Learning Visibility in Ray Space

by

Joey Litalien

Department of Electrical and Computer Engineering
McGill University, Montréal QC
April 2019

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Engineering (M.Eng.)

Résumé

Les ombres douces générées par des lumières surfaciques sont essentielles en synthèse d'images réalistes, mais elles requièrent le traçage de rayons secondaires afin d'évaluer la visibilité. Ce calcul de lumière directe est dispendieux et est souvent remplacé par des techniques d'approximation en rendu temps réel. Dans ce mémoire, nous investiguons l'utilisation de méthodes d'apprentissage profond pour modéliser le problème de visibilité. En traitant l'occlusion comme une tâche de classification binaire, nous entraînons un réseau de neurones artificiels pouvant estimer la visibilité directionnelle dans l'espace des rayons. Nous testons notre modèle sur différents objets maillés avec plusieurs sources de lumière et démontrons qu'un simple réseau de neurones à propagation avant est partiellement capable de généraliser à des positions de lumières jamais vues. Malheureusement, notre modèle rencontre des difficultés lors de la reconstruction d'ombres lointaines et ne peut donc pas simuler ces régions dans toutes les directions.

Abstract

Soft shadows from area lights are essential in generating compelling photorealistic images but require tracing secondary rays to evaluate visibility for direct lighting. This computation is costly and, as such, is usually replaced by image- or geometry-based approximations for real-time rendering. In this work, we investigate the use of deep learning techniques to solve the visibility problem. By treating occlusion as a binary classification task, we train a per-object artificial neural network that estimates the directional visibility profile in ray space. We test on low-to-moderate complexity meshes with different light sources and show that a simple feedforward neural network classifier is partially capable of generalizing to unseen light positions. Unfortunately, our model has difficulties reconstructing shadows past a certain distance and cannot accurately resolve shadows in all directions.

À mes parents, France et Marko

Preface

This thesis investigates the use of deep learning to solve the visibility problem in the context of soft shadow generation for interactive rendering. It is ultimately based on the work I have done in the second half of 2017 where I explored the idea of using a deep neural network to learn the visibility profile of simple mesh objects. When this project originally started, hardware-based acceleration solutions for ray tracing were not yet publicly available. The introduction of real-time ray tracing in mid-2018 by NVIDIA® and Microsoft® provided a watershed moment to the field of real-time rendering. In particular, it allowed for reflections and soft shadows to be accurately generated on-the-fly on the GPU for the very first time. While it can be argued that this considerably weakens the importance of this thesis, I still believe the experimental results obtained, albeit unsuccessful, present promising avenues of research for hybrid rendering.

Acknowledgements. First and foremost, I would like to express my sincere gratitude to my supervisor Derek Nowrouzezahrai, who played a crucial role in the early stage of this project. Derek is a tremendously supportive mentor who was always available to answer my questions and provide insightful guidance throughout the process. Most importantly, he instilled the importance of robust research methodologies early on in my research career, and this is something I am truly thankful for.

On a personal level, I would like to thank my parents France and Marko for their continuous support on my journey to become an applied scientist. I'd also like to thank Laurence for her patience and for listening to my fuzzy ideas throughout my two years of masters. Her warm company has helped me keeping my sanity in rougher times, especially when I had to miss four months of research due to a back surgery. She kept me on track and much of this thesis could not have been written without her support. *Je ne te remercierai jamais assez.*[1]

I also wish to thank those who helped me along the way, especially my close friends and once roommates Nicolas, Erick, Samuel and Renaud (also my unofficial typography consultant). When we first met in the gloomy lecture hall of Rutherford Building back in 2012, I wouldn't have

[1] The days of you facetiously claiming to have a higher level of education than me are finally over!

guessed that our relationship would haved blossomed into such a sincere and unique friendship. To all fellow members of the Lyapunov Club[2], thank you.

The McGill Graphics and Imaging Lab has a bright future and I am glad to be part of its first batch of graduate students. The GIL has allowed me to meet many devoted and fascinating people, including Keven, Nicolas, Fan, Damien, Chaitanya, Luis, and Adrien. I personally want to thank Keven for his moral support during the last month of writing this thesis, and Nicolas for his software engineering expertise he kindly and willingly shared with me in the past two years.

[2]The official Facebook group chat since 2015.

Contents

List of Figures

Chapter 1

Introduction

Synthesizing the appearance of the real world is an important and long-standing goal of computer graphics. In physically-based rendering, light transport algorithms aim to render realistic images that are indistinguinsable from photographs. The ever-growing demand for realism and visual fidelity in the feature film and video games industry has pushed the rendering field to continuously develop new algorithms that can accurately and efficiently simulate the world we live in. The ambitious nature of this research field has revealed that, even if we can now synthesize believable images that can trick the most meticulous of audiences, we are still very far away from reproducing this illusion at interactive framerates. This is why visual effects in blockbuster movies are usually more convincing than the best looking video games, even in 2018.[1]

Due to past hardware limitations, *offline* and *online* (real-time) rendering have essentially grown into two distinct research fields over the past few decades. While the former enjoys simulating the physics of light without critical time constraints, the latter only has a few milliseconds to output an image to the screen. Approximation algorithms for real-time applications are thus employed to alleviate this bottleneck, effectively trading visual quality for performance. As a result, full global illumination is still an ongoing quest in games and interactive films, and the majority of research in real-time rendering stems from the difficulty of simulating dynamical light transport efficiently.

[1] Of course, this depends on the type of movies you watch!

Lately, real-time rendering has gained extraordinary momentum with the release of hardware-based acceleration solutions for ray tracing, a method for propagating light rays in a virtual scene. Coupled with the recent advances in deep learning, this major shift has significantly narrowed the gap between offline and interactive techniques. Concretely, it has opened up a sea of promising research possibilities to navigate while bringing together all prominent actors of the field, such as feature film production scientists and video games developers. The goal of this thesis is to explore one such idea.

The primary focus of this thesis is *shadows*. More precisely, we are interested in the generation of *soft* shadows from area light sources. These shadowed regions are essential when synthesizing images since they provide critical depth cues and information about the surrounding lighting environment (Figure 1.1). Scenes without shadows can often look unconvincing and are generally more difficult to parse. This creates a need for simulating shadowed regions in a computationally efficient way.

(a) Rockstar's *Grand Theft Auto V* (2013) **(b)** CD Projekt's *Witcher 3* (2015)

Figure 1.1. Shadows in a virtual environment are a visual necessity to provide a feeling of immersion to the player. The shadows casted by the characters and the trees make the scene believable. *Images* © Rockstar Games® and CD Projekt® Red.

