + R^{t,k}$$

$\tilde{R}$ here represents the value that is stored in the radiance map. In equation 5.1, we omitted the dependence from time for the sake of clarity. We need an accumulation process in order to deal with the fact that the result from the previous computation are not reaching a satisfying result within one frame, so they need to be accumulated over a period of T frames in order to reach an appreciable result. In order to do this, the sampled points need to change on different frames (in Section 5.3.4 we give more details on the process).

Naturally, the accumulation process works only if the scene does not change. A change can be a relative change of positions between the points on the model and the light, so if the model gets translated, rotated or scaled, the accumulated

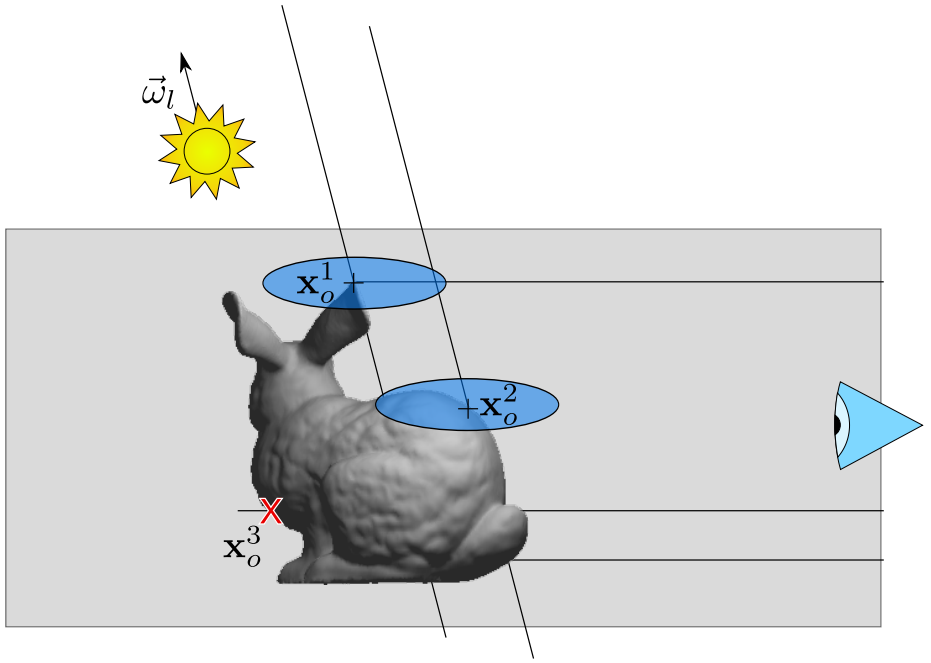


Figure 5.4: Render to cubemap, side view. The gray area represents the frustum of the current direction's camera (orthographic). We have three different cases of a point on the surface: $\mathbf{x}_o^1$ is visible from both the light and the camera, so the disk is placed on it. $\mathbf{x}_o^2$ is visible from the camera but not from the light, so the disk is placed in the closest position to the light. $\mathbf{x}_o^3$ is not visible from the camera, so it is discarded.

result has to be discarded and the accumulation started all over. The directional cameras are locked with the model, so the pixels are always aligned. This is why in equation 5.1 the dependence from time of the point $\mathbf{x}_o$ and $\vec{\omega}_o$ have been dropped.

Step 3 - Combination

In this step, we render our model. Using the matrices and the textures prepared in the previous step, for each fragment on the surface we sample all the layers in the texture as illustrated in Figure 5.5. In order to do this, we need also to sample the depth map generated in the previous step. We can define this sampling as a visibility function to test if a point $\mathbf{x}$ belongs to layer k:

$$V^k(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is visible from the } k\text{th camera} \\ 0 & \text{otherwise} \end{cases}$$

The function V is calculated by sampling the k-th layer depth map in a way

similar to shadow mapping (details in Section 5.3.5). Given this function, we can simply represent the outgoing radiance by simply averaging the summation over the K layers:

$$L_{SS}^{t}(\mathbf{x}) = \frac{A_c}{Nt} \frac{\sum_{k=0}^{K-1} V^k(\mathbf{x}) \tilde{R}^{t,k}(\mathbf{x})}{\sum_{k=0}^{K-1} V^k(\mathbf{x})}$$

The first factor $\frac{1}{t}$ is to average over the number of frames, while the second is the average area of a sample in the circle $\frac{A_c}{N}$, that is necessary to complete equation 4.2. We note that we moved all the layer-independent computation into the final computation step, in order to save as much performance as possible.

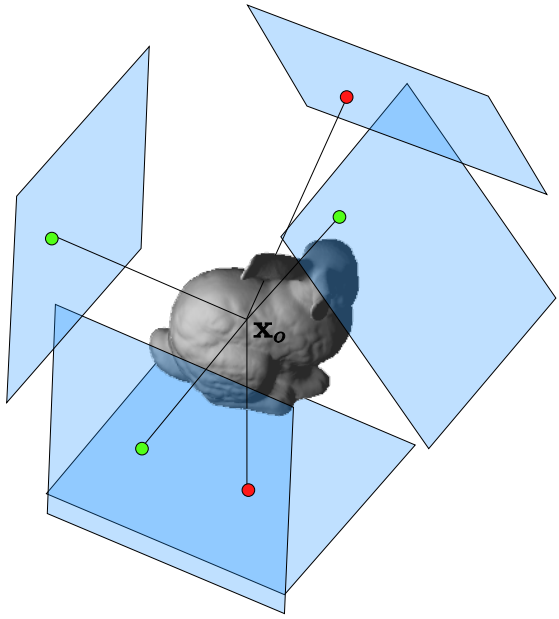


Figure 5.5: Final combination step. The blue quads indicates each one of the radiance map layers, as seen from their direction. We can see that in the point $\mathbf{x}_o$ the contribution from three faces (green dots) is considered. For the remaining two faces (red dots), the contribution is not considered as the point is not visible.

We are not done yet, as for now we have computed the radiance only deriving from subsurface scattering. For finally describing the illumination of our scene, we need also to include a factor based on the surface reflection. Since the subsurface scattering radiance is already multiplied by a transmittance Fresnel

term (in the BSSRDF equation in 3.9) $T(\eta, \vec{\omega}_o)$ we the reflection color multiplied by the converse transmission term $1 - T(\eta, \vec{\omega}_o)$:

$$L^t(\mathbf{x}, \vec{\omega}_o) = L^t_{SS}(\mathbf{x}) + (1 - T(\eta, \vec{\omega}_o))L_i(\mathbf{x}, \vec{\omega}_o - 2(\vec{\omega}_o \cdot \vec{n})\vec{n})$$

L_i can be the radiance coming from other objects or by an environment map. After this, we just need to perform gamma correction in order to get the final result. Given the gamma coefficient γ, we perform gamma correction by:

$$L^t_{gamma}(\mathbf{x}, \vec{\omega}_o, \gamma) = L^t(\mathbf{x}, \vec{\omega}_o)^{\frac{1}{\gamma}}$$

And, adter this, we can finally send the radiance to the output device.

Figure 5.6: Final Result of the method after 100 frames simulating a potato bunny of dimensions $\approx 30cm$. All the results presented in this report are gamma corrected with $\gamma = 1.8$.

Figure 5.7: Final Result of the method after 100 frames simulating a backlit bunny made of beer, dimensions $\approx 30cm$.

5.3 Implementation details

In this section, we will further expand the overview given in the previous section by adding further details. We organized this section in topics, in each one of them expanding a particular aspect of our algorithm. Each point clarifies with examples one technique or approach used in the method. The code given on each example does not necessarily come from our implementation, but it can be simplified for illustration purposes.

5.3.1 Render-to-texture

In a graphics API, and more specifically in OpenGL, all the output from a final shader stage is usually sent to the display device in order to be displayed on

the screen. However, it is possible to redirect the output into another memory area of the GPU and reuse it for further computations. This allows to create complicated rendering techniques such as the one described in this report.

In OpenGL, is possible to redirect the output to a texture object. We can do this through a so-called *framebuffer object* (FBO). A FBO is a collection of objects that allows off-screen rendering. A FBO has *attachment points*, to which we can attach textures. A texture can be attached to one of the output color channels of the fragment shader (`GL_COLOR_ATTACHMENT0`, `GL_COLOR_ATTACHMENT1`, ...), to the depth buffer output (`GL_DEPTH_ATTACHMENT`) or to the stencil buffer output (`GL_STENCIL_ATTACHMENT`).

The connection between the framebuffer and the texture can be set in a initialization step (listing 5.1). Then, by simply binding the FBO (listing 5.2) we redirect our screen output to the desired texture.

```
GLuint tex,fbo;
// Generating texture
glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);
[...] // setting up texture parameters...
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, size, size, 0, GL_RGBA, GL_FLOAT↩
    , 0);

glGenFramebuffers(1,&fbo);

// connecting current fbo and texture
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, ↩
    GL_TEXTURE_2D, tex, 0);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0); //Binding back main framebuffer
```

Listing 5.1: Render to texture example, initilalization phase. Note the call to `glFramebufferTexture2D`

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
GLenum buffers[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, buffers);
[...] // draw model
```

Listing 5.2: Render to texture example, rendering phase. Since we have not configured and FBO for depth and stencil buffers, depth testing and stencil should be disabled at this point.

5.3.2 Layered rendering

Layered rendering is a special feature used to render to a special type of texture called *layered texture*. Let us take as an example the step 2 of our algorithm, where we need to render the object from K different directions. A first approach to this would be to create K 2D textures of type `GL_TEXTURE_2D`, and perform K draw calls, rebinding the texture on the current FBO each time. OpenGL, however, provides a way to do this with a single draw call, potentially reducing the rendering costs due to context switching.

We will use the OpenGL provided type `GL_TEXTURE_2D_ARRAY`, that we initialize in the usual way, noting that an array texture should be allocated as a 3D texture. When binding it to an FBO, we use the generic `glBindFramebufferTexture`:

```
GLuint fbo, arraytex;
glGenFramebuffers(1,&fbo);
glGenTextures(1, &arraytex);

glBindTexture(GL_TEXTURE_2D_ARRAY, arraytex);
[...] // setting up texture parameters, omitted
glTexStorage3D(GL_TEXTURE_2D_ARRAY, levels, GL_RGBA32F, size, size,layers);

glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
glFramebufferTexture(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, arraytex, ↩
    0);
```

Listing 5.3: Initializing array texture. Note that the number of layers is passed to the `glTexStorage3D` command.

In order to render to a layered texture, we need then to introduce a geometry shader. In our example, the difference between each layer is basically a different view matrix. So, we move the computation of the position, usually left to the vertex shader, to the geometry shader. We first introduce the code:

```
#version 430
#define DIRECTIONS 16
layout(triangles) in;
layout(triangle_strip, max_vertices = 60) out;

uniform mat4 P;
uniform mat4 viewMatrices[DIRECTIONS];

void main(void)
{
    for(int i = 0; i < DIRECTIONS; i++)
    {
        gl_Layer = i;

        for(int k = 0; k < 3; k++)
        {
            vec4 v = gl_in[k].gl_Position;
```

```
            gl_Position = P * viewMatrices[i] * v;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Listing 5.4: Geometry shader for layered rendering. The multiplication by the model matrix of vertex v is performed in the vertex shader (not shown).

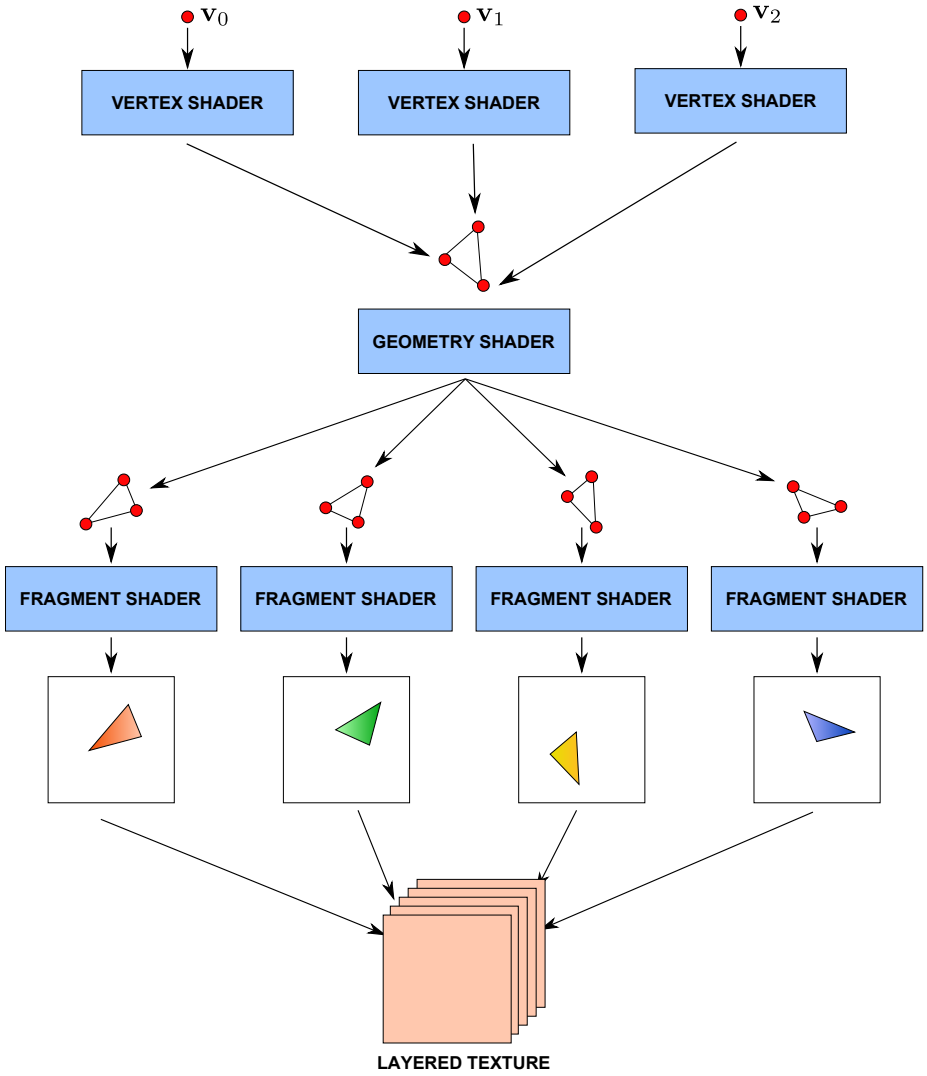


Figure 5.8: Diagram that illustrates layered rendering.

As we can see, we duplicate each one of the incoming triangles and then output it multiplied by a different view matrix. the `EndPrimitive()` function ensures that the output triangles in the final triangle strip are separated. In order to render each triangle to a different layer, we need to set the build-in variable `gl_Layer` to the layer we want to render before emitting the triangle. The whole process is illustrated in Figure 5.8.

5.3.3 Accumulation buffers

In the second step of our method we said we would like to accumulate the result of the computation, in order to progressively update the result over different frames. In order to do this, we encounter an obstacle: we would like to render to the currently bound texture, but at the same time we need to read the previous value stored on the texture. Unfortunately, the OpenGL Specification [Segal and Akeley, 2012] advices against reading from the same texture we are rendering to. This is made to avoid a situation called *feedback loop*. In fact, we cannot be sure of the results of what is stored in any pixel of the texture while we are rendering to it.

There are many possible solutions to the problem. The first approach is to rely on the driver implementation: some drivers, in fact, allow to render to the same texture we are bound to, under some conditions. However, if we want a general method that works over all platforms, this is not a viable path. The second solution is to use image textures, a new type introduced in OpenGL 4.2 that allows explicit load-store of values, as well as new special constructs for GPU memory management (atomic operations and memory barriers). Though the usage of these features is appealing, their performance is generally poor compared to a framebuffer-based implementation.

The final approach, and the one we use in our implementation, is to ping-pong between the two textures. The idea is to sacrifice memory space by employing two textures T_1 and T_2. In the first frame, we render to the first texture T_1. In the second frame, we use T_2 as a render target, and we sample T_1 and add it to the computed result. In the third frame, we render to T_1 and sample from T_2, and so on. From this alternance between the textures comes the name ping-pong. A minimal example of ping-ponging using a 2D texture is shown in listing 5.5.

```
// A global frame variable is initialized in order to keep track of the ↩
    current frame

GLuint fbo, tex1, tex2;
// Creating FBO, initializing textures and texture parameters.
// Also binding shader with glUseProgram in order to bind texture uniforms
[...]

GLuint tex_from, tex_to;

tex_from = (frame % 2 == 0)? tex1 : tex2;
tex_to = (frame % 2 == 0)? tex2 : tex1;

glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
glFramebufferTexture(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, tex_to, 0);
glDrawBuffer(GL_COLOR_ATTACHMENT0);

GLint location = glGetUniformLocation("source_texture");
glUniform1i(location, 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tex_from);

// more uniforms and rendering commands.
[...]
frame ++;
```

Listing 5.5: Minimal example of ping-pong textures.

5.3.4 Generation of uniformly distributed points

In our method we have at least twice the necessity to generate uniformly distributed points either on a disc or on a sphere. To do this, we employ a particular sequence of pseudo-random numbers, called *Halton points* [Halton, 1964]. We explain briefly the ideas behind the sequence. For a more mathematical complete discussion on its properties of pseudo randomness, see Niederreiter [1992].

First, given a prime number p and a non-negative integer n, we can express it in base p as:

$$n = a_0 + a_1 p + a_2 p^2 + ... + a_r p^r$$

Where $a_i \in [0, p-1]$. We now define a van der Corput sequence $\Phi_p(n)$ as:

$$\Phi_p(n) = \sum_{i=0}^{r} \frac{a_i}{p^{i+1}} = \frac{a_0}{p} + ... + \frac{a_r}{p^r}$$

This sequence, given the fact that is based on prime points, automatically assumes good qualities of randomness. In addition, the function is already normalized in the range $[0, 1)$. We define an Halton point as the combination of two Van der Corput sequences:

$$H_{p_1,p_2}(n) = (\Phi_{p_1}(n), \Phi_{p_2}(n))$$

Where p_1 and p_2 are two prime numbers, with $p_1 < p_2$. Usually, $(p_1, p_2) = (2, 3)$ gives good results. All Halton points belong to the region of space $[0, 1] \times [0, 1]$. In order to obtain a sampling of Halton points on a sphere, we convert them using an area-preserving cartesian-to-spherical coordinates formula:

$$H_{p_1,p_2}(n) = (\Phi_{p_1}(n), \Phi_{p_2}(n)) \rightarrow (s, t) \Rightarrow$$
$$\Rightarrow H^{sphere}_{p_1,p_2}(n) = (\sqrt{1-(2t-1)^2}\cos(2\pi s), \sqrt{1-(2t-1)^2}\sin(2\pi s), 2t-1)$$

And, to get the point on a disc, we simply take the point on a sphere and project it. In practice, we put the third coordinate to zero:

$$H_{p_1,p_2}(n) = (\Phi_{p_1}(n), \Phi_{p_2}(n)) \rightarrow (s, t) \Rightarrow$$
$$\Rightarrow H^{disc}_{p_1,p_2}(n) = (\sqrt{1-(2t-1)^2}\cos(2\pi s), \sqrt{1-(2t-1)^2}\sin(2\pi s))$$

Wong et al. [1997] provide an introduction to Halton points, as well as describing an implementation to generate a point on a Van der Corput sequence. We implemented their pseudo-code in C++ as follows:

```
float vanDerCorputPoint(int n, int basis)
{
    int kp = n;
    float pp = (float)basis;
    float phi = 0.0f;
    while(kp > 0)
    {
        int a = kp % basis;
        phi = phi + a / pp;
        kp = int(kp / basis);
        pp = pp * basis;
    }
    return phi;
}
```

Listing 5.6: Generating the p-adic Van der Corput point.

5.3.4.1 Exponentially biased points

In our algorithm, in order to obtain a better sampling, we need to have an exponentially biased distribution of points, as described in Section 4.2. To obtain the disc, we employ a technique called rejection sampling. The general idea is to generate a the sequence of Halton points, then calculate their radius r and calculate its probability distribution function using a value σ^*. Then, we use the following acceptance criterion:

$$e^{-\sigma^* r} > \zeta$$

Where $\zeta \in [0, 1)$ is a pseudo-randomly generated number. We can see that if the point is close to the center ($r \rightarrow 0$), $e^{-\sigma^* r} \approx 1$ and so the point is more probable

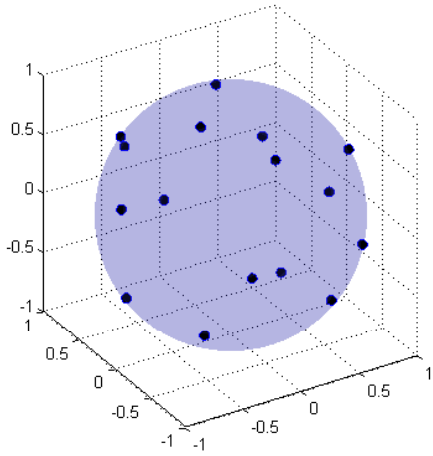


Figure 5.9: Positions of the first 16 cameras generated using Halton points on a sphere.

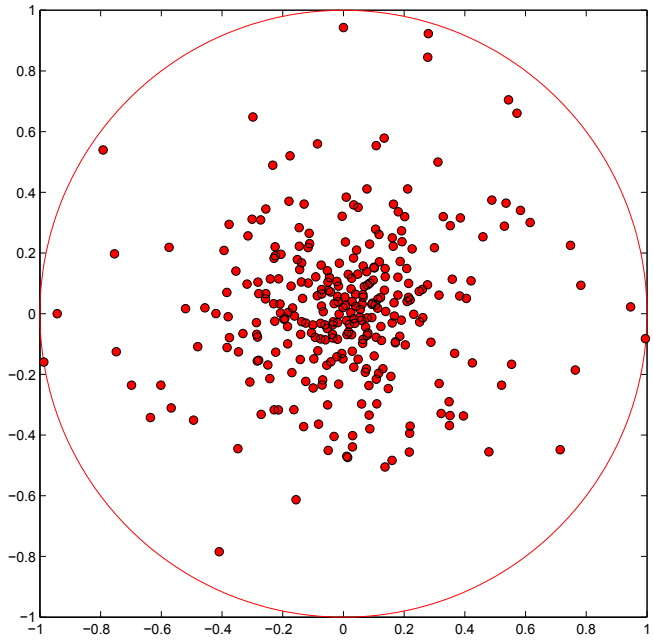


Figure 5.10: Exponentially biased Halton points on a disk. The maximum radius is 1, and $M = 300$.

to be accepted. On the other hand, if the point is far from the center of the disc ($r \to +\infty$), $e^{-\sigma^* r} \approx 0$ and so the point is less probable to be accepted. The code for generating a vector of accepted points is reported in listing 5.7.

```
void planeHaltonCircleRejectionExponential(std::vector<Vec2f> &result, int ↩
     m, float sigma_tr, float radius)
{
   uint accepted = 0;

   gel_srand(0); // Setting the seed of the random number
   int i = 1;
   while(accepted < m)
   {
        Vec2f point = haltonPointCircle(i, 2, 3);
        float rad = point.length() * radius;
        float expon = exp(-sigma_tr * rad);
        float zeta = gel_rand() / ((float)(GEL_RAND_MAX));
        if(zeta < expon)
        {
            result.push_back(radius * point);
            accepted++;
        }
        i++;
   }
}
```

Listing 5.7: Generation by rejection of a exponentially distributed disc. The function generates M points with distribution $e^{-\sigma_{tr} r}$, and where `radius` is the maximum final radius of the points.

In our algorithm, we generate a number of samples M that is fixed, with a radius equal to the size of the bounding box of the model. Then in the actual calculation only the N points closest to $\mathbf{x}_o$ are actually used, allowing us to tweak the performance and the result. Moreover, the value of the exponent σ^* we pass to this method is related to σ_{tr} (that is a vector, since the transmission rates are spectral), to account for the material scattering properties. We calculate σ^* with the formula:

$$\sigma^* = \frac{\min(\sigma_{tr,x}, \sigma_{tr,y}, \sigma_{tr,z})}{q}$$

Where q is a parameter tweakable from the user. We will justify why we use the minimum transmission coefficient has to be used in section 6.2.1. The idea is that the user can modify the distribution of the points, making them span a wider area (that is for $q > 1$) without increasing the number of samples N. In fact, a bigger N implies a worse performance, while a bigger q requires only to recompute the points. We will discuss the results for varying values of q in Chapter 6.

5.3.5 Shadow mapping

Shadow mapping is a common technique used in modern real-time graphics [Everitt et al., 2003, Segal et al., 1992, Williams, 1978]. The idea behind it is to render an object from a light's point of view, and then use the generated depth information in order to decide if a point is shadowed or not. First of all, we convert the point into the light camera space, using a special space conversion matrix. After this, if we have a point $\mathbf{p} = (p_x, p_y, p_z)$, we compare p_z to the texture T sampled in the point (p_x, p_y):

$$p_z > T(p_x, p_y) \tag{5.2}$$

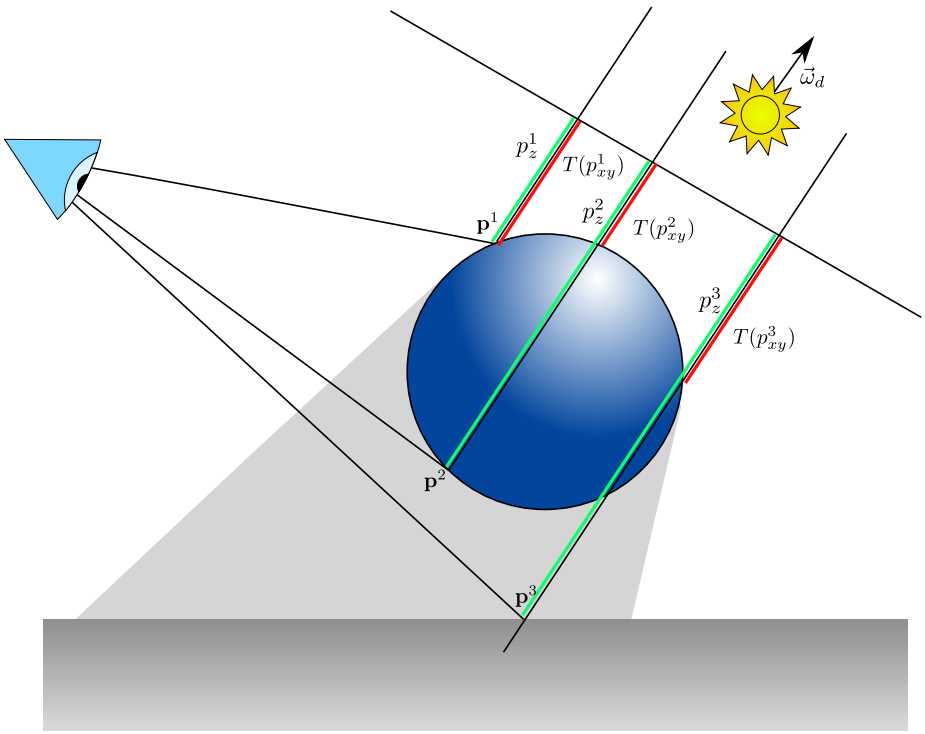


Figure 5.11: Shadow mapping. We see the different results for three points $\mathbf{p^1}$, $\mathbf{p^2}$ and $\mathbf{p^3}$. The red length represents the value sampled from the depth texture($T(p_x, p_y)$), while the green length represents the value we compare against (p_z). When p_z is less than $T(P_{xy})$, as in $\mathbf{p_1}$, the point is not shadowed. In the other cases, $\mathbf{p_2}$ and $\mathbf{p_3}$, $p_z > T(P_{xy})$and the point is not visible.

If the above condition is verified, it means that the current point is beneath the point visible from the light and then it should be shadowed. The matrix L used to convert a point from world space to texture space is the following, using the matrix definitions and notation in Section A:

$$L = T\left(\frac{1}{2}\right) \cdot S\left(\frac{1}{2}\right) \cdot P \cdot V$$

where P and V are the projection and view matrix we use to render with the light. The first two matrices are necessary to convert between clip and texture coordinates, as clip coordinates are in the range $[-1, 1] \times [-1, 1]$, while texture coordinates are in the range $[0, 1] \times [0, 1]$. The process is illustrated in Figure 5.11.

In our algorithm, we use the ideas behind shadow mapping in two different occasions. The first occasion is to get the points x_i in step 2 from the texture generated in step 1, where we use the matrix L in order to convert the world point $\mathbf{x}_o$ to $\mathbf{x}_d$, the center of the disc in texture space.

The second occasion is in the final combination step, where we use also shadow mapping in order to compute the visibility function $V^k(\mathbf{x})$, and to sample the radiance map. Technically, the "light cameras" in this case are the directional cameras from where we render the scene, but the concept is the same. We can see how we sample the shadow map in the final combination shader in the following listing:

```
#version 430
uniform sampler2DArrayShadow depthMap;
uniform mat4 cameraMatrices[DIRECTIONS];

float sample_shadow_map(vec3 world_pos, int layer)
{
    vec4 light_pos = cameraMatrices[layer] * vec4(world_pos,1.0f);
    light_pos.z -= shadow_bias; //bias to avoid shadow acne
    if(light_pos.x < 0.0 || light_pos.x > 1.0) return 1.0;
    if(light_pos.y < 0.0 || light_pos.y > 1.0) return 1.0;
    return texture(depthMap,vec4(light_pos.x,light_pos.y,layer,light_pos.z)↩
        ).r;
}
[...] //shader code
```

Listing 5.8: Sampling of the shadow map texture in step 3 of our method.

In the above code, `cameraMatrices[layer]` corresponds to the L matrix. We use a special type of sampler, `sampler2DArrayShadow`, that makes something more than its equivalent `sampler2DArray`. The latter, in fact, accepts a `vec3`, and simply retrieves the value in the texture. The former, on the other hand, accepts an extra parameter z_{camera} (which is `light_pos.z`), and compares it

to the value stored in the depth texture z_{tex}, performing the test in equation 5.2. If the depth of the point is less than the depth stored in the texture ($z_{camera} < z_{tex}$), 1 is returned as the point is closer to the directional camera, and thus visible. On the other hand, if $z_{camera} \geq z_{tex}$, it means the point is under the surface, and thus not visible, and 0 is returned.

To configure the `sampler2DArrayShadow` texture, we need to specify some extra parameters during the depth texture initialization:

```
glBindTexture(GL_TEXTURE_2D_ARRAY, depthtex);
glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_COMPARE_MODE, ↩
    GL_COMPARE_REF_TO_TEXTURE);
glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_COMPARE_FUNC, GL_LESS);
glTexStorage3D( GL_TEXTURE_2D_ARRAY, 1, GL_DEPTH_COMPONENT32F, size, size, ↩
    layers);
```

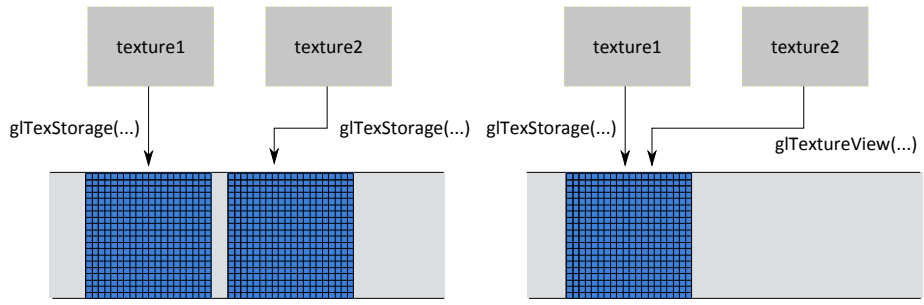
Listing 5.9: Configuration of a shadow map depth texture.

We can see that we specify two extra parameters: `GL_TEXTURE_COMPARE_MODE`, once set to `GL_COMPARE_REF_TO_TEXTURE`, means that sampling the depth texture in a shader will give a value based on the comparison between an extra value z and the depth value d in the texture. The second parameter `GL_TEXTURE_COMPARE_FUNC` specifies how to compare the values: `GL_LESS` means that 1 is returned if $z < d$, and zero is returned otherwise.

5.3.6 Memory layout

In this section, we briefly describe how we allocate memory in our method. The great advantage of using OpenGL 4.3 is that we can use *texture views* [Segal and Akeley, 2012]. Texture views are a way to create allocate storage for a texture in OpenGL, but on the opposite hand of `glTexImage*D` or the `glTexStorage*D` families of functions they allow to use another texture's storage space. To create a texture view, we use `glTextureView`. We compare the standard way of allocating texture with our without texture view in Figure 5.12.

In our method, this comes to great advantages in two areas. First of all, since the depth texture of the radiance map is not accumulated, we can make the two radiance maps that are using ping-ponging share the same depth map. In addition, we can render directly between two mipmap levels of the same texture, something that we will need later to generate a blurred version of our texture.



(a) Without texture views **(b)** With texture views

Figure 5.12: Diagram illustrating texture views. On the top part of the picture the texture ids on the GPU, on the bottom the state of the memory. In the first case, we create two memory areas on the GPU using `glTexStorage`, in the second case we make a single call to `glTexStorage` and then a second call to `glTextureView` to make them share the same storage space.

A complete example memory layout for our algorithm is reported in table 5.1. As we will see, we will account for the contribution from multiple lights as more layers on the light map texture. More textures are loaded in order to make the method work, but their contribution to the overall memory consumption is negligible.

If L is the number of lights, W_l the size of the light map, K the number of directions and W_s the size of the radiance map, we can obtain a direct formula to calculate the memory consumption in bytes:

$$4\left\{\left[(4+4)\,\frac{4}{3}+1\right]KW_s^2+[3+3+1]\,LW_l^2\right\}$$

The 4/3 factor account for the extra space reserved for mipmaps. The 4 factor at the beginning is because each channel has 32 bits, that equal to 4 bytes.

	Ch.	Bits	Width	Height	Depth	Size (MB)
Light map (vertex)	3	32	512	512	1	3
Light map (normals)	3	32	512	512	1	3
Light map (depth)	1	32	512	512	1	1
Radiance map 1 (color)	4	32	512	512	16	64
Mipmaps 1	4	32	-	-	-	21
Radiance map 2 (color)	4	32	512	512	16	64
Mipmaps 2	4	32	-	-	-	21
Radiance map (depth)	1	32	512	512	16	16
						193 MB

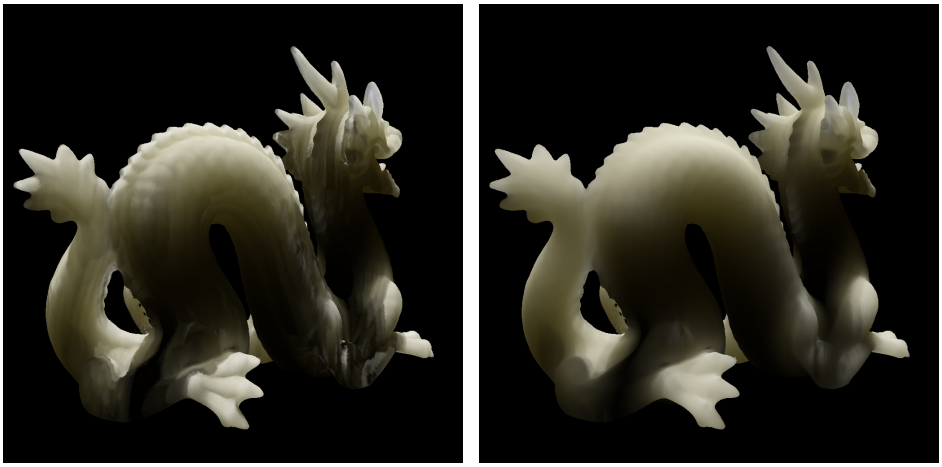
Table 5.1: Memory occupation of our method, for one light. Mipmaps are accounted for one additional third on the size of the radiance map. "Ch.' is the number of channels in the texture, "Bits' is the number of bits per channel.