The core challenge in computing shadows emerges from the necessity of tracing secondary rays *on-the-fly* to evaluate the mutual visibility between two points. This, in turn, entails intersecting rays with the scene geometry by the mean of a spatial data structure. Although some optimizations of this tree structure exist, its evaluation remains too slow, which can make it impractical for real-time usage. Several approximate interactive methods have been developed throughout the years, but—although computationally inexpensive—they are generally not very

robust and their imprecise behaviour is often obvious. In any case, these approaches cannot scale to large scenes without incurring unacceptable performance penalties. This motivates the need for a model that would be both robust and fast enough for interactive scene relighting. In this work, we propose to investigate deep learning techniques to solve this problem.

1.1 Thesis Overview

This thesis consists of six chapters. In Chapter 2 we review the fundamentals of light transport and describe the light transport problem. We also provide a general overview of deep learning techniques that shall be used later on. In Chapter 3 we describe pratical solutions for generating soft shadows in offline and real-time scenarios, and we also review the state-of-the-art in shadow generation. These two chapters will lay the foundations for our work and will also introduce the necessary notation and terminology to reason about visibility. In Chapter 4 we present a framework for learning the visibility of simple mesh objects. In particular, we show that it is possible to sample the visibility of simple convex meshes and learn this distribution using a deep neural network. We present our experimental results in Chapter 5 and conclude with an analysis of the learning process in Chapter 6.

Chapter 2

Background

There are many models of the physics of light with various levels of complexity. Spectral rendering models treat the light as a wave and thus treat the spatial-temporal propagation of light, while simpler models rely on geometric optics to trace out the light trajectories. The former relies on a more accurate representation of light, but their efficient implementation is often intractable. The latter techniques, on the other hand, amount to the corpuscular theory of light and thus assume that light travel in a straight line. This important simplification allows for the decoupling of the RGB channels for an efficient simulation, but cannot handle wave optics phenomenom like diffraction and interference. These events are unperceivable from the human eye since the wavelength of light is negligible compared to the size of objects encountered in a scene. Consequently, modern rendering frameworks prefer the classical approach of geometric optics to model the visible light only. This level of abstraction allows to concentrate the focus on the phenomena of interest when developing new light transport algorithms or appearance models.

In this chapter, we first introduce the fundamentals of light transport. In particular, we present a general overview of physically-based rendering, including radiometry, reflectance models, and Monte Carlo techniques for solving the rendering equation. Given the nature of the problem we are trying to solve, we will proceed to review deep neural networks in the context of classification and describe how these models can be trained to approximate arbitrary nonlinear functions. In particular, we will see how artifical neural networks are robust tools for learning complex distributions from big data.

2.1 Domains and Measures

We introduce several important domains and related measures to facilitate mathematical formulations discussed in the following text.

We assume that the scene to be rendered is made of finite two-dimensional surfaces embedded in $\mathbf{R}^3$. The union of these surfaces forms the *scene manifold* and shall be denoted by $\mathcal{M} \subseteq \mathbf{R}^3$. A point on this union of surfaces will be denoted by $\mathbf{x} \in \mathcal{M}$. We begin by defining the *area measure* A on $\mathcal{M}$ such that $A(D)$ represents the surface area of some region $D \subseteq \mathcal{M}$. We can take the (Lebesgue) integral over all such regions, assuming $f : \mathcal{M} \to \mathbf{R}$ is integrable:

$$F = \int_{\mathcal{M}} f(\mathbf{x}) \, \mathrm{d}A(\mathbf{x}). \tag{2.1}$$

If f is the identity function then F is the total surface area of the scene, which amounts to the sum of all the triangle areas. We can also define F with respect to directions. If we consider normalized directions ω in the unit sphere $\mathcal{S}^2$, we can define the *solid angle measure* σ which corresponds to the surface area measure on the sphere. Representing directions in polar coordinates allows us to integrate over solid angles:

$$F = \int_{\mathcal{S}^2} f(\omega) \, \mathrm{d}\sigma(\omega) = \int_0^{2\pi} \int_0^{\pi} f(\theta, \phi) \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi. \tag{2.2}$$

The *projected solid angle measure*, denoted $\sigma^\perp$ and also defined on $\mathcal{S}^2$, is used to integrate a physical quantity that arrives at or leaves a surface point, and as such, introduces a foreshortening (cosine) factor:

$$\int_{\mathcal{S}^2} f(\omega) \, \mathrm{d}\sigma^\perp(\omega) = \int_0^{2\pi} \int_0^{\pi} f(\theta, \phi) |\cos\theta| \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi, \tag{2.3}$$

where we use the notation $|\cdot|$ to denote the clamping $\max(0, x)$. Hence, the differential $\mathrm{d}\sigma^\perp$ can be interpreted as the perpendicular projection of the solid angle on the hemisphere $\mathcal{H}^2(\mathbf{x})$ onto its basis disk (Figure 2.1). In the formalism of Veach [Vea98], we finally define the *throughput measure* as the product of both measures. Let $\mathcal{R} := \mathcal{M} \times \mathcal{S}^2$ denote the *ray space*, *i.e.*, the space

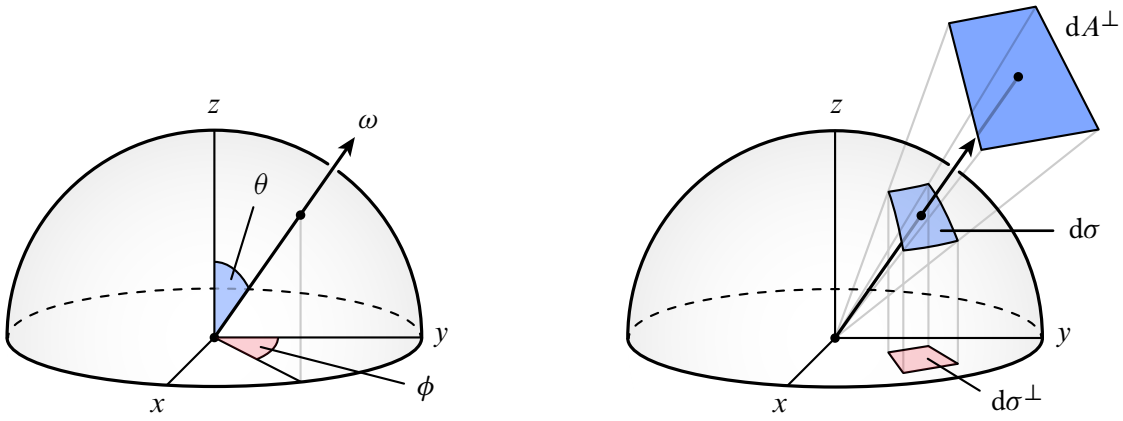


Figure 2.1. Spherical coordinates (*left*) and the relationship between the various differential quantities introduced so far (*right*). The perpendicular differential area $\mathrm{d}A^\perp$ corresponds to the differential area $\mathrm{d}A$ projected onto the plane perpendicular to ω, and the projected differential solid angle $\mathrm{d}\sigma^\perp$ can be viewed as the projection of $\mathrm{d}\sigma$ onto the equatorial tangent plane. These differentials are related by $\mathrm{d}A^\perp\,\mathrm{d}\sigma = \mathrm{d}A\,\mathrm{d}\sigma^\perp$.