5.4 Caveats

The algorithm described so far, if implemented *as is*, unfortunately does not give a result that we can appreciate. In order to obtain the desired result, some extra corrections are necessary, and we are going to describe them in this section.

5.4.1 Random rotation of samples

As we note in equation 5.1, we never specified how the points in the samples are processed before actually being used in the calculation. This can lead to the assumption that the same disc pattern is used on every point $\mathbf{x}_o$ of the model. This causes a problem because this generates banding artifacts, as we can see in Figure 5.13. In order to avoid the artifacts, in exchange for noise, we randomly rotate the pattern for each pixel of the radiance map. Given that we are able to generate a random point r, we can take each one of the samples on the disk



(a) Constant rotation **(b)** Random rotation

Figure 5.13: White grapefruit juice dragon. The only difference in the two images is that the samples on the right figure are randomly rotated.

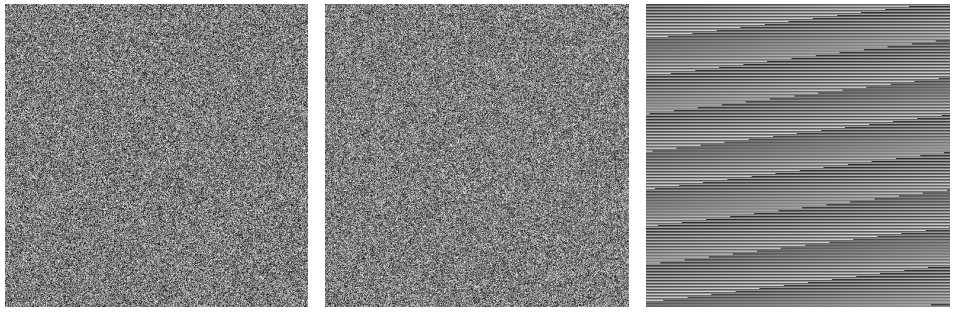
$\mathbf{d}_i$ and rotate it in order to obtain the new sample $\mathbf{d}_i^*$:

$$\begin{aligned} \theta &= 2\pi r \\ \mathbf{d}_i^* &= \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \mathbf{d}_i \end{aligned} \tag{5.3}$$

Even if noisy, we eliminate the artifacts from the result. We will see how to reduce the noise using mipmaps in Section 5.4.2. As we observed before in the overview, we also need the result to evolve on a time basis, in order not to compute the same results on every frame. Recalling that the current frame is t and the maximum amount of frames before we stop the computations is T, we make our computation evolve using the following θ_t in equation 5.3:

$$\theta_t = 2\pi \left(r + \frac{t}{T} \right)$$

This causes a progressive rotation of the disc around the point over time. However, we need still to specify how to calculate r. As a function, $r = (x, y, l)$ needs to depend on the fragment coordinates (x, y), as well as from the current layer l (to avoid two layers make the same computation). Since we are on the GPU, we cannot use a built-in random function, so we need either to load the random points from the CPU as a separate texture or to generate them on-the-fly. For the latter technique, we tried a sine based generator, a linear shift (LSR) random generator and a congruential linear generator (LCG), all in



(a) Sine noise **(b)** LSR noise **(c)** LCG noise

Figure 5.14: Three examples of deterministic noise generation on GPU. While the sine and linear shift noises give a uniform and pleasant noise, the linear congruential generator has a periodicity.

listing 5.10. The input point that was given was in both cases $(l \cdot x, l \cdot y)$. We can see some results in Figure 5.14. The sine based generator theoretically gives the best result, but the linear shift generator gives an indistinguishable result with better performance.

```
highp float noise_sine(vec2 co)
{
    highp float a = 12.9898;
    highp float b = 78.233;
    highp float c = 43758.5453;
    highp float dt = dot(co.xy,vec2(a,b));
    highp float sn = mod(dt,3.14);
    return fract(sin(sn) * c);
}

highp float noise_lsrg(vec2 co, int size)
{
    int n = co.x + size * co.y;
    n = (n << 13) ^ n;
    int s = (n * (n*n*15731+789221) + 1376312589) & 0x7fffffff;
    return s / (4294967296.0f) * 2;
}

highp float noise_lcg(vec2 co, int size)
{
    int n = co.x + size * co.y;
    return float((n * 1664525 + 1013904223) % 4294967295) / 4294967296.0f;
}
```

Listing 5.10: Three GLSL functions to generate random points on the GPU. The extra size parameter is the height of one layer of the radiance map texture.

5.4.2 Mipmap generation

The randomization we performed in the previous step comes with a drawback, i.e. we increase the noisiness of the result, especially in the initial steps. Once we reach convergence, the noisiness slowly disappears. In order to improve our method, we need to introduce a way to take the intermediate result and make it as close as possible to the final result at convergence. Our approach was to use a filter to blur the result multiple times, storing the results in the mipmaps of the radiance map generated in step 2 of our method. This results in an additional step between step 2 and step 3, where the mipmaps are calculated.

The filter that gave us the most promising results is the bilinear filter with the shape shown in Figure 5.15. The filter uses the alpha value of the texture as a weight for the sampled color, so the filter maintain the edges of the texture.

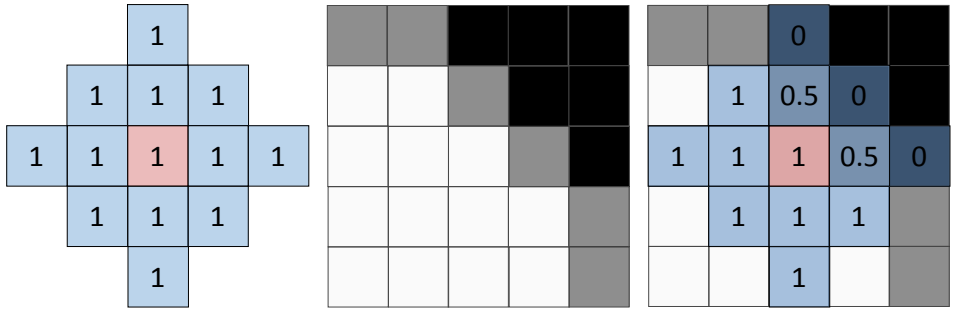


Figure 5.15: Alpha-weighted bilinear filter. In 5.15a we have the shape of the filter. In 5.15b we have the alpha channel of an example image, to which we show the weights used in the filters in Figure 5.15c, based on the alpha channel of the picture.

On the implementation level, the mipmaps need to be rendered one after the other, as we need the result of the first computation in order to perform the second. So, we use always the same shader, reported in listing 5.11, to filter between two textures. In the first step, we bind layer zero (where the result of the rendering of step 2 is stored) as source and layer 1 as destination. Then, we bind layer 1 as source and layer 2 as destination, and so on. To perform the filtering, we render a full screen quad.

However, OpenGL does not allow to bind the same image both as a source and as a destination, even if the mipmap levels are different. So, in order to overcome this difficulty, we use texture views as we described them in 5.3.6. We create a new texture for the mipmaps, but then we configure it to use the

storage reserved to the mipmaps of the radiance map. So, we can bind it to the FBO as it was a completely different texture, but then once we render the result will be rendered in the right memory location.

```
#version 430
in vec3 _tex;
uniform sampler2DArray source;

out vec4 fragColor;

uniform float texStep;
uniform int scaling;

void main(void)
{
  int layer = gl_Layer;
  float t_step = texStep * 0.5 * scaling;
  vec4 c0  = texture(source,vec3(_tex.xy,layer));
  vec4 c1  = texture(source,vec3(_tex.xy + vec2(t_step,0.0f),layer));
  vec4 c2  = texture(source,vec3(_tex.xy - vec2(t_step,0.0f),layer));
  vec4 c3  = texture(source,vec3(_tex.xy + vec2(0.0f, t_step),layer));
  vec4 c4  = texture(source,vec3(_tex.xy - vec2(0.0f, t_step),layer));

  vec4 c5  = texture(source,vec3(_tex.xy + 2 * vec2(t_step,0.0f),layer));
  vec4 c6  = texture(source,vec3(_tex.xy - 2 * vec2(t_step,0.0f),layer));
  vec4 c7  = texture(source,vec3(_tex.xy + 2 * vec2(0.0f, t_step),layer));
  vec4 c8  = texture(source,vec3(_tex.xy - 2 * vec2(0.0f, t_step),layer));

  vec4 c9  = texture(source,vec3(_tex.xy + vec2(t_step,t_step),layer));
  vec4 c10 = texture(source,vec3(_tex.xy - vec2(t_step,t_step),layer));
  vec4 c11 = texture(source,vec3(_tex.xy + vec2(t_step, -t_step),layer));
  vec4 c12 = texture(source,vec3(_tex.xy - vec2(t_step, -t_step),layer));

  float v0 = clamp(c0.a,0.0f,1.0f);
  float v1 = clamp(c1.a,0.0f,1.0f);
  [...] //omitted
  float v12 = clamp(c12.a,0.0f,1.0f);

  vec4 step1 = c0 * v0 + c1 * v1 + c2 * v2 + c3 * v3 + c4 * v4;
  float vstep1 = v0 + v1 + v2 + v3 + v4;
  vec4 step2 = c5 * v5 + c6 * v6 + c7* v7 + c8 * v8;
  float vstep2 = v5 + v6 + v7 + v8;
  vec4 step3 = c9 * v9 + c10 * v10 + c11 * v11 + c12 * v12;
  float vstep3 = v9 + v10 + v11 + v12;

  fragColor = (step1 + step2 + step3 )/max(vstep1 + vstep2 + vstep3, 1.0f);
}
```

Listing 5.11: Custom mipmap filtering on GPU. `_tex` are the texture coordinates on the screen aligned quad.

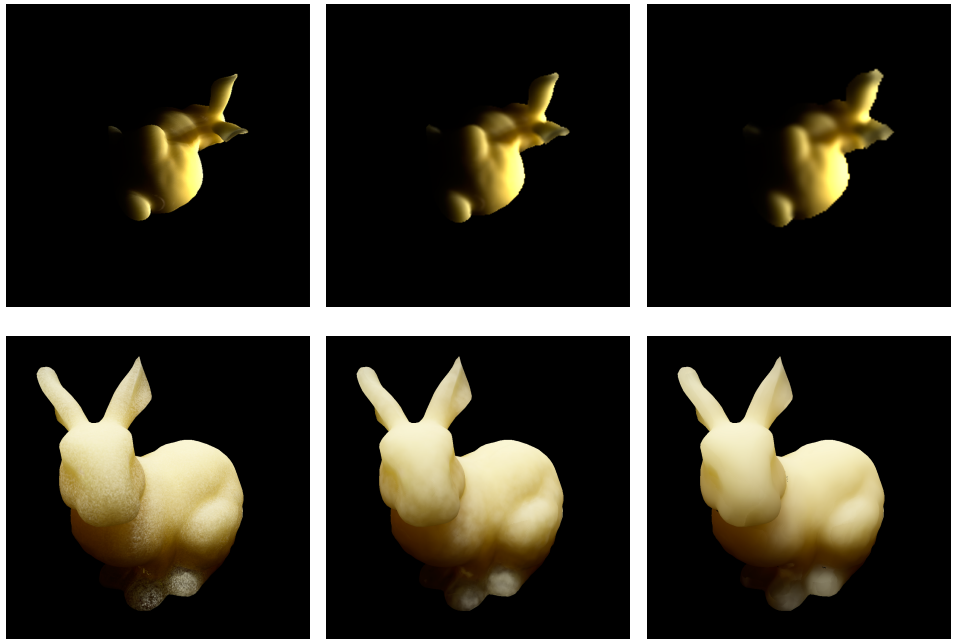


Figure 5.16: Mipmap generation. On the first row, left to right, the base level and two subsequent applications of the bilinear filter (the images have been cropped for a better visualization). On the bottom row, the model sampled with the different mipmap levels. We note that at level 2 artifacts start to emerge along the camera seams.

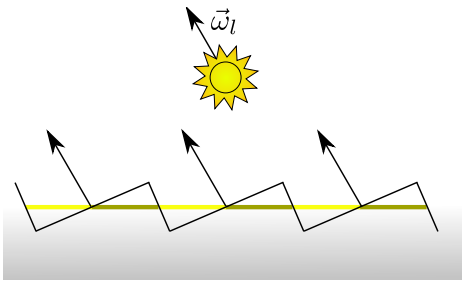
5.4.3 Shadow bias

The shadow mapping described in Section 5.3.5 has an obvious problem, called *shadow acne*. As we can see in Figure 5.17, the discretization of depth comes with a problem: depending on the incidence angle, the surface starts generating an alternating pattern of dark and light on the visible areas. The solution in this case is to introduce a constant factor when we are comparing the texture space position and the depth of a point, called *shadow bias* ϵ_b. The test then becomes:

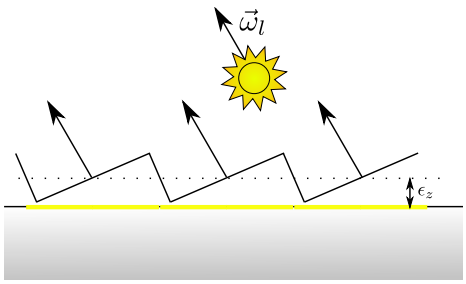
$$p_z - \epsilon_b < T(p_x, p_y)$$

The result is depicted in Figure 5.17. As we can see, we raise the sampling by the bias factor, so the lit areas do not present shadow acne anymore. This comes with a drawback: the shadow bias, for thin objects, it introduces another artifact, called *peter panning*, i.e. the shadow and the object become disconnected. In the case of our method, a too large bias makes the different directions to

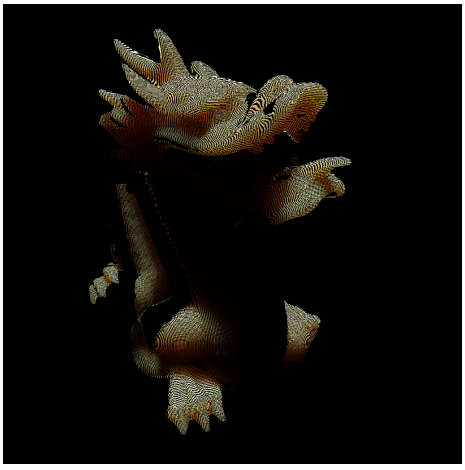
become ”disconnected”, not representing a faithful result.



(a) Shadow acne



(b) Shadow acne, with shadow bias



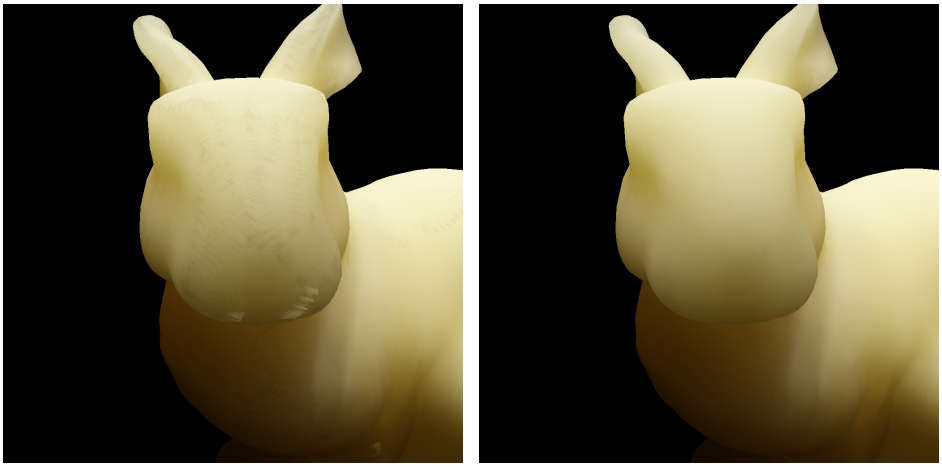
(c) Shadow acne, result

(d) Shadow acne, with shadow bias, result

Figure 5.17: Shadow acne (Figure 5.17a) origin: the angle of the light causes little bands to appear. Adding a small offset ϵ_z to the sampling (5.17b) corrects the result, leading to a full lit surface. On the bottom row, the two figures 5.17c and 5.17d show the results with and without the correction.

5.4.4 Texture discretization artifacts

A problem related to the previous one is texture discretization artifacts, that appear as seams in the texture in Figure 5.18. The problem is similar to the one of depth bias, but it manifests on the direction perpendicular to the directional cameras. Because of the discretization and the depth bias in the shadow sampling, some pixels become false positives, being incorrectly sampled even if they



(a) Without correction **(b)** With correction

Figure 5.18: Combination offset artifacts. Without the correction, artifacts start to appear. With a correction of $0.002 \approx 2/1024$ the result is greatly improved.

are not visible. To fix this, we apply a transformation to the vertex in the final step that shrinks it a little bit towards the center of the texture. The shrinking is made using the camera direction as a main axis, according to the formula:

$$\mathbf{x}_o^{shrink} = \mathbf{x}_o - \epsilon_c(\vec{n}_o - \vec{\omega}_d(\vec{n}_o \cdot \vec{\omega}_d));$$

Where ϵ_c is a parameters set by the user. In this way we obtain a shrinkage of one pixel of the model when we use $\mathbf{x}_o^{shrink}$ for sampling the radiance map. As we can see in Figure 5.18, adding this correction solves the problem.

5.5 Extensions to the method

In this section, we describe briefly how we extended our implementation in order to cover a wider range of cases, especially regarding different types of light.