of all rays that start at points on the scene surfaces. Then, for any $D \subseteq \mathcal{R}$ we define μ as

$$\mu(D) := \int_D \mathrm{d}A(\mathbf{x})\,\mathrm{d}\sigma^\perp(\omega). \tag{2.4}$$

Intuitively, we can think of μ as measuring the light-carrying capacity of a bundle of light rays. When convenient and unambiguous, we will write $\mathrm{d}\omega$ instead of $\mathrm{d}\sigma(\omega)$ and $\mathrm{d}\omega^\perp$ instead of $\mathrm{d}\sigma^\perp(\omega)$ for conciseness. We now turn to radiometry to reason about the quantities these rays carry.

2.2 Radiometry

Radiometry is the study of the measurement of the electromagnetic radiation. It is based on the geometric optics, where we consider the light as a movement of the rigid particles. The light spectrum domain will be assumed to be the classical red-green-blue (RGB) color domain $\mathbf{R}^3_+$, namely positive real-valued channels including zero (high-dynamic-range).

The quantity of interest in light transport is the *radiance*. Given a point $\mathbf{x}$ and a direction ω, the radiance $L : \mathcal{M} \times \mathcal{H}^2 \subseteq \mathbf{R}^5 \to \mathbf{R}^3_+$ is defined as the flux density Φ per differential solid angle

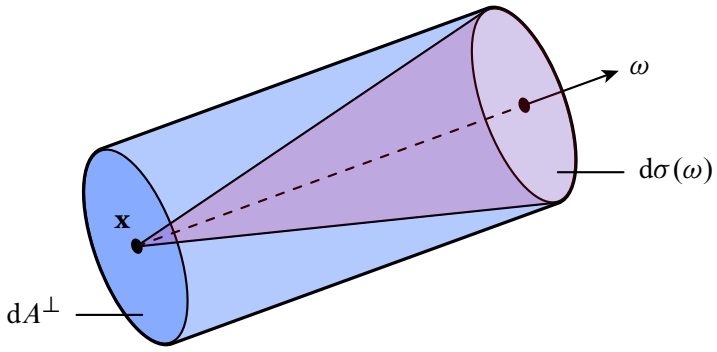


Figure 2.2. Radiance emitted in the direction ω from the point $\mathbf{x}$ is defined as the amount of energy traveling through the differential beam defined by the projected area $\mathrm{d}A^{\perp}$ perpendicular to ω and the solid angle $\mathrm{d}\sigma(\omega)$.

$\mathrm{d}\sigma$ and differential perpendicular area $\mathrm{d}A^{\perp}$ (Figure 2.2). More precisely, we can write this as

$$L(\mathbf{x},\omega) := \frac{\mathrm{d}^2\Phi(\mathbf{x},\omega)}{\mathrm{d}\sigma(\omega)\,\mathrm{d}A^{\perp}(\mathbf{x},\omega)}, \tag{2.5}$$

measured in $\mathrm{W/m^2\,sr}$. Let $N : \mathcal{M} \to S^2$ be the Gauss map that associates an oriented unit normal vector to its surface point. Then, the projected differential area is $\mathrm{d}A^{\perp} = \cos\theta\,\mathrm{d}A$, where θ is the angle between $N(\mathbf{x})$ and ω.

Radiance is invariant along a line segment, assuming light travels through a vacuum (*i.e.*, without a participating media) and remains unobstructed:

$$L(\mathbf{x},\omega) = L(\mathbf{x}+t\omega,\omega), \quad t \geqslant 0. \tag{2.6}$$

This key characteristic is crucial to rendering since it allows us to express radiance as a finite sequence of surface interactions. Other radiometric quantities can be expressed in terms of the radiance. For instance, rewriting (2.5) in terms of the *radiant flux* Φ over a surface D yields the cosine-weighted double integral:

$$\Phi = \int_D \int_{\mathcal{H}^2} L(\mathbf{x},\omega)|N(\mathbf{x})\cdot\omega|\,\mathrm{d}\sigma(\omega)\,\mathrm{d}A(\mathbf{x}). \tag{2.7}$$

The *incident radiance* $L_i(\mathbf{x},\omega)$ is defined as the measure of radiance *arriving* at $\mathbf{x}$ from direction

ω while *exitant radiance* $L_o(\mathbf{x}, \omega)$ measures the radiance quantity *leaving* $\mathbf{x}$ in direction ω. In a vacuum, these two quantities are such that

$$L_i(\mathbf{x}, \omega) = L_o(\mathbf{y}, -\omega) = L(\mathbf{x}, \omega), \quad \mathbf{x}, \mathbf{y} \in \mathcal{M}. \tag{2.8}$$

This is obvious by Equation 2.6: since radiance is constant along a unobstructed differential beam, the incoming radiance at point in a given direction is equal to outgoing radiance at another in the opposite direction.