5.5.1 Rendering with multiple lights

Rendering with multiple lights requires only a slight extension to the method. We just need to adjust the computation in 5.1 in order to account for multiple

Figure 5.19: Rendering with two point lights. The material parameters used are for white grapefruit juice. One light comes from the left ($\vec{\omega}_l = (1, 0, 0)$), with radiance $L_d = (13, 5, 5)$. One comes from the top ($\vec{\omega}_l = (0, 0, 1)$), with radiance $L_d = (5, 5, 13)$.

lights, obtaining a result as the one in Figure 5.19. We notice that this computation may come with a performance penalty, since we are now computing the contribution from LN samples instead of N, where L is the number of lights. To avoid this, we account in advance and calculate only $\frac{N}{L}$ samples per light:

$$R^{t,k}(\mathbf{x}_o) = \sum_{l=1}^{L} L_l \sum_{i=1}^{N/L} S(\mathbf{x}_i^{t,k}, \vec{\omega}_l^t, \mathbf{x}_o, \vec{\omega}_o) \exp\left(\sigma_{tr} r_i^{t,k}\right), \quad t \in [0, T], \quad k \in [0, K-1]$$

We can see that only the light vector $\vec{\omega}_l^t$ and the radiance L_l depend on the current light, while the other parameters are left unchanged. On the implementation level, we first need to change the light map in order to be a layered texture as well. As in step 2, we employ layered rendering to render all the light

maps at same time. Then we simply add an extra loop in the shader of step 2 to account for multiple lights. As for the light parameters, they are passed to the shader as arrays of uniforms. The outline of the new shader illustrating how lights are computed is listed in listing 5.12.

```
#version 430
layout(location = 0) out vec4 fragColor;

[...] // constants, includes, textures omitted

uniform sampler2DArray vtex; // texture with vertices
uniform sampler2DArray ntex; // texture with normals

smooth in vec3 position;
smooth in vec3 norm;

uniform vec4 light_pos[MAX_LIGHTS];
uniform vec4 light_diff[MAX_LIGHTS];
uniform int light_number;
uniform mat4 lightMatrices[MAX_LIGHTS];

[...] //BSSRDF code, uniform and functions

void main(void)
{
    vec3 xo = position;
    vec3 no = normalize(norm);

    [...] // sampling the old color, noise calculation

    for(int k = 0; k < light_number; k++)
    {
        vec3 wi = light_pos[k].xyz;
        vec4 light_post = lightMatrices[k] * vec4(xo,1.0f);
        vec2 circle_center = light_post.xy;
        vec3 Li = light_diff[k].xyz;

        for(int i = 0; i < max_samples; i++)
        {
            [...] // sampling from layer k in textures vtex and ntex
            [...] // summing over BSSRDF on vector accumulate
        }
    }

    fragColor = vec4(accumulate,1.0f);
    [...] //adding old color
}
```

Listing 5.12: Outline of the shader in step 2 with support for multiple lights.

5.5.2 Rendering with other kinds of light

Our method does not depend on a specific type of light in order to render a material. Once we have a formula for the radiance for the light, we can simply introduce it in equation 5.1 and compute the result. So, for example, for a point light the equation would become.

$$R^{t,k}(\mathbf{x}_o) = I_l \sum_{i=1}^{N} \frac{S(\mathbf{x}_i^{t,k}, \frac{\mathbf{x}_l - \mathbf{x}_i}{\|\mathbf{x}_l - \mathbf{x}_i\|}, \mathbf{x}_o, \vec{\omega}_o)}{\|\mathbf{x}_l - \mathbf{x}_i\|} \exp\left(\sigma_{tr} r_i^{t,k}\right), \quad t \in [0,T], \quad k \in [0, K-1] \tag{5.4}$$

Where $\mathbf{x}_l$ is the light position and I_l the light intensity. More generally, if we have a function to compute the light direction $\vec{\omega}_l(\mathbf{x}_i)$ and a function to compute radiance $L(\mathbf{x}_i, \vec{\omega}_l(\mathbf{x}_i))$ we can express equation 5.1 in the following way:

$$R^{t,k}(\mathbf{x}_o) = \sum_{l=1}^{L} \sum_{i=1}^{N/L} L^t(\mathbf{x}_i, \vec{\omega}_l(\mathbf{x}_i)) S(\mathbf{x}_i^{t,k}, \vec{\omega}_l^t(\mathbf{x}_i), \mathbf{x}_o, \vec{\omega}_o) \exp\left(\sigma_{tr} r_i^{t,k}\right)$$

If we have only point and directional lights, we have the following setup for the two formulas:

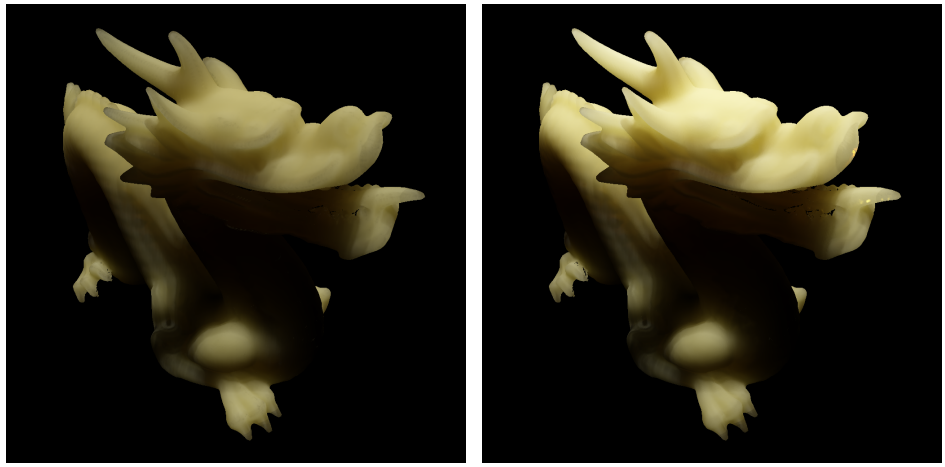
$$\vec{\omega}_l(\mathbf{x}_i) = \begin{cases} \vec{\omega}_l & \text{if l is directional with } \vec{\omega}_l,\ L_l \\ \frac{\mathbf{x}_l - \mathbf{x}_i}{\|\mathbf{x}_l - \mathbf{x}_i\|} & \text{if } l \text{ is point with } \mathbf{x}_l,\ I_l \end{cases}$$

$$L(\mathbf{x}_i, \vec{\omega}_l(\mathbf{x}_i)) = \begin{cases} L_l & \text{if l is directional with } \vec{\omega}_l,\ L_l \\ \frac{I_l}{\|\mathbf{x}_l - \mathbf{x}_i\|} & \text{if } l \text{ is point with } \mathbf{x}_l,\ I_l \end{cases}$$

On the implementation level, we can still use same uniforms that we use for multiple lights: the position vector will become either the position of the direction of the light. We make the position vector of the light carry some extra information in the alpha channel: if the alpha is zero (i.e. the vector is of type $(x, y, z, 0)$), we are dealing with a directional light, while if the alpha is one (i.e. the vector is of type $(x, y, z, 1)$) we are using a point light. The code to calculate lights thus becomes:

```
int current_light = k;
vec3 wi = light_pos[current_light].xyz;
vec4 rad = light_diff[current_light];
vec3 topoint = wi - xo;
float light_type = light_pos[current_light].a;
wi = (light_type > 0)? normalize(topoint) : normalize(wi);
vec3 Li = light_diff[current_light].xyz;
Li = (light_type > 0)? Li / dot(topoint,topoint) : Li;
```

Listing 5.13: GLSL Code to calculate the radiance `Li` and the incoming light direction `wi` for the k-th directional or point light source.



(a) Directional: $\vec{\omega}_l = (0, 0, 1)$ $L_d = 25$ **(b)** Point: $\mathbf{x}_l = (0, 0, 2)$ $I_d = 60$

Figure 5.20: Rendering with a directional and a point light. The material parameters are the ones for potato.

5.5.3 Rendering with environment light illumination

As for the environment lighting, we proceed as we outlined in Section 4.4. In the method chapter, we omitted how we do actually organize and sample our CDF. The idea is to have one CDF for the rows of the function that gives us the CDF according to the probability $p_u(u)$. Then we have a CDF for each one of the column of the functions, that sample the probability $p_v(v|u)$. When we select a random couple of random points (ζ_1, ζ_2) we first sample the first function according to ζ_1, then we sample the resulting column using ζ_2.

We define our discrete CDF as and array C of $s + 1$ elements as follows, where s is the size of the distribution P (that would be either n or m in our case). P is an array of N elements.

$$C[i] = \frac{\sum_{k=0}^{i-1} C[k] + P[i]}{s\ C[s]}$$

Then, we sample our CDF using a generic value ζ using the following:

$$u = j + \frac{\zeta - C[j]}{C[j+1] - C[j]}$$

Where $j = \min_j\{j \ \in \ [0, s] \mid C[j] > \zeta\} - 1$ is the CDF value right before the CDF surpasses ζ. This is basically the inverse function we discussed in Section

Figure 5.21: Rendering a dragon using sixteen directional lights generated from the Doge map (with reflection). The material parameters are a potato material.

4.4. Once we have calculated the two values (u_1, u_2), as before we convert them to spherical coordinates, from which we create a set of directional lights that approximate our environment light. We can appreciate the rendering with different numbers of generated lights using the doge light map in Figure 5.21.

5.6 Discussion

In this section, we anticipate some of the results that will be presented in Chapter 6, in order to analyze our method and present advantages and disadvantages of the method, and more specifically of our implementation.

5.6.1 Advantages

The great advantage of this method, that comes from the BSSRDF model, is that it already accounts for self shadowing and self occlusion. So, no extra steps are required in order to account for these effects, that usually are difficult to evaluate in a real-time fashion. Moreover, if we reuse the same light map for all the objects in the scene, it is possible to account for the occlusion between objects. So, if an object is placed between the light and another object, a soft shadow of the first object on the second one will appear. However, since the material properties are not stored in the light map, the contribution from the other points will use the same properties that we are using to render the object (so, another object will cast a "potato shadow" on a potato object).

Another advantage of our method is that is tightly coupled with the shadow mapping pipeline. Shadow mapping is widely used nowadays in most of real time graphics applications, so the cost of introducing our method is greatly reduced, since the light map data are usually available for use. In addition, the method is generally adaptable to any kind of shadow mapping technique, providing that it is able to store vertices and normals, and that those are accessible from step two in the algorithm. The cost of adding normal and vertex storage is usually only heavy on the memory requirements, since the vertex position is calculated anyway during the render to the shadow map.

Compared to other techniques for rendering translucent materials (such as voxelization), our method has low memory requirements. As we will see, the size of the textures employed in the method should not be too big in order to achieve a reasonable quality. This also comes from the fact that subsurface scattering effects are result of blurring of the incoming light, so a low size texture can represent the effects very well.

Finally, another advantage of the method is that the final step relies only on the knowledge of the exiting position and normal, so it can be integrated in both a forward and a deferred rendering pipeline, making it very flexible.

5.6.2 Disadvantages

The first and biggest disadvantage of this method is that one frame is giving only an approximate and noisy approximation of the final result, that needs to be blurred using mipmaps in order to get a smooth result. In addition, every time something changed in the scene, the computations need to be redone every frame, not allowing the method to ever reach convergence. In this case, ping

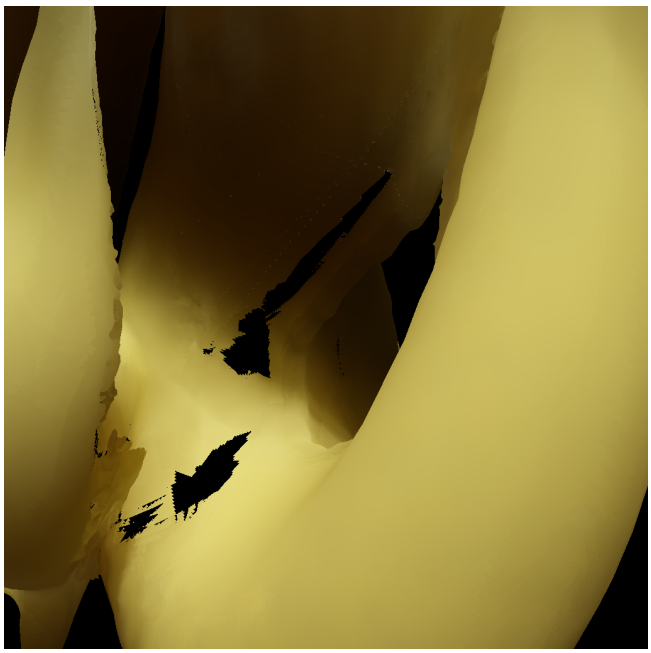


Figure 5.22: Back holes in the final appearance caused by a suboptimal placement of cameras.

pong is not used at all, which results in a waste of precious memory space. The ping pong can be reused to account partially for the computation of the previous frame. In this way, the noise is reduced, at the price that the method is slower to react to swift changes of condition in the scene, generating ghosting artifacts.

As we can see in Figure 5.22, it is difficult for our method to cover the whole surface of the object. The most hidden areas, such as the internal part of the dragon mouth, are not covered by any camera, and so the result are black holes on the surface. The problem is well known when acquiring an object from a 3D camera, where a rotation is often necessary in order to capture all the features of a mesh. A solution for this problem is to let the artist place the cameras himself/herself, but with an additional time effort.

Another problem is that our method, relying on shadow mapping at its core, inherits all the defects and problems of shadow mapping. One of these is that constant shadow bias and offsets are sometimes not enough to cover the defects that inevitably generate when using this method. In addition, the artist must tweak this parameters in order to get a good result.

Finally, a defect that comes from the method we are using to generate the directional lights from the environment map generally has the problem that in certain radiance maps, where there are a lot of very bright spots. In these maps, it tends to account only for the very bright spots and thus not approximate the overall result very well. In this case, a solution based on spherical harmonics [Sloan, 2008] environment lighting may be preferable.

Chapter 6

Results

In the last chapter, we discussed the implementation of our method. In this chapter, we are going to check it after the assumptions we stated in chapter 1, discussing advantages, disadvantages, and trade-offs. In the first section, we will discuss the quality result of our method, comparing it to path-traced results. In the second section, we will discuss how close we can get to the expected results in the performance domain, testing it for different parameters.

6.1 Parameters

The parameters were introduced in the previous chapter 5. We will sum them up here, in order to have a reference for this chapter.

- M, the total number of points in the disk. We recall that we make a disc of the size of the object bounding box and then generate M exponentially based samples in it. Unless otherwise stated, $M = 1000$
- q, the modifier of the exponential distribution of the samples. Unless otherwise stated, $q = 1$.
- N, the number of samples used from the disc. It is always $N < M$.

- K, the number of directions used in the radiance map. Unless otherwise stated, $K = 16$.
- L, the number of lights in the scene.
- W_l, the size in pixel of the light map. Unless otherwise stated, $W_l = 512$.
- W_r, the size in pixel of the radiance map. Unless otherwise stated, $W_r = 1024$.

Other parameters are available in the method, such as all the parameters of the cameras. In this section, we listed only the parameters that makes sense for the user to tweak, and that can directly influence the quality or the performance of the final result.

6.2 Quality comparisons

In the domain of quality, we compared our method to offline rendered solutions, always made using the directional dipole. The solutions are compared only visually, since if was not possible to perfectly match the perspective and view cameras of the original pictures. We will see that the visual results of our method, given enough samples, can produce a result comparable to the one of a offline rendered solution. We will compare our results at convergence (after 100 frames of evolution), unless otherwise stated.

6.2.1 Optimal radius

In section 5.3.4.1, we discussed how to generate exponentially biased points in our implementation. In our algorithm, to choose how to distribute the sampling points, we adopt the strategy of choosing a distribution biased towards the following exponent:

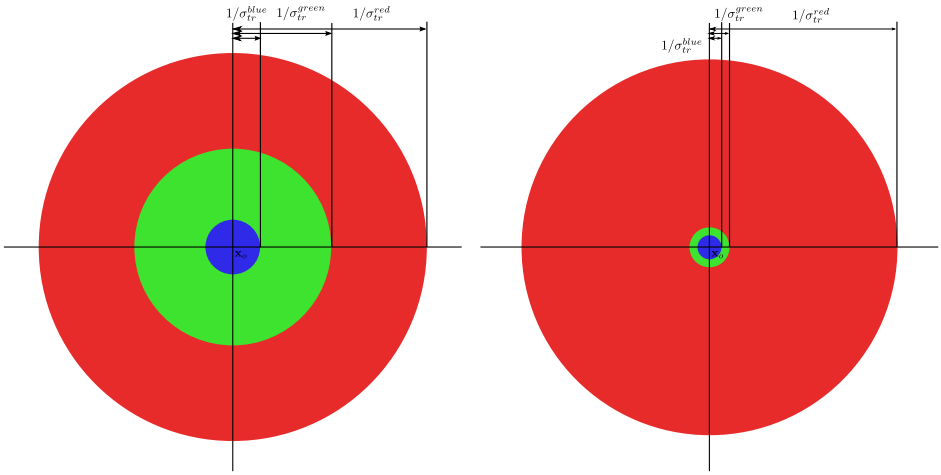
$$\sigma^* = \frac{\min(\sigma_{tr,x}, \sigma_{tr,y}, \sigma_{tr,z})}{q}$$

We can see why we use the minimum coefficient in Figures 6.1 and 6.2. σ_{tr} as a coefficient, is related to how much the scattering effects propagate. If we look at the directional dipole formula in 3.13, we can see that the dipole has a leading term depending on $e^{-\sigma_{tr}t}$. If we consider only this term, the average distance the light travels in the medium is $1/\sigma_{tr}$. Figure 6.1 shows these radii for the potato and the ketchup material. So, in order to account for all channels in

our sampling, we need choose as a exponent for our sampling pdf the maximum radius:

$$\max\left(\frac{1}{\sigma_{tr,x}}, \frac{1}{\sigma_{tr,y}}, \frac{1}{\sigma_{tr,z}}\right) = \min(\sigma_{tr,x}, \sigma_{tr,y}, \sigma_{tr,z})$$

And this is how we obtain Equation 6.2.1. We will see in the next section that the q parameter in the equation is necessary in order to allow better performance with a fewer number of samples. In Figure 6.2 we can see the effects of choosing different radii according to the three coefficients in ketchup. When we use the red radius, all the effects are accounted for. When we use the green radius, we lose the red absorption part. If we use the blue channel, the green and the red part of the absorption spectrum disappear, resulting in a gray/blue appearance.