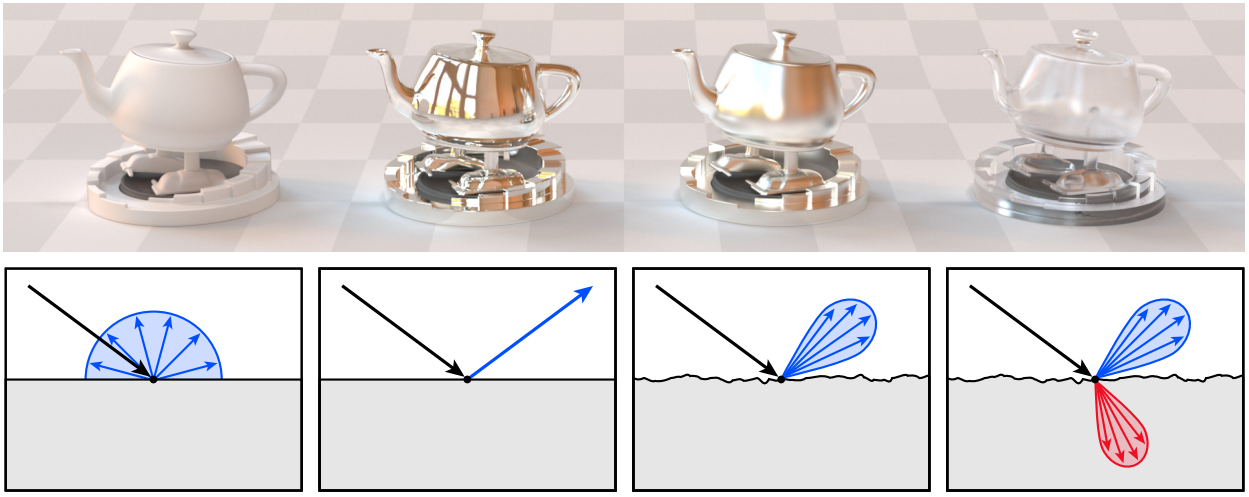
2.3 Light Interactions with Surfaces

Describing the appearance of surfaces under varying illumination requires a robust mathematical model of surface scattering. When light is emitted from a light source, photons travel along a differential beam of light and are either reflected or transmitted at the object surfaces. A formal framework is thus necessary to study the surface's response to illumination, in particular, how light is absorbed or dissipated as heat at an arbitrary surface point. We define the *bidirectional reflectance distribution function* (BRDF) $f_r : \mathcal{H}^2 \times \mathcal{H}^2 \subseteq \mathbf{R}^4 \to \mathbf{R}^3_+$ as the fraction of light arriving at a point from the incoming direction ω_i that gets reflected in the outgoing direction ω_o.[1] This function governs the appearance of an object and is formally defined as

$$f_r(\mathbf{x}, \omega_i, \omega_o) := \frac{\mathrm{d}L_o(\mathbf{x}, \omega_o)}{L_i(\mathbf{x}, \omega_i)\, \mathrm{d}\sigma^{\perp}(\omega_i)} = \frac{\mathrm{d}L_o(\mathbf{x}, \omega_o)}{L_i(\mathbf{x}, \omega_i) \cos\theta_i\, \mathrm{d}\sigma(\omega_i)}, \tag{2.9}$$

where we include the point $\mathbf{x}$ in the notation for clarity. A valid BRDF is always positive and follows the Helmholtz reciprocity principle $f_r(\mathbf{x}, \omega_i, \omega_o) = f_r(\mathbf{x}, \omega_o, \omega_i)$. This entails that reversing the incident and outgoing directions does not change the amount of light being scattered. Moreover, a plausible BRDF must conserve energy. This property is essential to ensure that rendering algorithms converge to the radiant equilibrium: if light can gain energy at every bounce, transported radiance may grow out of bounds as light propagates indefinitely in a scene.

[1] Here, ω_i and ω_o are assumed to be unitary vectors in the hemisphere $\mathcal{H}^2(\mathbf{x})$, that is, the half-sphere centered at $\mathbf{x}$ with normal $N(\mathbf{x})$. See Figure 2.4a for an illustration.



(a) Smooth diffuse **(b)** Smooth conductor **(c)** Rough conductor **(d)** Rough dielectric

Figure 2.3. The BSDF captures the overall look of an object as different distributions yield different visual effects (*Teapot model* © Disney Pixar).

Definition 2.9 can be extended to handle transmittance, which is known as the *bidirectional transmittance distribution function* (BTDF) f_t. In this case, ω_i and ω_o lie on opposite sides of the surface plane (*i.e.*, $\omega_i \cdot \omega_o \leqslant 0$). Combining the BRDF and the BTDF (Figure 2.3) yields the *bidirectional scattering distribution function* (BSDF) $f_s : \mathcal{S}^2 \times \mathcal{S}^2 \subseteq \mathbf{R}^4 \to \mathbf{R}^3_+$, where the absolute value of the cosine term is used in (2.9) to handle both reflection and transmission. These function definitions implicitly ignore the volumetric effect on the subsurface, which implies that light is scattered at a single point on the surface. Further generalizations of these reflectance models, such as the *bidirectional subsurface scattering distribution function* (BSSRDF), can handle different entry and exit locations to capture internal transmittance within the volume underneath a surface. From now on, we shall use the BSDF for full generality.

2.4 The Rendering Equation

The rendering equation describes the equilibrium of the radiance at a surface interaction. Intuitively, it characterizes how a distribution of the scattered light varies according to the surface property. Introduced by [Kaj86], it has become the fundamental equation to solve in light transport simulation. In this section, we present various formulations of the rendering equation to

gain insights on the potential challenges of solving it numerically.

2.4.1. Spherical Formulation. We can formulate the radiance leaving a surface point $\mathbf{x}$ as the sum of *emitted radiance* $L_e(\mathbf{x}, \omega)$ and scattered radiance $L_s(\mathbf{x}, \omega)$:

$$L_o(\mathbf{x}, \omega) := L_e(\mathbf{x}, \omega) + L_s(\mathbf{x}, \omega). \tag{2.10}$$

The first term captures the self-emission of the surface at $\mathbf{x}$ which enables the modeling of various emission profiles such as the sun or neon lights; it is naturally zero for nonemissive surfaces. The second term can be directly obtained by integrating the BSDF over the the unit sphere of incident direction at $\mathbf{x}$, that is,

$$L_s(\mathbf{x}, \omega_o) := \int_{\mathcal{S}^2} f_s(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) |N(\mathbf{x}) \cdot \omega_i| \, \mathrm{d}\sigma(\omega_i). \tag{2.11}$$

The integrand is made of three terms: the incident radiance term $L_i(\mathbf{x}, \omega_i)$ determines the amount of light arriving at $\mathbf{x}$, the foreshortening (cosine) term $|N(\mathbf{x}) \cdot \omega_i| = |\cos\theta|$ weights this radiance depending on the incoming angle with respect to the surface normal, and the BSDF f_s captures how much of the light received is scattered towards the outgoing direction ω_o. Substituting (2.11) into (2.10) gives the *rendering equation* in its full solid angle form:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\mathcal{S}^2} f_s(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) |N(\mathbf{x}) \cdot \omega_i| \, \mathrm{d}\sigma(\omega_i). \tag{2.12}$$

Solving this equation is the crux of light transport algorithms.