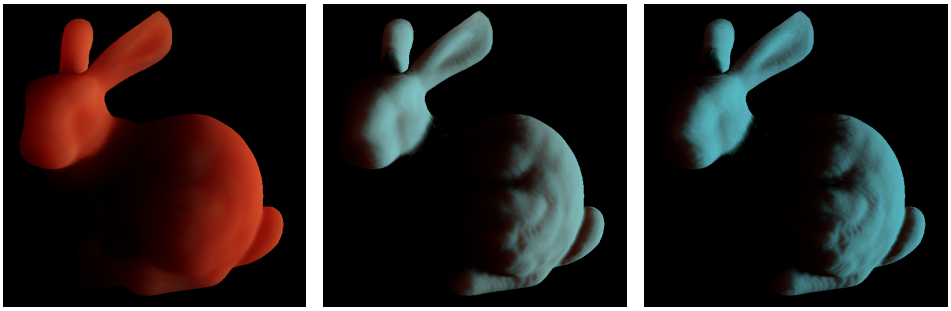
(a) Potato **(b)** Ketchup

Figure 6.1: Average radius of which the scattering effects are important for ketchup and potato materials.

6.2.2 Tests with different number of samples

The material tested where:

- Potato, a highly scattering isotropic material ($\alpha = \sigma_s/\sigma_a \approx 10^2$),
- Marble, a even higher high scattering material ($\alpha \approx 10^4$)
- White grapefruit juice, a material with a big forward scattering component ($\alpha \approx 10^2$, $g \approx 0.5$)



(a) $\sigma^* = \sigma_{tr,x}$ **(b)** $\sigma^* = \sigma_{tr,y}$ **(c)** $\sigma^* = \sigma_{tr,z}$

Figure 6.2: Bunny rendered with different radii. We can appreciate the different absorptions according to the distribution radius. $N = 20$, $M = 160$, $q = 1$ in all images.

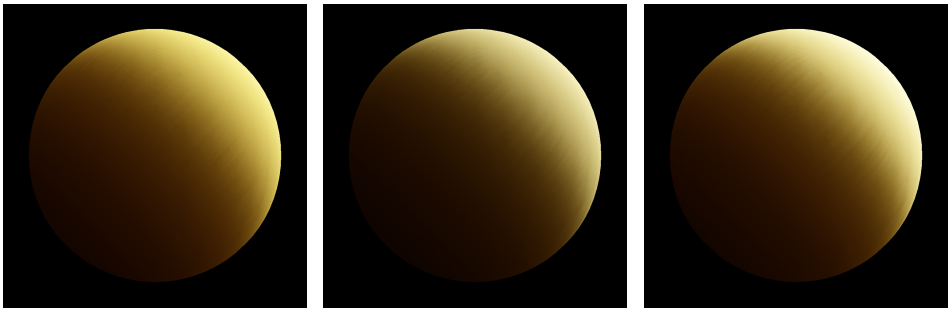
- Ketchup, a material that has a strong absorption component in the red channel ($\alpha_{red} \approx 10^{-2}$, $\alpha_{green} \approx \alpha_{blue} \approx 10^{2}$)
- Beer, a high absorption material with nearly no scattering.

We are biased against highly scattering material because those are the material where our BSSRDF model gives our best results. However, we included materials as beer and ketchup to provide a comparison of our method even where the original BSSRDF model should fail.

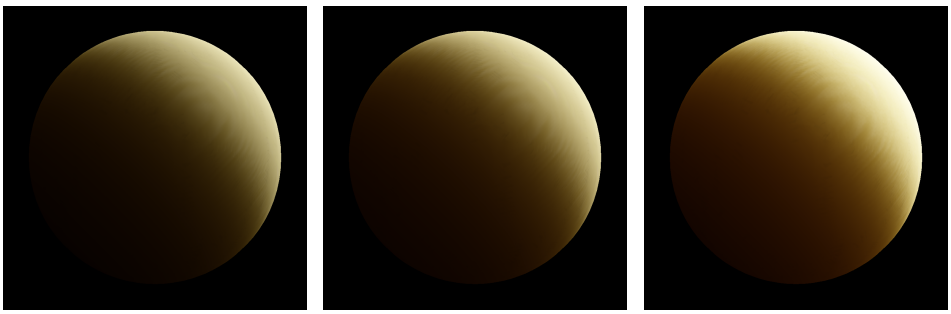
The first comparison we make is with the path traced spheres generated in Figure 4.6. The results are illustrated in Figure 6.3, comparing different values of M and N for a potato sphere. We can see that different values of M and N greatly influence the results. In fact, for a big M and the same N, the samples tend to be closer to the exiting point x_o, so the results are more accurate on the highlight region. If instead we chose a relatively small M, the points are widely spread on the surface, to the absorption of the material is accounted for more. This results in a sphere with gradients that are closer to the reference. In this case, the highlights are way more difficult to see.

The result that gets closer to the reference solution is the one where $N = M = 1000$, but as we will see this values are unfeasible in the realm of performance, even for simple models. So, it is the artist that should find the right balance between M and N in order to get a satisfying result.

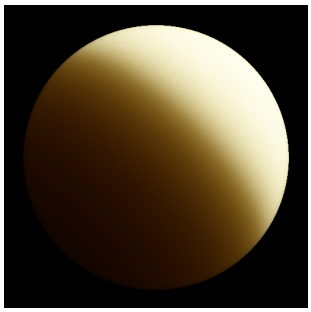
In the next set of experiments, we can see the utility of the q parameter. We tested a sphere of white grapefruit juice in Figure 6.4, a material with a high



(a) $N = 120$, $M = 120$ **(b)** $N = 120$, $M = 300$ **(c)** $N = 300$, $M = 300$



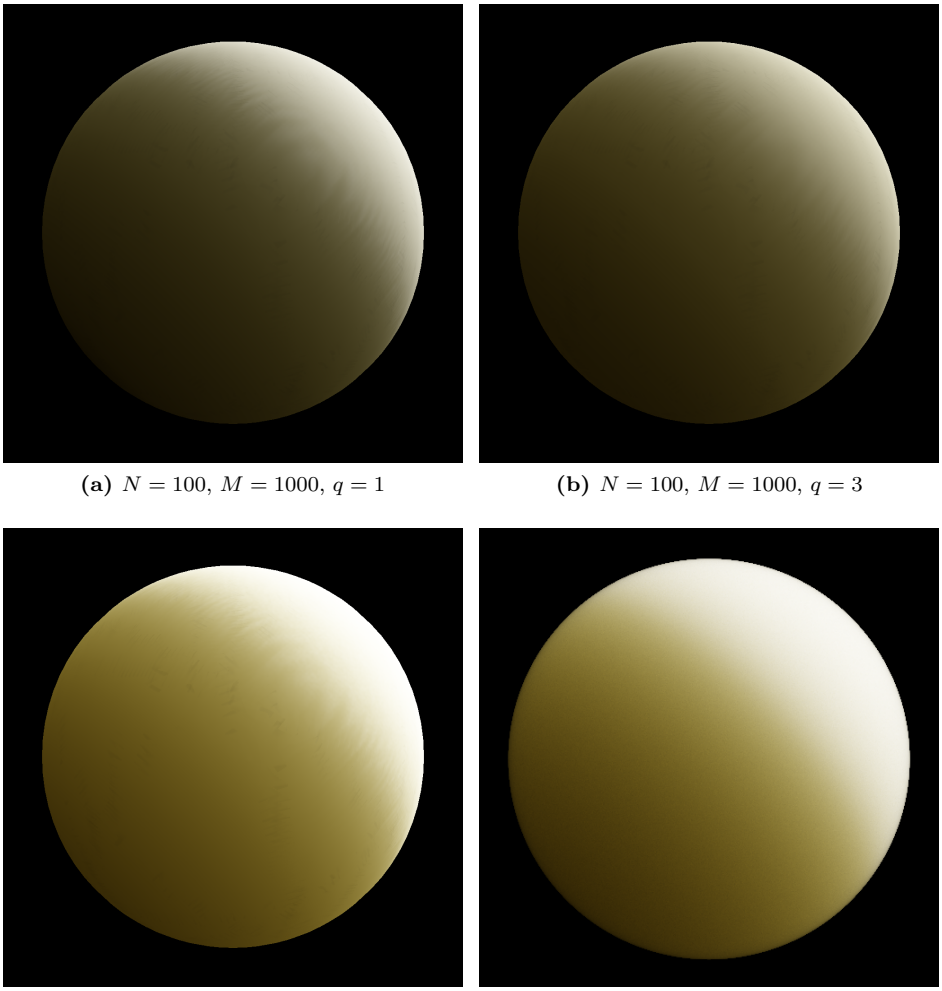
(d) $N = 120$, $M = 1000$ **(e)** $N = 300$, $M = 1000$ **(f)** $N = 1000$, $M = 1000$



(g) Path traced solution

Figure 6.3: Path traced rendering on a sphere of potato material compared with the results of our method. The parameters are from table 4.1.

forward scattering component. In this case we can see that that with a smaller number of samples $N = 100$ and a $q = 3$ we can approximate very well the solution where $N = 1000$ and $q = 1$, way more expensive for the GPU. In a test with marble, in Figure 6.5 we can see that our sampling introduces artifacts, especially on the color transition area. In this case, q slightly relieves the artifacts generated from the sampling.



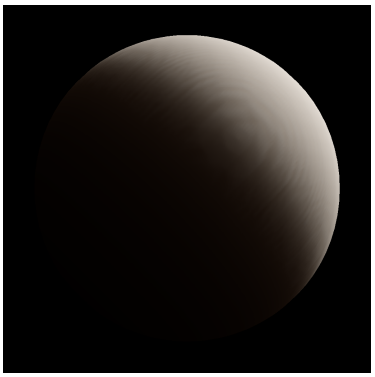
(a) $N = 100$, $M = 1000$, $q = 1$

(b) $N = 100$, $M = 1000$, $q = 3$

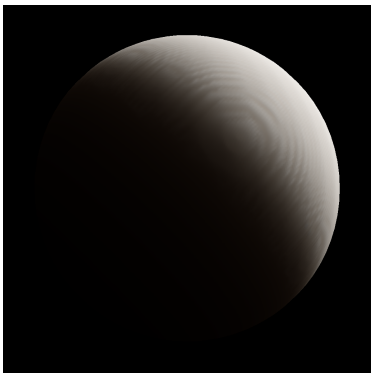
(c) $N = 1000$, $M = 1000$, $q = 1$

(d) Path traced result

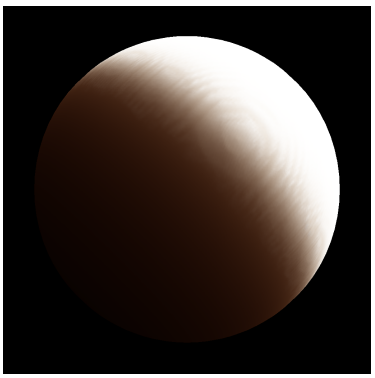
Figure 6.4: Path traced rendering on a sphere of white grapefruit material compared with the results of our method. The parameters are from table 4.1. We can see that a higher q helps us in approximating the path traced solution with fewer samples.



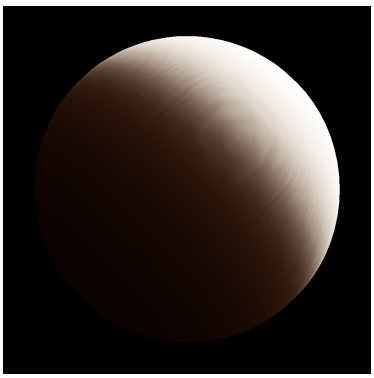
(a) $N = 120$, $M = 1000$, $q = 1.5$



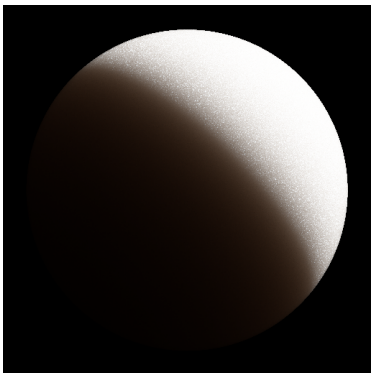
(b) $N = 120$, $M = 1000$, $q = 1$



(c) $N = 1000$, $M = 1000$, $q = 1.5$



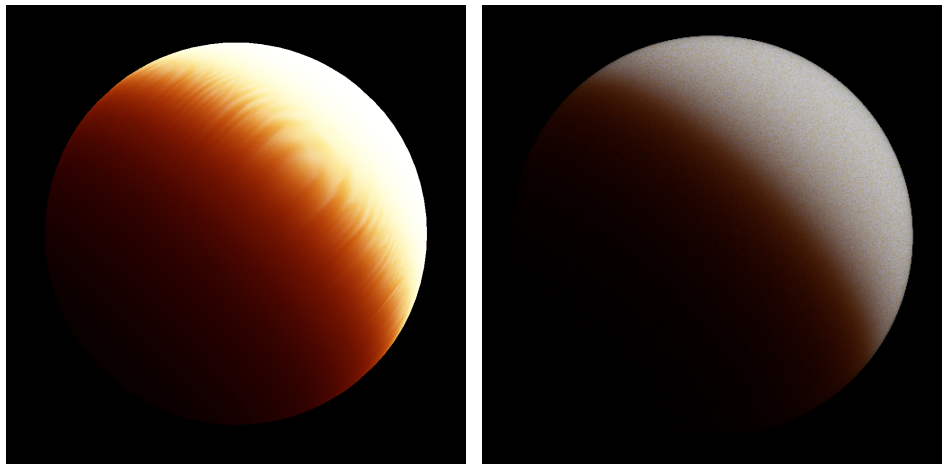
(d) $N = 1000$, $M = 1000$, $q = 1$



(e) Path traced result

Figure 6.5: Path traced rendering on a sphere of marble material compared with the results of our method. The parameters are from table 4.1.

In Figure 6.6, we can see a comparison between our method and a beer material: despite the obvious artifacts due to sampling, our results show a more realistic result that a path traced one. This happens because of the extremely low scattering coefficient of beer, that makes it unfeasible to use a Monte Carlo path tracer, since we need to sum a lot of contribution in order to remove all the noise.

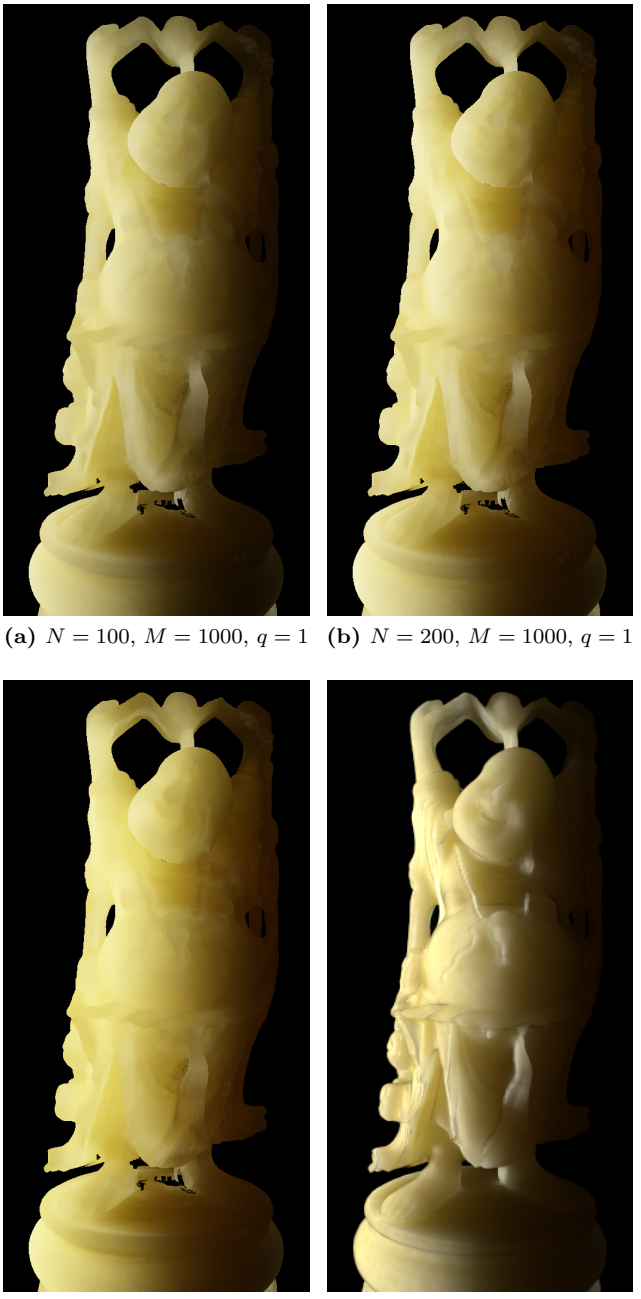


(a) $N = 100$, $M = 100$, $q = 1$ **(b)** Path traced result

Figure 6.6: Path traced rendering on a sphere of beer material compared with the results of our method. The parameters are from table 4.1.

The results that we obtained in the sphere renderings affect also the rendering of the full models. We tested a Buddha made of potato and a Dragon made of ketchup, that will be tested for performance in section 6.3. For the Buddha, we observe a slight color shift compared to reference as for the spheres in Figure 6.3.

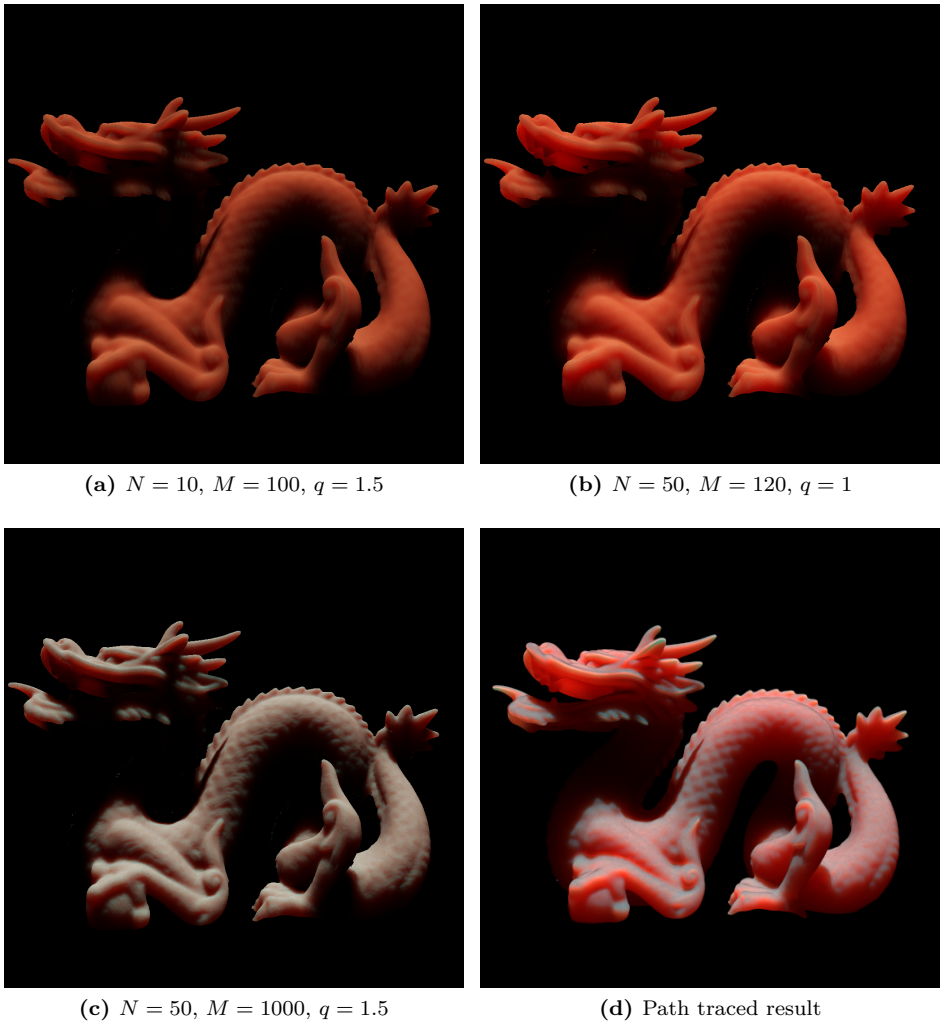
For the ketchup Dragon, instead, we observe that for a very low number of concentrated samples, the absorption contribution disappears, leaving only a gray scattering contribution that exhibits a pearling effect. If we reduce M, the highlights reduce and the absorption of the red part of the spectrum is accounted for.



(a) $N = 100$, $M = 1000$, $q = 1$ **(b)** $N = 200$, $M = 1000$, $q = 1$

(c) $N = 100$, $M = 300$, $q = 1$ **(d)** Path traced result

Figure 6.7: Rendering of a potato Buddha using the directional dipole, with increasing number of samples.



(a) $N = 10$, $M = 100$, $q = 1.5$

(b) $N = 50$, $M = 120$, $q = 1$

(c) $N = 50$, $M = 1000$, $q = 1.5$

(d) Path traced result

Figure 6.8: Rendering of a ketchup Dragon using the directional dipole, with changing values of M and N.

6.2.3 Radiance map sizes tests

Finally, we tested the effect of reducing the size of the texture used for the radiance map, for the Dragon test. As we can see, for diminishing values of W_s the quality does not get too much worse until $W_s = 128$, where artifacts due to shadow mapping become evident, like bright seams and jagged pixels. In Figure 6.9 the difference between the 1024 and the 256 image is reported. In the image, we can see that most of the difference are within 10% of the high resolution value (since the image is enlarged 10 times and the full colored pixels are only a minority).

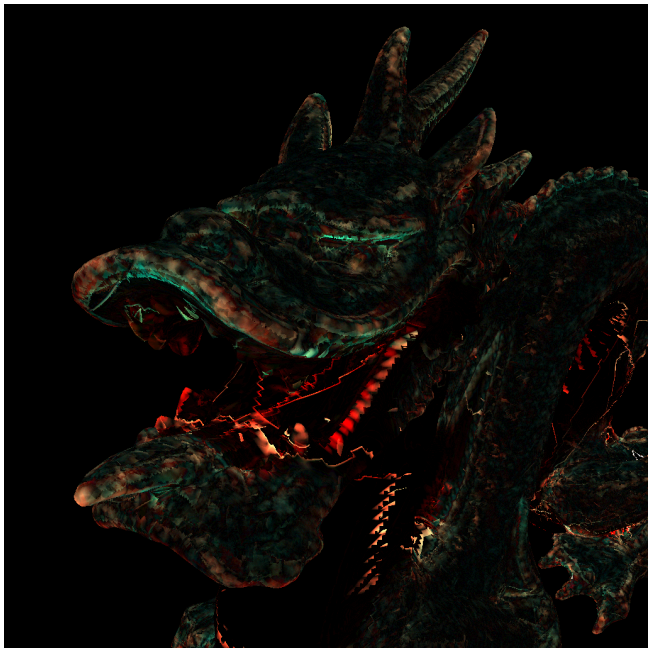
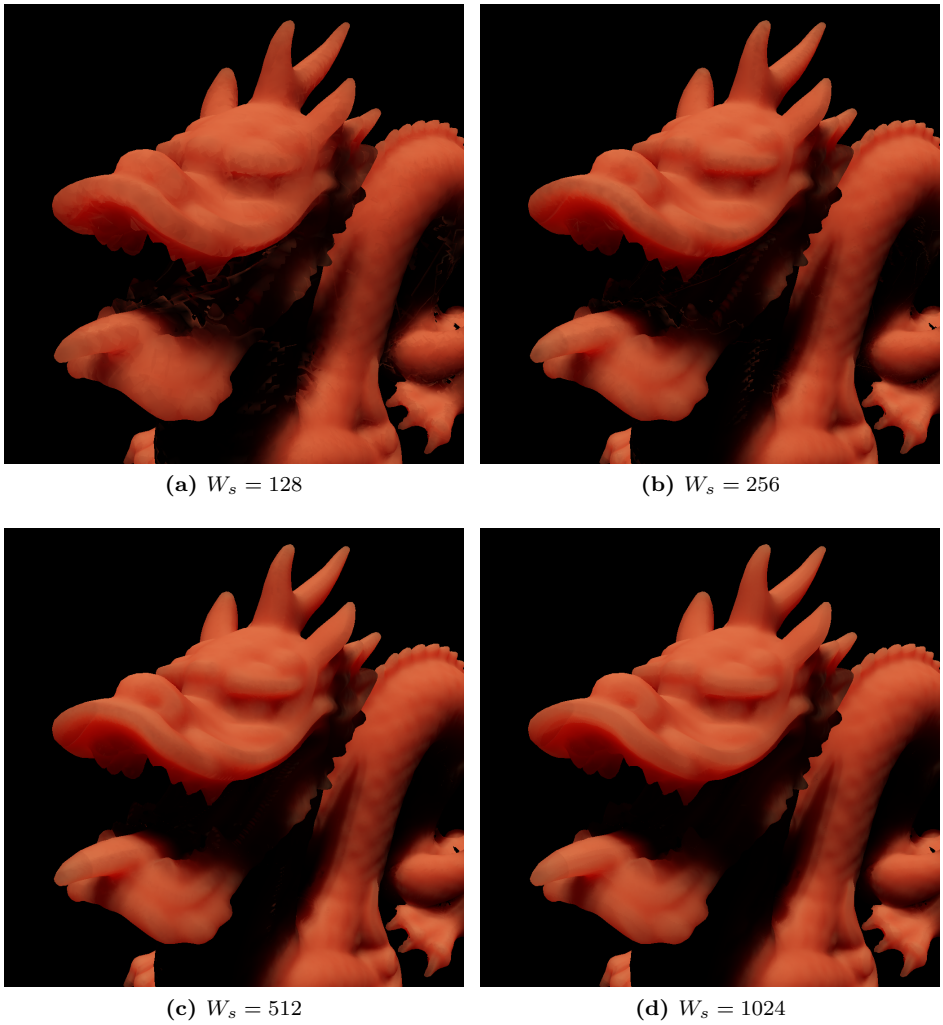


Figure 6.9: Difference multiplied by 10 times between 6.10d and 6.10b.



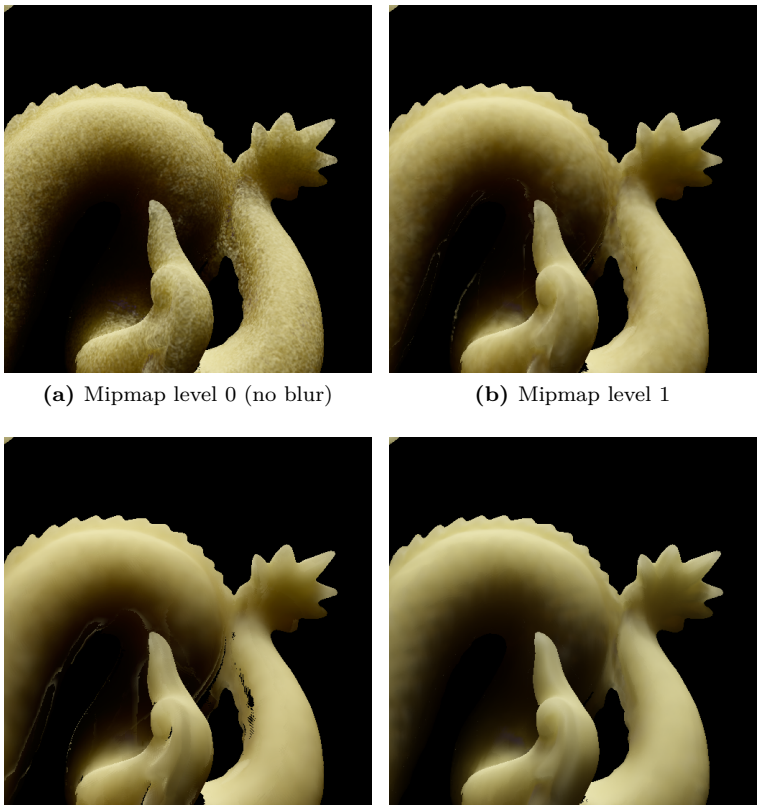
(a) $W_s = 128$ **(b)** $W_s = 256$

(c) $W_s = 512$ **(d)** $W_s = 1024$

Figure 6.10: Rendering of a ketchup Dragon using the directional dipole, with varying radiance map size W_s.

6.2.4 Tests of mipmap blurring quality

In this part we tested the quality improvement from the mipmap blurring introduced in section 5.4.2 to the final image. We can see the result of our method at the first frame of our simulation for different mipmap levels. At the beginning of the evolution, a strong blurring (two passes) is needed to compensate the high level noise, as we can see from image 6.11. During the evolution, a lesser level of mipmaps is needed in order to preserve a noiseless image. At convergence, we do not need mipmap blurring at all, as the result converge to the solution.



(a) Mipmap level 0 (no blur) **(b)** Mipmap level 1

(c) Mipmap level 2 **(d)** Convergence

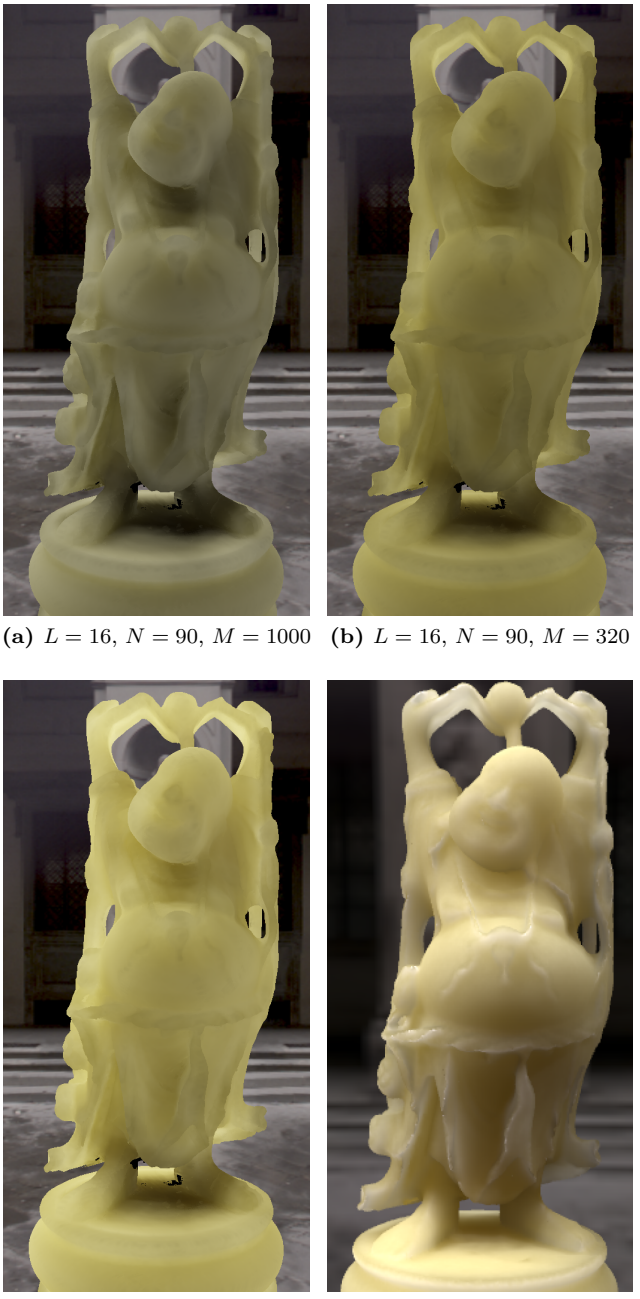
Figure 6.11: Detail of a potato Dragon in the first frame of the computation for different mipmap levels. We can see that the second mipmap level is very close to the final result. However, some artifacts due to the stretching of the mipmap appear, such as black spots and bright seams. All the Figures use $N = 32$ and $M = 300$.

6.2.5 Environment map illumination

All the given consideration so far are the same irregardless if we have an environment map or a single directional light. In this section we present the results obtained we environment light illumination. Visually, we obtain a nice result. Obviously, since the samples are split between different lights, an overlapping is inevitable, and we need an higher number of samples to obtain a decent result. The Bunny in Figure 6.12 was obtained using 16 directional lights, sampled using the method in 5.5.3 and 4.4, with $N = 80$ (5 samples per light) and $M = 1000$.

As for reference, we compared our results to the reference image of the potato Buddha from Frisvad et al. [2014] in Figure 6.13. We had to try to match the light settings and the camera, but here as well we notice the same color shift as for directional lights. Also in this case we used 16 directional lights to represent our skybox.

Figure 6.12: Marble Bunny rendered in the Doge environment map. Note that the light is predominantly from the up direction.

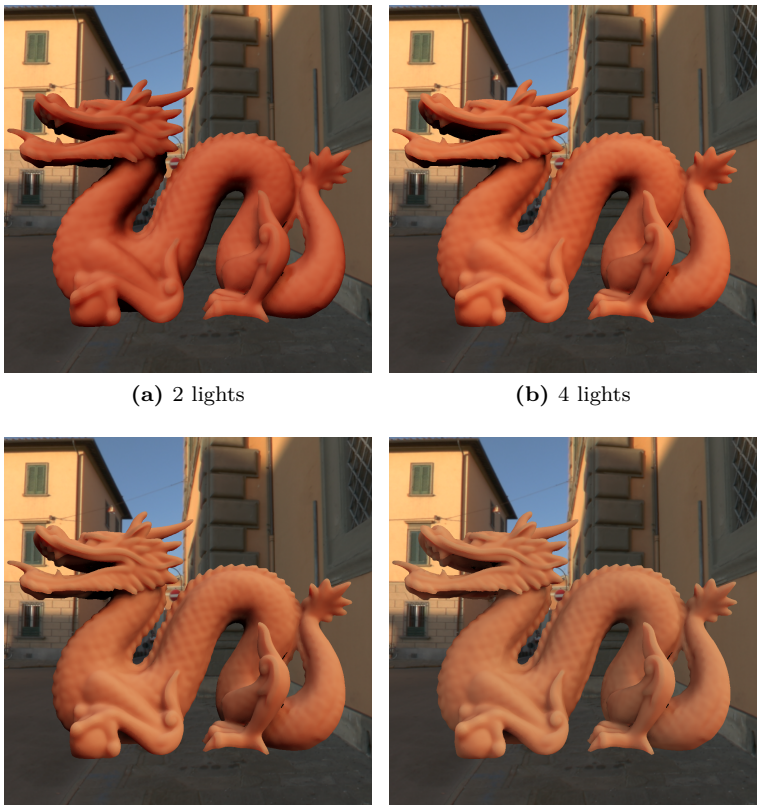


(a) $L = 16$, $N = 90$, $M = 1000$ **(b)** $L = 16$, $N = 90$, $M = 320$

(c) $L = 16$, $N = 90$, $M = 160$ **(d)** Hierarchical integration

Figure 6.13: Rendering of a potato Buddha using our environment lighting and the Uffizi map.

In the image of Figure 6.14, we compare the result using the Pisa Courtyard environment map on a Stanford Dragon, comparing a different number of lights. We can see the more lights we introduce, the more closer we get to a result that approximate true environment illumination. We observe also that the images have more scattering than absorption the more we increase the number of lights: this is because the total number of samples N does not change, so each light has less and less samples available. In this way, the samples tend to concentrate in the center and produce a result where the scattering component is predominant.



(a) 2 lights **(b)** 4 lights

(c) 8 lights **(d)** 16 lights

Figure 6.14: Rendering of a potato Dragon ($N = 32$, $M = 170$) using a different number of directional light to represent the environment map.