2.4.2. Recursive Formulation. Converting Equation 2.12 into a recursive equation is straightforward. Let $r_{\mathcal{M}} : \mathcal{R} \to \mathcal{M}$ denote the *ray tracing* function that casts a ray from $\mathbf{x}$ in direction ω and returns the closest scene surface point along this direction. Formally, we can write

$$r_{\mathcal{M}}(\mathbf{x}, \omega) := \mathbf{x} + \underbrace{\inf\left\{t > 0 : \mathbf{x} + t\omega \in \mathcal{M}\right\}}_{d_{\mathcal{M}}(\mathbf{x}, \omega)} \cdot \omega, \tag{2.13}$$

where $d_{\mathcal{M}}(\mathbf{x}, \omega) := \infty$ if no intersection can be found. This definition allows us to relate L_i and L_o based on the preservation of radiance along unoccluded rays; in other words, we have that

$$L_i(\mathbf{x}, \omega) = L_o(r_{\mathcal{M}}(\mathbf{x}, \omega), -\omega). \tag{2.14}$$

This observation suggests that finding the incident radiance along a ray $(\mathbf{x}, \omega)$ merely amounts to determining the nearest surface visible from $\mathbf{x}$ and evaluating its outgoing radiance in the opposite direction. As a result, we obtain a recursive formulation of the rendering equation, known as the *light transport equation* (LTE):

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{S^2} f_s(\mathbf{x}, \omega_i, \omega_o) L_o(r_{\mathcal{M}}(\mathbf{x}, \omega_i), -\omega_i) \, \mathrm{d}\sigma^{\perp}(\omega_i). \tag{2.15}$$

Note the presence of the L_o term on both sides of the equation.

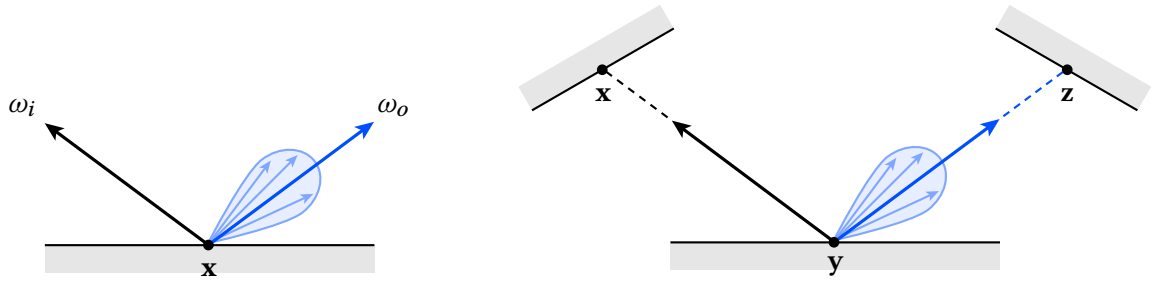
2.4.3. Surface Area Formulation. We can change the domain of integration to the scene surfaces and integrate with respect to the surface area measure A. This formulation of the rendering equation is quite useful as it allows us to integrate over all area light sources in the scene. Transforming the differential projected solid angle measure $\mathrm{d}\omega^{\perp}$ to a differential surface area measure $\mathrm{d}A$ introduces a Jacobian determinant to accommodate the change of coordinates. Given two points $\mathbf{x}, \mathbf{y} \in \mathcal{M}$ forming a unit direction $\omega = (\mathbf{x} - \mathbf{y})/\|\mathbf{x} - \mathbf{y}\| \equiv (\mathbf{x} \to \mathbf{y})$, the *geometry term* $G : \mathcal{M} \times \mathcal{M} \to \mathbf{R}^3_+$ is the product of the binary *visibility function* and this Jacobian:

$$G(\mathbf{x} \leftrightarrow \mathbf{y}) := V(\mathbf{x} \leftrightarrow \mathbf{y}) \left| \frac{\mathrm{d}\omega^{\perp}}{\mathrm{d}A} \right| = V(\mathbf{x} \leftrightarrow \mathbf{y}) \frac{|N(\mathbf{x}) \cdot \omega| \, |N(\mathbf{y}) \cdot \omega|}{\|\mathbf{x} - \mathbf{y}\|^2}, \tag{2.16}$$

where mutual visibility is defined as

$$V(\mathbf{x} \leftrightarrow \mathbf{y}) := \begin{cases} 1, & \text{if } \{t\mathbf{x} + (1-t)\mathbf{y} : t \in (0,1)\} \cap \mathcal{M} = \emptyset \\ 0, & \text{otherwise.} \end{cases} \tag{2.17}$$

Intuitively, the visibility function is an indicator function $V : \mathcal{M} \times \mathcal{M} \to \{0, 1\}$ that returns one when two points are mutually visible and zero when an object occludes the line segment between



(a) Directional form **(b)** Positional form

Figure 2.4. Directional and positional forms of the BSDF.

these points. Rewriting incident and exitant directions ω_i and ω_o in positional form $(\mathbf{x} \to \mathbf{y} \to \mathbf{z})$ with $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathcal{M}$ (Figure 2.4) yields the surface area form of the rendering equation:

$$L(\mathbf{y} \to \mathbf{z}) = L_e(\mathbf{y} \to \mathbf{z}) + \int_{\mathcal{M}} f_s(\mathbf{x} \to \mathbf{y} \to \mathbf{z}) L(\mathbf{x} \to \mathbf{y}) G(\mathbf{x} \leftrightarrow \mathbf{y}) \, \mathrm{d}A(\mathbf{y}). \tag{2.18}$$

Note that we have introduced a new notation for the quantities studied so far; double arrows are used to emphasize the symmetric nature of some functions while left-to-right arrows follow the propagation of light from points to points. For instance, the relationship between the two notations for the BSDF is

$$f_s(\mathbf{x} \to \mathbf{y} \to \mathbf{z}) \equiv f_s\left(\mathbf{y}, \frac{\mathbf{z} - \mathbf{y}}{\|\mathbf{z} - \mathbf{y}\|}, \frac{\mathbf{x} - \mathbf{y}}{\|\mathbf{x} - \mathbf{y}\|}\right). \tag{2.19}$$

2.5 Monte Carlo Methods

We review Monte Carlo (MC) methods for solving the light transport problem for surfaces. First introduced by Cook et al. [CPC84] to render distribution effects (*e.g.*, motion blur) and then applied to the rendering equation to simulate global illumination by Kajiya [Kaj86], Monte Carlo integration has revolutionized the field of physically-based rendering. Unlike other deterministic approaches, MC techniques do not suffer from the curse of dimensionality that incurs exponential computation costs as the number of dimensions increases. This makes Monte Carlo integration a tool of choice for solving the light transport problem.

2.5.1. Monte Carlo Integration. Suppose we want to integrate the function $f : \Omega \to \mathbf{R}$ over some measurable space $(\Omega, \mathcal{F}, \nu)$:

$$F = \int_\Omega f(x)\, \mathrm{d}\nu(x). \tag{2.20}$$

Monte Carlo integration relies on random sampling to estimate the value of F. Hence, let $X \in \Omega$ be a random variable with probability density function (pdf) $p_X(x)$, and further let $\psi : \Omega \to \mathbf{R}$ be any function of X. By definition,

$$\mathbf{E}[\psi(X)] = \int_\Omega \psi(x) p_X(x)\, \mathrm{d}\nu(x). \tag{2.21}$$

If we let $\psi := f(x)/p_X(x)$ then by construction, the expected value of $\psi(X)$ is F:

$$\begin{aligned} \mathbf{E}[\psi(X)] &= \int_\Omega \frac{f(x)}{p_X(x)} p_X(x)\, \mathrm{d}\nu(x) \\ &= \int_\Omega f(x)\, \mathrm{d}\nu(x) \\ &= F. \end{aligned} \tag{2.22}$$

Therefore, it suffices to design an estimator for $\mathbf{E}[\psi(X)]$ to numerically approximate F. This can be achieved by choosing N independent and identically distributed random variates $\{X_i\}_{i=1}^N \sim p_X$ and then computing the estimate

$$\langle F_N \rangle := \frac{1}{N} \sum_{i=1}^N \psi(X_i) = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)} \tag{2.23}$$

Here, $\langle F_N \rangle$ denotes the *Monte Carlo estimator* for F and is itself a random variable. This estimator gives the correct result on average and converges at a rate of $O(N^{-1/2})$. In other words, to decrease the expected estimation error by a factor of two, we must use four times as many samples, which is a relatively poor convergence rate. This result holds even if the variance is infinite, provided that $\mathbf{E}[\langle F_N \rangle]$ exists. This is guaranted by the strong law of large numbers. This also holds regardless of the dimension of Ω, unlike other numerical integration techniques such as

Gaussian quadrature rules. This property is particularly interesting for photorealistic rendering as paths can have arbitrarily long lengths.

2.5.2. Importance Sampling. It is clear that the probability density p_X inherently controls the variance of $\langle F_N \rangle$; a pdf approximately proportional to the integrand is thus necessary for the Monte Carlo estimator to be efficient. To see this, suppose that we have a perfect sampling density $p_X^*(x) = \alpha f(x)$ for some $a > 0$. Then,

$$1 = \int_\Omega p_X^*(x)\,\mathrm{d}\nu(x) = \alpha \int_\Omega f(x)\,\mathrm{d}\nu(x) \quad \Longrightarrow \quad \alpha^{-1} = \int_\Omega f(x)\,\mathrm{d}\nu(x). \tag{2.24}$$

This implies that the Monte Carlo estimator has zero variance since

$$\langle F_N \rangle = \frac{1}{N}\sum_{i=1}^{N} \frac{f(X_i)}{p_X^*(X_i)} = \frac{1}{N}\sum_{i=1}^{N} \alpha^{-1} = \int_\Omega f(x)\,\mathrm{d}\nu(x) \tag{2.25}$$

for all sample points X. In other words, a single sample is sufficient to resolve the integral. Unsurprisingly, this is a *chicken and egg* situation since it assumes we already know the value of F in order to compute the normalization constant α! This observation suggests that a probability density p_X chosen to resemble f should reduce the variance of $\langle F_N \rangle$. This technique is known as *importance sampling* and is used to accelerate the convergence of MC estimators. It is possible to importance sample different parts of the integrand. For instance, if $f = gh$, we could sample both g and h separately. Ideally, sampling the product f would be the most effective method to controlling the variance of the estimator. This is difficult to do in practice, and as such, *local importance sampling* of individual terms is usually preferred.

2.6 Deep Feedforward Networks

We now move onto reviewing the basics of feedforward neural networks and then introduce some concepts for designing efficient learning algorithms. We start by describing the classical components of neural nets and then explain how these models can be trained to infer on unseen data.

2.6.1. Formal Definition. Deep feedforward networks, also called *feedforward neural networks*, are the cornerstones of deep learning. The main goal of this family of models is to approximate some function Φ^*. In the context of classification, $\Phi^*(\mathbf{x}) = y$ maps an input $\mathbf{x} \in \mathcal{X}$ to a class or category $y \in \{1, \ldots, n\} = \mathcal{Y}$. For binary classification ($n = 2$), there are only two possible labels (*e.g.*, cats and dogs). More generally, a neural network defines a mapping $\Phi(\mathbf{x}; \boldsymbol{\theta}) = \hat{y}$ and learns the parameters $\boldsymbol{\theta}$ that result in the best function approximation to Φ^*. The parameters of the network are optimized with respect to a pre-specified cost function on a finite dataset to obtain the closest possible approximation of Φ^*. More generally, feedforward neural networks can be seen as function approximation machines that are carefully designed to achieve statistical generalization.

There are many components to a feedforward neural net. At a very high level, a deep neural network implements a directed acyclic graph that describes a composition of functions, for instance $\Phi = \Phi^{(d)} \circ \cdots \circ \Phi^{(2)} \circ \Phi^{(1)}$. Each function is associated to a *layer* and the overall length of this chain gives the *depth* of the network. The first layer of this structure is called *input layer* and the last layer is called the *output layer*. We train these networks by using a collection of noisy, approximate example pairs $(\mathbf{x}, \Phi^*(\mathbf{x})) \equiv (\mathbf{x}, y)$. These training examples inform the network of possible realizations of Φ^* and are used to learn Φ directly. The underlying learning algorithm must determine how to use the internal (hidden) layers of the network to produce its best representation of Φ^*. Each hidden layer $\mathbf{h}^{(i)}$ is vector-valued and its dimensionality (also known as its *width*) can vary from layer to layer. Each element of this vector is called a *neuron* or *unit* and its role resembles that of an actual brain neuron as it reacts to an input signal, by providing an activation response as output. The *activation function* for these neurons introduces nonlinearity in the approximator which make neural nets particularly good candidates for estimating arbitrary functions.