6.3 Performance tests

In this section, we examine and analyze the performance of our method. The tests in this section were made by keeping in mind the performance requirements we made in chapter 1, and they exploit how the different parameters listed in 6.1 influence the final result. For the tests, we used three main models, to which we will refer as Bunny, Dragon and Buddha (see table 6.1). The models cover a varying range of complexity. The Bunny model represents a typical model used in modern games and consoles (10^4 triangles), while the Dragon represents a highly detailed model, usually for visualization purposes (10^5 triangles). Finally, the Buddha model represent a high resolution model, and it will be used as a stress test for our algorithm (10^6 triangles).

Model	Vertices ($\#V$)	Triangles ($\#\Delta$)
Bunny	3581	$21474 \approx 10^4$
Dragon	50000	$300000 \approx 10^5$
Buddha	549409	$3262422 \approx 10^6$

Table 6.1: The three models used for our tests, with number of vertices and triangles.

All the tests were performed on a NVIDIA GeForce GTX 780Ti, a high-end modern GPU with OpenGL 4.3 capabilities. All the timings measure the average milliseconds during the 20th and the 40th frame during an evolution to convergence of 100 frames. The first frames were not measured because of the overhead introduced by the initialization procedure (texture creation, model parsing and loading, shader compilation, etc.).

6.3.1 Time algorithm breakdown

The first test we are going to perform is to time how much time the algorithm takes to perform the different steps illustrated in chapter 5. When we were timing the algorithm, we divided the whole computation into five different steps:

1. Initialization phase, where all the different constants and data structures in the algorithm are created and initialized.

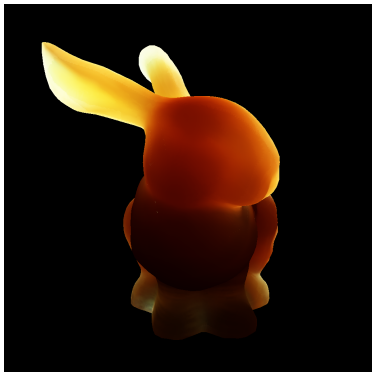
2. Render to light map step, that corresponds to step 1 in our outline.

3. Render to radiance map step, that corresponds to step 2 in our outline.
4. Mipmap generation, as in the extension outlined in section 5.4.2.
5. Final render combination, that corresponds to step 3 in our outline.

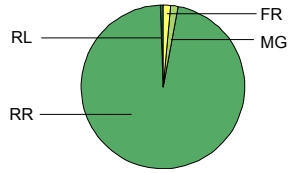
For this tests we tried all the three models, and the results are shown in Figure 6.15. The parameters were tweaked in order to reach the best compromise between visual quality and performance, in order to measure realistic timings.

We can observe that, irregardless of the test case, most of the time is spent computing the BSSRDF function for the samples in step 2. Thee render to lightmap step does not have a big performance impact, likely because all the tests were made with one light. We will test in the next section the impact of a different number of lights on the performance. Another consideration to do is that, as to be expected, the more triangles we have the less samples we can use. As we will see, this is not the only factor that causes a performance drop. However, the big number of triangles that has to be multiplied by the number of directions generates a big load on the GPU for rasterization procedures.

Regarding the mipmap generation, as expected its performance is not tied to the size of the model, but only to the size of the radiance map texture W_s and to the number of mipmaps we generate. Finally, the final combination step is tied as well to the size of the model, rising slightly with an increasing model size. However, in this case we are rendering only once the model to the main framebuffer, and not K times as in step 2.

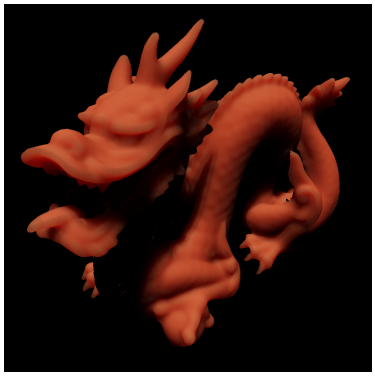


(a) Beer Bunny, 67.6ms. Point light. $N = 180$, $M = 210$, $q = 2$

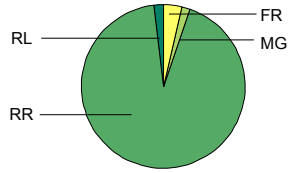


RL	RR	MG	FR	Tot
0.20	66.18	0.8	0.44	67.72

(b) Detailed timings.

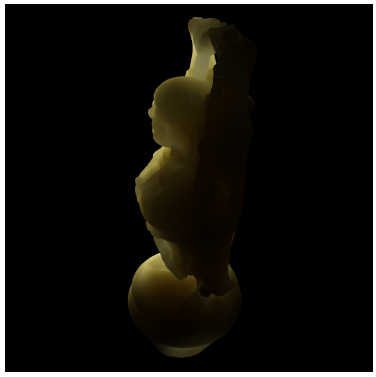


(c) Ketchup Dragon, 62.0ms. Directional light. $N = 20$, $M = 180$.

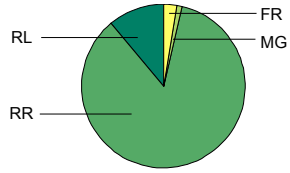


RL	RR	MG	FR	Tot
1.07	57.59	0.76	2.56	62.04

(d) Detailed timings.



(e) Potato Buddha, 98.0ms. Point light. $N = 10$, $M = 300$, $q = 1$



RL	RR	MG	FR	Tot
11.18	83.90	0.86	1.41	98.00

(f) Detailed timings.

Figure 6.15: Rendering of different models, and graphs that illustrate how the timings are split into the different phases of the algorithm. All tests use $W_l = W_s = 512$. The acronyms represent RL = Render to lightmap, RR = Render to radiance map, MG = Mipmap Generation and FR = Final Rendering.

6.3.2 Tests for varying parameters

In this section, we will discuss how well our method behaves for changing parameters. From the quality tests before we have learned that the parameters that are related mostly to the quality of the are N, M, q and to a certain extent W_s. The number of directions K is important in order to ensure to cover the whole model, and if we are generating the cameras automatically it cannot be too low. Of the mentioned parametrs, M and q do not directly influence performance, as they are used only at the beginning of the computation in order to distribute the points on the disc.

N parameter
We start by discussing N. We can see the results for different value of the parameter in the following table:

		Number of samples (N)			
Model	$\#\Delta$	1	10	50	100
Bunny	10^4	2.1	5.3	19.8	38.2
Dragon	10^5	12.5	35.2	140.6	275.3
Buddha	10^6	96.7	97.7	128.0	216.0

Table 6.2: Timings in milliseconds of our method for different models and number of samples N (potato material properties). The other parameters were $L = 1$, $W_s = W_l = 512$, $M = 1000$, $K = 16$.

We tested N in the range $1 - 100$. The test with $N = 1$ was introduced to measure the baseline performance of the method (where the most expensive operation is the generation of the random number for the rotation). We did not use zero samples because otherwise the shader compiler optimizations would have removed the base code as well, making the results not significant.

We note that from the graph the time to render the Dragon with 50 samples is slower that rendering the Buddha with the same parameters, despite the Buddha having 10 times the triangles as the Dragon. This suggests that N and the triangle size of the model are not the only factors involved in the performance. Since the heavy computational step is made on a fragment shader in step 2, the number of pixels involved in the computation matters: the Buddha on average occupies less texture area on the directional cameras, and thus the number of computations performed in total is less that the ones for the Dragon. For a low number of samples, the performance is bound by the rasterization time, so the

Dragon outperforms the Buddha in this case because of the fewer number of triangles.

We tested this changing performance according to the area in the following test: we kept the same settings for the directional cameras, but we tried different Dragon sizes. By size, we mean the size of the Dragon using a different scaling transform matrix.

		Size of the model (units)			
Model	#Δ	1	0.5	0.25	0.125
Dragon	10^5	142.1	74.3	34.9	15.4

Table 6.3: Timings in milliseconds of our method for different model size (potato material properties). The size of the camera is 2 units, and the model is 2 units wide. The camera does not scale with the model. The other parameters were $N = 50$, $L = 1$, $W_s = W_l = 512$, $M = 1000$, $K = 16$.

The Dragon, by changing size, occupies a different area of the camera, and so the algorithm performs differently according to the Dragon size. A degrading in quality of the final result is also visible, because of the less number of pixels involved. This problem should be addressed in an eventual extension to the implementation, that should account for these problems in a optimal camera placement.

W_s parameter
The next test we performed was on the size of the radiance map.

		Size of the radiance map (W_s)		
Model	#Δ	256	512	1024
Bunny	10^4	11.4	20.0	39.0
Dragon	10^5	75.4	142.1	299.4
Buddha	10^6	98.2	127.0	258.2

Table 6.4: Timings in milliseconds of our method for different models and size of the radiance map W_s (ketchup material properties). The other parameters were $N = 50$, $L = 1$, $W_l = 512$, $M = 1000$, $K = 16$.

As we can see from table 6.4, the performance scales linearly with the radiance map size. This means that if we double the size of the radiance map, the time

to render the model roughly doubles. This comes from the fact that most of the time in the computation is spent in rendering to the radiance map, that is a pixel bound operation. So if we double the size of map, we roughly double the number of pixels involved and thus the number of the fragment shader invocations in step 2.

K parameter
The next test is about the number of directions used to render the model, K.

		Number of directions (K)			
Model	#Δ	4	8	16	32
Bunny	10^4	6.6	10.0	20.1	42.1
Dragon	10^5	36.7	70.1	143.0	306.2
Buddha	10^6	32.5	55.8	128.3	363.5

Table 6.5: Timings in milliseconds of our method for different models and different number of directions K (ketchup material properties). The other parameters were $N = 50$, $L = 1$, $W_s = W_l = 512$, $M = 1000$, $q = 1$.

We can see that the performance for the models scales linearly. We can see from this test that is a big advantage to place the cameras manually, instead of relying on an automatic algorithm. A careful placement of the cameras can cover the whole model evenly with a smaller number of cameras that an automatic placement. In our implementation, we needed to use at least 16 cameras to ensure that most of the models were covered. On the other hand, with careful placement of the cameras, we were able to lower the number of cameras up to 8 in all cases, thus doubling the performance. Also in this test, we can see that the minor area occupation of the Buddha improves its performance for 4,8, and 16 directions. For 32 directions, the number of triangles that actually need to be rasterized ($32 \cdot 10^6$) probably becomes so high that memory issues on the GPU become important.

Different types of lights
We tested the different times in performance for different kind of lights. Since the operations performed are roughly the same (the point light requires an extra division), also is the performance:

We times a third case, that is when we strip the outer loop in listing 5.12, thus obtaining a shader suitable for only one light. In this case, we can see that the performance slightly improves by fractions of a millisecond.

Model	#Δ	Type of light		
		Point light	Directional light	No loop
Dragon	10^5	61.8	60.3	60.1

Table 6.6: Timings in milliseconds of our method for different types of light (potato material properties), one light. The other parameters were $N = 20$, $W_s = W_l = 512$, $M = 1000$, $K = 16$.

Different materials

We finally tried to see if different materials gave a different performance on the method. As we expected, the material does not influence the rendering time, as the computations in the BSSRDF do not actually take advantage on the material type to easy any computation. We can see the test result in the following table:

Model	#Δ	Material			
		Ketchup	Beer	White grapefruit	Potato
Dragon	10^5	61.7	61.6	62.0	62.1

Table 6.7: Timings in milliseconds of our method for different materials. The other parameters were $N = 20$, $L = 1$, $W_s = W_l = 512$, $M = 300$, $K = 16$.

6.3.3 Tests on environment lighting

The considerations so far discussed apply also in environment map lighting, that we converted to a certain number L of directional lights. So discussing the impact on performance of multiple lights and the impact of environment lighting is essentially the same, apart from a initialization delay to generate the lights from the environment map that we will not consider.

The timings for different kind of lights are reported in the following table: As

		Number of lights L				
Model	$\#\Delta$	1	2	4	8	16
Dragon	10^5	96.6	98.0	100.0	103.8	110.5

Table 6.8: Timings in milliseconds of our method for different number of lights (environment map approximation, material properties for potato). The other parameters were $N = 32$, $W_s = W_l = 512$, $M = 300$, $K = 16$.

we can see, even if we maintain the same number of samples ($N = 32$, so the number of samples per light changes) the performance worsen increasing the number of lights. To look at this into more detail, we broke down the timings again in the different steps. We can see in Figure 6.16 that the increase in timing is due to step 1, the render to lightmap: in fact, we have to render the model once for each light, that implies a performance penalty.

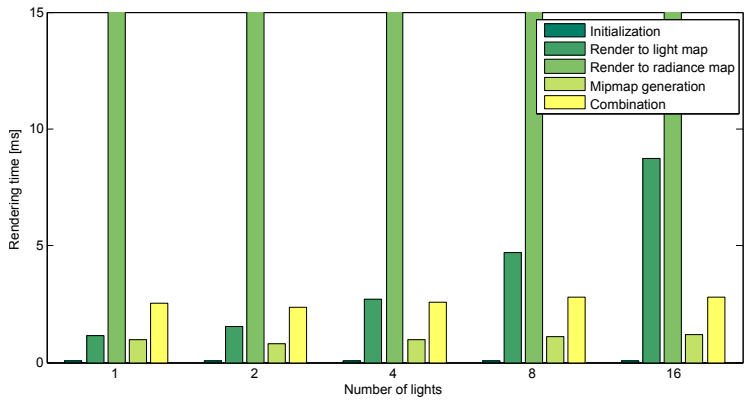


Figure 6.16: Rendering times for the environment lighting of Figure 6.8, split into the various components. The rendering times of step 2 are not shown, but they are [91.8 93.3 93.5 95.0 97.1].

Chapter 7

Future work

In this chapter we discuss the improvements that can be done to our method in order to improve the solution.

7.1 Improving the quality

As we have seen the results section, catching the right effect balancing absorption and scattering requires a lot of tweaking of the parameters M and q. This depends from the fact that an exponential sampling based on σ_{tr} is not probably the best suited for the solution, thus providing good results. A further investigation of optimal sampling patterns would be required in order to obtain faithful results.

An idea we would like to explore is to use two separate disks with a different sampling radius, one accounting the absorption and one for the scattering contribution. The size of the radius of the two disk should be related to the scattering parameters, such as σ_a and σ_s. Further investigation is required in this realm in order to obtain results applicable to a wide range of participating media.

Another direction we would like to explore is the automatic placing of the cameras. Because of the variety and possible concavity of meshes, it is very difficult to place cameras in a way to ensure the complete coverage of the object. The field in which to find ideas in order to fully cover the object would be 3D mesh acquisition using cameras. In this way, it will be possible for a user to avoid the tedious operation of placing the cameras in order to get a full coverage of the object.

Another possibility in this area is to rotate the cameras during the rendering phase: in this case, fewer cameras will be necessary, but then an accumulation process would not be possible in the way we implemented it. It would require a great change to the way the algorithm works in order to get a solution.

7.2 Improving the performance

Regarding the implementation, we tought about some possible extensions in order to make the implementation faster and more reliable.

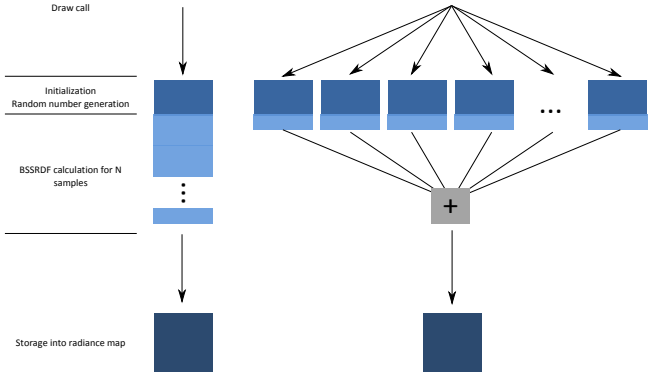


Figure 7.1: Different approaches to rendering to the radiance map: on the left, our approach (one random number generation, multiple samples in the same shader invocation). On the right, the proposed approach: smaller shader invocations, that can be distributed on the GPU.

The first thing we have thought about would be to change the loop in step 2. The idea would be to have instead to have a single fragment shader invocation with cycling N times on the samples, to have N separate fragment shader invocations that calculate the result. The single invocations can be summed over using blending or atomic operations available since OpenGL 4.2. This would allow the GPU more flexibility in scheduling its work. However, we have seen in 7.1

that also the random number generation is an expensive operation for the GPU, and that would have to be repeated for each of the N fragments, potentially undermining the advantages of GPU scheduling.

Another improvement in order to make the performance more reliable would be to make the cameras more adaptive. At the moment, the size of the light and directional cameras is fixed and set up by the user. A simple extension to the method would be to make the camera size adaptive, in order to adapt to the object bounding box. This would make the area in the radiance map texture space more consistent regarding to the size of the object, leading to a more predictable result.

Chapter 8

Conclusions

In this thesis, we have presented an approch to rendering translucent materials using a new directional subsurface scattering BSSRDF models. We focused on creating a method that renders translucent materials in real-time, thus approximating well path traced results.

In Chapter 2 we gave an overview of the different approaches to rendering subsurface scattering in literature, identifying the gap in the knowledge in rendering translucent materials efficiently using a directional approach.

In Chapter 3, we instroduced the mathematical concepts that are needed to give a basic understanding of light transport theory and BSSRDF models, introducing the nthe formulation of two BSSRDF dipole models, the standard dipole model and the directional dipole model. Using this knowledge, we gave the reader a theoretical introduction to our method, as well as scattering parameters and how they are acquired in Chapter 4. Chapter 4 provided an essential bridge between theory and implementation, that we described in Chapter 5. In the implementation, we described a method that employs the advantages of the formulation of the directional dipole in order to create a robust method that implements the directional dipole taking advantage of the GPU rendering pipeline.