Each layer $\mathbf{h}^{(i)}$ is finally connected to the next by a real-valued weight matrix $\mathbf{w}^{(i)}$ of size $N_{i-1} \times N_i$, where N_i is the width of layer i. Propagating an input through the network results in a sequence of matrix multiplications followed by pointwise application of a nonlinear function $g^{(i)}$ at each neuron. A *bias* term $\mathbf{b}^{(i)}$ is also commonly added to shift the activation function while providing each neuron with a trainable constant value. Combining all components, we can

express the first layer of the neural network as

$$\mathbf{a}^{(1)} = \mathbf{w}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)}, \tag{2.26}$$

$$\mathbf{h}^{(1)} = g^{(1)}\big(\mathbf{a}^{(1)}\big), \tag{2.27}$$

where $\mathbf{x}$ is the input of the network viewed as a column vector in $\mathbf{R}^M$, and $\mathbf{a}^{(i)}$ denotes the *activation* of layer i. More generally, we can write the i-th layer recursively as

$$\mathbf{h}^{(i)} = g^{(i)}\big(\mathbf{w}^{(i)\top}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}\big), \qquad 2 \leqslant i \leqslant d, \tag{2.28}$$

where d is the depth of the network. We shall denote by $\mathbf{W}$ and $\mathbf{B}$ the set of all weights and biases of the model (viewed as flattened row vectors), respectively. The architecture of a simple neural network is depicted in Figure 2.5. We next discuss how this model can be trained on empirical data to approximate an arbitrary function Φ^*.

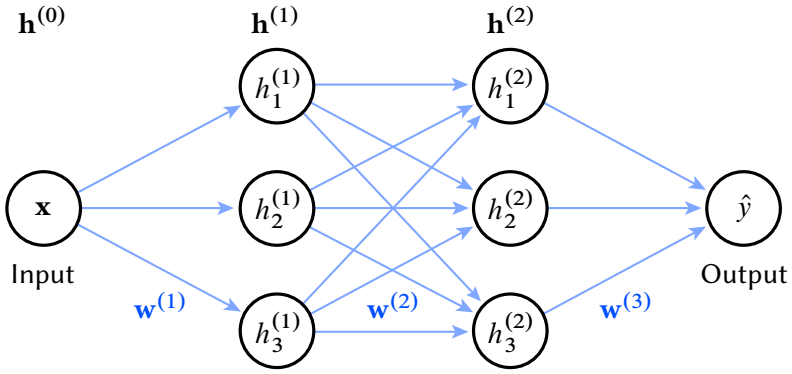


Figure 2.5. Example of a simple neural network with two hidden layers (also called a *multilayer perceptron*) with 3 neurons each. All components of the input $\mathbf{x} \in \mathbf{R}^M$ are connected to each neuron $h_j^{(1)}$ of the first layer $\mathbf{h}^{(1)}$ via the weight matrix $\mathbf{w}^{(1)} \in \mathbf{R}^{M \times N_1}$ (blue arrows). A nonlinearity $g^{(i)}$ is then applied pointwise to each entry before adding a bias term (here omitted from the diagram). This computation is passed to the next layer as input, propagating $\mathbf{x}$ through the network until it reaches the last layer, where $\Phi(\mathbf{x}) = \hat{y}$.

2.6.2. Gradient-Based Learning. Training neural networks is an optimization problem where we update the parameters $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{B}\}$ of the model based on some cost function we wish to minimize. When we use a feedforward neural net, we propagate an input $\mathbf{x}$ through the network to

produce an output $\hat{y}$. This information flows from the first to the last layer. During training, this *forward propagation* continues onward to produce a scalar cost $\mathcal{L}(\hat{y}, y)$. This *loss function* measures how close the network output $\hat{y}$ is to the real target y with the current parameters. The *backward propagation* algorithm [RHW86] (also known as *backprop*) allows the information from this cost to flow backward in the network. By recursively applying the chain rule on the computational graph characterized by the network, we can compute the gradients on the activations $\mathbf{a}^{(i)}$ for each layer i. These indicate how each layer's response should change in order to reduce the global error of the network. From there, the gradient of $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$ can be computed and used as part of a stochastic gradient update.

In practice, we use a finite training set $(\mathcal{X}, \mathcal{Y})$ consisting of example/target pairs $(\mathbf{x}, y)$ to learn a model. The goal of a deep neural net is to minimize the cost function over this training set using the information provided by the backpropagated gradients. We can think of the loss function $\mathcal{J}(\boldsymbol{\theta})$ as an average over this training set, that is,

$$\mathcal{J}(\boldsymbol{\theta}) := \mathbf{E}_{(\mathbf{x},y)\sim\hat{p}_d}\, \mathcal{L}(\Phi(\mathbf{x};\boldsymbol{\theta}), y) = \frac{1}{K}\sum_{i=1}^{K} \mathcal{L}\big(\Phi(\mathbf{x}^{(i)};\boldsymbol{\theta}), y^{(i)}\big) \tag{2.29}$$

where $\hat{p}_d$ is the empirical distribution of the data and $|\mathcal{X}| = K$. Intuitively, learning amounts to searching for the best parameters for the model:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}). \tag{2.30}$$

As $m \to \infty$, the computational cost associated with the expectation (2.29) increases with $O(n)$ for a single gradient update. Gradient descent in general is often too slow and unreliable to be useful. The insight of *stochastic gradient descent* (SGD) is that the gradient is in fact an expectation which can be estimated by a smaller collection of examples. By sampling a *minibatch* $\mathcal{B} = \big(\mathbf{x}^{(i)}, y^{(i)}\big)_{i=1}^{m} \subset (\mathcal{X}, \mathcal{Y})$ for some fixed $m \ll K$ uniformly, we can estimate the gradient as

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} \mathcal{L}\big(\Phi(\mathbf{x}^{(i)};\boldsymbol{\theta}), y^{(i)}\big). \tag{2.31}$$

Empirically, m is commonly chosen to be a moderately low power of two (*e.g.*, $m = 128$) to reduce interface overhead with GPUs when training.[2]

Given Equation (2.31), we can then update the parameters $\boldsymbol{\theta}$ by taking a step in the negative direction of this gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}), \tag{2.32}$$

where η is called the *learning rate*. This procedure is typically repeated $|\mathcal{X}|/|\mathcal{B}| = K/m$ times to complete a full *epoch* until some user-defined stopping criterion is met (*e.g.*, $\mathcal{J}(\boldsymbol{\theta}) \leqslant \epsilon$ for some $\epsilon > 0$). SGD does not guarantee to converge to a local minimum but, in practice, it often finds low values of $\mathcal{J}$ in a reasonable amount of time, hence its predominance in deep learning. We shall refer to all parameters whose value is set *before* the learning process begins as *hyperparameters*. These include the width of each layer, the depth of the network, the learning rate, and so on.