In Chapter 6, we compared our result to path traced solutions, proving that

our method can produce results that approximate well path-traced solutions, providing speed ups of four order of magnitude compared to path tracing on CPU. We also proved that our method for models of the size commonly used in the computer game industry performs in real-time on a high-end modern GPU, limiting the size of memory used to a minimum.

Finally, in Chapter 7, we introduced some possible ideas to expand our method in order to produce a higher quality result retaining its real-time capabilities.

To sum up, we think that most of the goals of the thesis stated in the introduction Chapter have been satisfied. This thesis hopefully provides an insight on a new way to approach the efficient rendering of translucent materials using directional subsurface scattering.

Appendix A

Model matrices

In this chapter, we report the different model, view and projection matrices formulas we used in our mater thesis.

A.1 Model matrices

Translation matrix

$$T(\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & \mathbf{t}_x \\ 0 & 1 & 0 & \mathbf{t}_y \\ 0 & 0 & 1 & \mathbf{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation matrix

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale matrix

$$S(\mathbf{s}) = \begin{bmatrix} \mathbf{s}_x & 0 & 0 & 0 \\ 0 & \mathbf{s}_y & 0 & 0 \\ 0 & 0 & \mathbf{s}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.2 View Matrix

e is the camera position, **d** is the camera direction, **u** is the up vector.

$$\begin{aligned} \vec{c} &= -\frac{\mathbf{d}}{\|\mathbf{d}\|} \\ \vec{a} &= \frac{\vec{c} \times \mathbf{u}}{\|\vec{c} \times \mathbf{u}\|} \\ \vec{b} &= \vec{a} \times \vec{c} \end{aligned}$$

$$V(\mathbf{e}, \mathbf{d}, \mathbf{u}) = \begin{bmatrix} \vec{a}_x & \vec{a}_y & \vec{a}_z & -\mathbf{e}_x \\ \vec{b}_x & \vec{b}_y & \vec{b}_z & -\mathbf{e}_y \\ \vec{c}_x & \vec{c}_y & \vec{c}_z & -\mathbf{e}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.3 Projection Matrix

Orthographic matrix

$\mathbf{l}$ is the left bottom near corner of the frustum, $\mathbf{r}$ is the top right far corner.

$$O(\mathbf{l},\mathbf{r}) = \begin{bmatrix} \frac{2}{\mathbf{r}_x-\mathbf{l}_x} & 0 & 0 & -\frac{\mathbf{r}_x+\mathbf{l}_x}{\mathbf{r}_x-\mathbf{l}_x} \\ 0 & \frac{2}{\mathbf{r}_y-\mathbf{l}_y} & 0 & -\frac{\mathbf{r}_y+\mathbf{l}_y}{\mathbf{r}_y-\mathbf{l}_y} \\ 0 & 0 & -\frac{2}{\mathbf{r}_z-\mathbf{l}_z} & -\frac{\mathbf{r}_z+\mathbf{l}_z}{\mathbf{r}_z-\mathbf{l}_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective matrix

fov is the field of view in radians, A is the aspect ratio, n_p is the near camera plane and f_p is the far camera plane. We define:

$$f_c = \frac{1}{\tan(\frac{fov}{2})}$$

$$P(fov, A, n, f) = \begin{bmatrix} \frac{f_c}{A} & 0 & 0 & 0 \\ 0 & f_c & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Appendix B

Directional dipole GPU code

```
const float EPSILON_MU = 0.0f;
const vec3 one = vec3(1.0f);

vec3 S_infinite(vec3 _r, vec3 _r_sq, float x_dot_w12, float no_dot_w12, ↩
    float x_dot_no)
{
    vec3 _r_tr = transmission * _r;
    vec3 _r_tr_p1 = _r_tr + one;
    vec3 _T = exp(-_r_tr);
    vec3 coeff = _T / (_r * _r_sq);
    vec3 first = C_s * ( _r_sq * D_rev + 3 * _r_tr_p1 * x_dot_w12);
    vec3 second = C_e * ( three_D * _r_tr_p1 * no_dot_w12 - (_r_tr_p1 + ↩
        three_D * (3 *_r_tr_p1 + _r_tr * _r_tr) / (_r_sq) * x_dot_w12) * ↩
        x_dot_no);
    vec3 _S = coeff * (first - second);
    return _S;
}

vec3 S_infinite_vec(vec3 _r, vec3 _r_sq, vec3 x_dot_w12, float no_dot_w12, ↩
    vec3 x_dot_no)
{
    vec3 _r_tr = transmission * _r;
    vec3 _r_tr_p1 = _r_tr + one;
    vec3 _T = exp(-_r_tr);
    vec3 coeff = _T / (_r * _r_sq);
    vec3 first = C_s * ( _r_sq * D_rev + 3 * _r_tr_p1 * x_dot_w12);
    vec3 second = C_e * ( three_D * _r_tr_p1 * no_dot_w12 - (_r_tr_p1 + ↩
        three_D * (3 *_r_tr_p1 + _r_tr * _r_tr) / (_r_sq) * x_dot_w12) * ↩
        x_dot_no);
    vec3 _S = coeff * (first - second);
    return _S;
}
```

```
vec3 bssrdf(vec3 _xi,vec3 _nin,vec3 _wi,vec3 _xo, vec3 _no)
{
    vec3 _x = _xo - _xi;
    float r_sqr = dot(_x,_x);
    vec3 _w12 = refract2(_wi,_nin);

    float mu = -dot(_no, _w12);
    float dot_x_w12 = dot(_x,_w12);
    float dot_x_no = dot(_x,_no);

    vec3 _r_sqr = r_sqr.xxx;
    vec3 _dr_sqr = _r_sqr;

    float edge = step(mu, EPSILON_MU); // == 1.0 if mu > EPSILON_MU
    vec3 project = vec3(sqrt(r_sqr - dot_x_w12 * dot_x_w12)/sqrt(_r_sqr + ↩
        de_sqr));
    vec3 _D_prime = abs(mu) * D * edge + one_over_three_ext * (1.0f - edge)↩
        ;
    _dr_sqr += _D_prime * (_D_prime + two_de * project * edge);

    vec3 _dr = sqrt(_dr_sqr);

    float edge_nistar = step(abs(dot_x_no),0.01f); // 1.0 if dot_x_no > ↩
        0.01
    vec3 _t = normalize(cross(_nin,_x));
    vec3 _nistar = _nin * edge_nistar + cross(normalize(_x),_t) * (1.0f - ↩
        edge_nistar);

    mat3 _xov = mat3(_x,_x,_x) - outerProduct(_nistar, two_a_de);
    vec3 _dv_sqr = vec3(dot(_xov[0],_xov[0]),dot(_xov[1],_xov[1]),dot(_xov↩
        [2],_xov[2]));
    vec3 _dv = sqrt(_dv_sqr);
    vec3 _wv = _w12 - 2 * dot(_w12,_nistar) * _nistar;

    vec3 _x_dot_wv = _wv * _xov;
    vec3 _x_dot_no = _no * _xov;

    vec3 _S_r = S_infinite(_dr, _dr_sqr, dot_x_w12, -mu, dot_x_no);
    vec3 _S_v = S_infinite_vec(_dv, _dv_sqr, _x_dot_wv, dot(_no,_wv), ↩
        _x_dot_no);
    vec3 _S = _S_r - _S_v;

    _S *= fresnel_T(_wi,_nin);
    _S = max(vec3(0.0f),_S);

    return _S;
}
```

Listing B.1: Directional dipole code optimized for GPU. Uniforms and Fresnel formulas code is not reported.

Bibliography

Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

Michael Ashikmin, Simon Premože, and Peter Shirley. A microfacet-based brdf generator. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 65–74, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. URL `http://dx.doi.org/10.1145/344779.344814`.

James Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977. URL `http://doi.acm.org/10.1145/965141.563893`.

Jesper Børlum, Brian Bunch Christensen, Thomas Kim Kjeldsen, Peter Trier Mikkelsen, Karsten Østergaard Noe, Jens Rimestad, and Jesper Mosegaard. Sslpv: Subsurface light propagation volumes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 7–14, New York, NY, USA, 2011. ACM. URL `http://doi.acm.org/10.1145/2018323.2018325`.

Max Born and Wolf Emil. *Principles of Optics*. Cambridge University Press, 7 edition, 1999.

Subrahmanyan Chandrasekar. *Radiative Transfer*. Oxford University Press, 1950.

Michael Cohen, John Wallace, and Pat Hanrahan. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

Carsten Dachsbacher and Marc Stamminger. Translucent shadow maps. In *Proceedings of the 14th Eurographics Workshop on Rendering*, EGRW '03,

pages 197–201, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. URL http://dl.acm.org/citation.cfm?id=882404.882433.

Boris Davison and John Bradbury Sykes. *Neutron transport theory*. Oxford University Press, 1958.

Eugene D'Eon. A better dipole (a publicly available manuscript). Technical report, -, 2012. URL .http://www.eugenedeon.com/papers/betterdipole.pdf.

Eugene D'Eon and Geoffrey Irving. A quantized-diffusion model for rendering translucent materials. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 56:1–56:14, New York, NY, USA, 2011. ACM. URL http://doi.acm.org/10.1145/1964921.1964951.

Eugene d'Eon, David Luebke, and Eric Enderton. Efficient rendering of human skin. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, pages 147–157, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. URL http://dx.doi.org/10.2312/EGWR/EGSR07/147-157.

Agns Desolneux, Lionel Moisan, and Jean-Michel Morel. *From Gestalt Theory to Image Analysis: A Probabilistic Approach*. Springer Publishing Company, Incorporated, 1st edition, 2007.

Craig Donner. *Towards Realistic Image Synthesis of Scattering Materials*. PhD thesis, University of California, San Diego, La Jolla, CA, USA, 2006. AAI3226771.

Craig Donner and Henrik Wann Jensen. Light diffusion in multi-layered translucent materials. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1032–1039, New York, NY, USA, 2005. ACM. URL http://doi.acm.org/10.1145/1186822.1073308.

Craig Donner, Jason Lawrence, Ravi Ramamoorthi, Toshiya Hachisuka, Henrik Wann Jensen, and Shree Nayar. An empirical bssrdf model. *ACM Trans. Graph.*, 28(3):30:1–30:10, July 2009. URL http://doi.acm.org/10.1145/1531326.1531336.

Julie Dorsey, Alan Edelman, Henrik Wann Jensen, Justin Legakis, and Hans Køhling Pedersen. Modeling and rendering of weathered stone. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 225–234, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. URL http://dx.doi.org/10.1145/311535.311560.

D'Eon Eugene and Luebke David. Chapter 14. advanced techniques for realistic real-time skin rendering. In *GPU Gems 3*. Addison-Wesley Professional, 2007.

Cass Everitt, Ashu Rege, and Cem Cebenoyan. Hardware shadow mapping. Technical report, NVIDIA Corporation, 2003.

Raanan Fattal. Participating media illumination using light propagation maps. *ACM Trans. Graph.*, 28(1):7:1–7:11, February 2009. URL `http://doi.acm.org/10.1145/1477926.1477933`.

Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

J. R. Frisvad, T. Hachisuka, and T. K. Kjeldsen. Directional dipole for subsurface scattering in translucent materials. *ACM Transactions on Graphics, 2014*, -:–, 2014. URL `http://www2.imm.dtu.dk/pubdb/p.php?6646`. To appear.

Robin Green. Spherical harmonic lighting: The gritty details, January 2003. Sony Computer Entertainment America.

Simon Green. Chapter 16. real-time approximations to subsurface scattering. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, December 1964. URL `http://doi.acm.org.globalproxy.cvt.dk/10.1145/355588.365104`.

Akira Ishimaru. *Wave propagation and scattering in random media.* IEEE, 1997.

ITU. Parameter values for the hdtv standards for production and international programme exchange. Technical Report ITU-R BT.709-2, ITU-R, June 2001. URL `http://www.itu.int/rec/R-REC-BT.709-2-199510-S/en`.

Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. *ACM Trans. Graph.*, 21(3):576–581, July 2002. URL `http://doi.acm.org/10.1145/566654.566619`.

Henrik Wann Jensen and Per H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 311–320, New York, NY, USA, 1998. ACM. URL `http://doi.acm.org/10.1145/280814.280925`.

Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 511–518, New York, NY, USA, 2001. ACM. URL `http://doi.acm.org/10.1145/383259.383319`.

Jorge Jimenez, Veronica Sundstedt, and Diego Gutierrez. Screen-space perceptual rendering of human skin. *ACM Trans. Appl. Percept.*, 6(4):23:1–23:15, October 2009. URL http://doi.acm.org/10.1145/1609967.1609970.

J. H. Joseph, W. J. Wiscombe, and J. A. Weinman. The delta-Eddington approximation for radiative flux transfer. *Journal of Atmospheric Sciences*, 33:2452–2459, December 1976.

James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4): 143–150, August 1986. URL http://doi.acm.org/10.1145/15886.15902.

Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM. URL http://doi.acm.org/10.1145/1730804.1730821.

Douglas Scott Kay and Donald Greenberg. Transparency for computer synthesized images. *SIGGRAPH Comput. Graph.*, 13(2):158–164, August 1979. URL http://doi.acm.org/10.1145/965103.807438.

Takahiro Kosaka, Tomohito Hattori, Hiroyuki Kubo, and Shigeo Morishima. Rapid and authentic rendering of translucent materials using depth-maps from multi-viewpoint. In *SIGGRAPH Asia 2012 Posters*, SA '12, pages 45:1–45:1, New York, NY, USA, 2012. ACM. URL http://doi.acm.org/10.1145/2407156.2407206.

Hendrik P. A. Lensch, Michael Goesele, Philippe Bekaert, Jan Kautz, Marcus A. Magnor, Jochen Lang, and Hans-Peter Seidel. Interactive rendering of translucent objects. In *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*, PG '02, pages 214–, Washington, DC, USA, 2002. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=826030.826632.

S. Menon, Q. Su, and R. Grobe. Determination of g and μ using multiply scattered light in turbid media. *Phys. Rev. Lett.*, 94:153904, Apr 2005. URL http://link.aps.org/doi/10.1103/PhysRevLett.94.153904.

Tom Mertens, J. Kautz, P. Bekaert, F. Van Reeth, and H.-P. Seidel. Efficient rendering of local subsurface scattering. In *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, pages 51–58, Oct 2003a.

Tom Mertens, Jan Kautz, Philippe Bekaert, Hans-Peter Seidelz, and Frank Van Reeth. Interactive rendering of translucent deformable objects. In *Proceedings of the 14th Eurographics Workshop on Rendering*, EGRW '03, pages 130–140, Aire-la-Ville, Switzerland, Switzerland, 2003b. Eurographics Association. URL http://dl.acm.org/citation.cfm?id=882404.882423.

Srinivasa G. Narasimhan, Mohit Gupta, Craig Donner, Ravi Ramamoorthi, Shree K. Nayar, and Henrik Wann Jensen. Acquiring scattering properties of participating media by dilution. *ACM Trans. Graph.*, 25(3):1003–1012, July 2006. URL `http://doi.acm.org/10.1145/1141911.1141986`.

F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometrical considerations and nomenclature for reflectance. In Lawrence B. Wolff, Steven A. Shafer, and Glenn Healey, editors, *Radiometry*, chapter Geometrical Considerations and Nomenclature for Reflectance, pages 94–145. Jones and Bartlett Publishers, Inc., USA, 1992. URL `http://dl.acm.org/citation.cfm?id=136913.136929`.

H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, 1992.

Matt Pharr and Pat Hanrahan. Monte carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 75–84, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. URL `http://dx.doi.org/10.1145/344779.344824`.

Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 497–500, New York, NY, USA, 2001. ACM. URL `http://doi.acm.org/10.1145/383259.383317`.

Marc Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*, version 4.3 (core profile) edition, August 2012.

Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *SIGGRAPH Comput. Graph.*, 26(2):249–252, July 1992. URL `http://doi.acm.org.globalproxy.cvt.dk/10.1145/142920.134071`.

M.A. Shah, J. Konttinen, and S. Pattanaik. Image-space subsurface scattering for interactive rendering of deformable translucent objects. *Computer Graphics and Applications, IEEE*, 29(1):66–78, Jan 2009.

Peter-Pike Sloan. Stupid spherical harmonics (sh) tricks. In *Game Developers Conference 2008, February 2008 (updated 2/10/2010)*, 2008.

Jos Stam. Multiple scattering as a diffusion process. In PatrickM. Hanrahan and Werner Purgathofer, editors, *Rendering Techniques 1995*, Eurographics, pages 41–50. Springer Vienna, 1995. URL http://dx.doi.org/10.1007/978-3-7091-9430-0_5.

Anna Tomaszewska and Krzysztof Stefanowski. Real-time spherical harmonics based subsurface scattering. In Aurélio Campilho and Mohamed Kamel, editors, *Image Analysis and Recognition*, volume 7324 of *Lecture Notes in Computer Science*, pages 402–409. Springer Berlin Heidelberg, 2012. URL http://dx.doi.org/10.1007/978-3-642-31295-3_47.

K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. In Lawrence B. Wolff, Steven A. Shafer, and Glenn Healey, editors, *Radiometry*, pages 32–41. Jones and Bartlett Publishers, Inc., USA, 1992. URL http://dl.acm.org/citation.cfm?id=136913.136924.

Unity. *Unity Manual: Forward Rendering Path Details*, 2012. URL http://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html.

Yajun Wang, Jiaping Wang, Nicolas Holzschuch, Kartic Subr, Jun-Hai Yong, and Baining Guo. Real-time rendering of heterogeneous translucent objects with arbitrary shapes. *Comput. Graph. Forum*, 29(2):497–506, 2010. URL http://dblp.uni-trier.de/db/journals/cgf/cgf29.html#WangWHSYG10.

Lance Williams. Casting curved shadows on curved surfaces. *International Conference on Computer Graphics and Interactive Techniques*, -:270–274, 1978.

Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with hammersley and halton points. *J. Graphics, GPU, & Game Tools*, 2(2):9–24, 1997. URL http://dblp.uni-trier.de/db/journals/jgtools/jgtools2.html#WongLH97.