2.6.3. Deep Learning Best Practices. Hitherto, we have seen that training a neural network merely amounts to iteratively updating the weights and biases using gradient information for the loss function. In its simplest form, a deep learning algorithm using a neural network is just a combination of a training dataset, a network architecture, a cost function, and an optimization procedure. Carefully choosing each of these components is almost an art form. In fact, there is a whole research field devoted to tailoring this recipe to certain types of problems in order to create robust solutions that generalize well to unseen data. Here we discuss a few important concepts for stable training in the context of classification.

NETWORK ARCHITECTURE. Feedforward neural nets with hidden layers provide a universal approximation framework. The universal approximation theorem [HSW89, Cyb89] states that a neural net with a linear output layer and at least one hidden layer with any "squashing" activation function (to be discussed soon) can approximate any Borel measurable function arbitrarily well. Put simply, this theorem shows that there always exists a network large enough to achieve

[2]Recent work [ML18] has shown that training with lower values such as $m = 16$ can be more reliable and stable in specific cases. For multilayer perceptrons, training is already very stable, so we use a larger batch size to reduce training time.

the desired degree of accuracy, but it does not provide a bound on its width. This is problematic as the layer may be infeasibly large and thus the algorithm may fail to learn the optimal parameters and generalize correctly. Consequently, determining the number of layers and neurons per layer for a supervised learning task is still an open problem. Empirically, a deeper network will perform better than a wider one on an arbitrary task [BLPL06]. Great depth can help reducing the amount of generalization error and almost always result in easier training due to fewer units.

In general, the layers of a neural network need not be connected in a chain. Many techniques have been developed to fit different tasks. Standard feedforward deep nets can generalize to other types of architectures, motivated by the learning problem to solve. *Convolutional neural networks* (ConvNets) are a specialized family of nets for processing data that has a known, grid-like topology such as color images, while *recurrent neural networks* add feedback connections at each layer to handle sequential data such as text. We shall focus on the classical multilayer perceptron model from now on.

Activation Functions. When we defined neural networks, we have introduced the concept of a hidden layer, but this definition requires us to choose the activation functions that will be used to compute the neurons values $h_j^{(i)}$. In modern neural networks, the most commonly used activation function is the *rectifying linear unit* (also known as ReLU) [JKRL09] defined as $\mathrm{ReLU}(z) := \max(0, z)$. This function is nonlinear but because it is also nearly linear, it preserves several of the properties that make linear models easy to optimize with gradient descent techniques.[3] This definition is useful because the derivatives through a rectifying linear unit always remain large whenever the unit is active. Therefore, the gradients are large and also consistent across the network, which turns out to be highly valuable for learning. There are other generalizations of the ReLU, including the *leaky rectifying linear unit* which ensures that gradients can flow through everywhere and not only when the neuron is active. The activation of the last layer in a network is usually chosen to suit the task. For instance, in a classification problem, it is common to apply the softmax operator to "squash" a vector in $\mathbf{R}^k$ to a vector in $(0, 1)^k$ to represent a categorical distribution. When $k = 2$, the softmax is equivalent to the logistic sigmoid

[3] The non-differentiability of the ReLU at $z = 0$ in SGD is handled by treating the function as piecewise linear, essentially setting its gradient to zero at this point.

$s(z) := 1/(1 + \exp(-z))$.[4]

REGULARIZATION. The main goal of a neural network classifier is to be able to infer the labels from unseen examples. Multiple strategies from the machine learning literature are designed to reduce the generalization error of a learning algorithm. These methods are collectively known as *regularization*. In the context of deep learning, regularizers are used to decrease variance without overly increasing the bias of the estimator. Limiting the capacity of the model by adding a parameter penalty term $\Omega(\boldsymbol{\theta})$ to the objective function $\mathcal{J}$ is one way of regularizing the model, *i.e.*, $\widetilde{\mathcal{J}}(\boldsymbol{\theta}) = \mathcal{J}(\boldsymbol{\theta}) + \alpha\Omega(\boldsymbol{\theta})$ for some $\alpha > 0$. *Tikhonov regularization* (also known as L^2 *regularization* or *weight decay*) sets $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{W}\|_2^2$. It can be shown that weight decay is equivalent to early stopping in the case of linear model with a quadratic error function. Other regularization techniques include dropout [SHK+14], batch normalization [IS15], and bagging [Bre96].

ADAPTIVE LEARNING RATES. There exists several variants of stochastic gradient descent that adapt the learning rate during training. This idea is based on the observation that the partial derivative of the loss with respect to the parameters can vary and thus an efficient optimization algorithm should adapt to these changes. Methods such as AdaGrad [DHS11] and RMSProp [Hin12] keep a history of accumulated gradients to inform the parameter updates, while Adam [KB14] use moments of the gradients to indirectly incorporate momentum in the updates. The latter is generally considered the most robust optimizer with respect to the choice of hyperparameters.

A comprehensive and complete overview of deep learning techniques for classification is well beyond the scope of this thesis. For a detail discussion of deep learning, we refer the reader to Goodfellow *et al.* [GBC16]. A more concise introduction to the field is also given by Lecun *et al.* [LBH15]. From now on, we shall assume basic notions and concepts related to the efficient training of deep neural networks, occasionally referring to research literature to justify certain design choices.

[4] In fact, the softmax is a generalization of the logistic function in higher dimensions.

Chapter 3

Soft Shadows

Soft shadows from area light sources are important in physically-based rendering since they provide visual depth cues about the lighting settings. It is clear that shadows significantly add to the believability of a scene; without them, a scene would look flat and overly synthetic. The main challenge in computing these shadowed regions comes from evaluating the spherical binary visibility function $V(\cdot, \cdot)$ at every shading point: given an intersection point $\mathbf{x}$, can we make an unobstructed connection to a point $\mathbf{y}$ on a light? The numerical solutions to the light transport problem discussed thus far all assume the existence of a ray tracing function $r_{\mathcal{M}}$ that intersects a ray $\mathbf{r} = (\mathbf{x}, \omega) \in \mathcal{R}$ with the scene geometry to retrieve a new point $\mathbf{y} \in \mathcal{M} \cup \{\infty\}$. Up until now, we have merely abstracted the notion of ray tracing and completely disregarded how such a function is implemented in a modern rendering framework. The notion of mutual visibility between two points is evident and intuitive, but implementing this query in a renderer that supports discretized scenes comprising several objects is nontrivial.

For offline rendering—which aims to render still images in a physically-accurate manner—the mathematical framework for visibility directly encapsulates what it means for two points to be mutually visible. Tracing a ray at every shading point to evaluate the light contribution seems quite natural, provided an almost unlimited computational budget. Interactive techniques, on the other hand, do not have this luxury and can only use a fraction of the cost to compute an image frame. Video games, for instance, require to render multiple frames per second (*e.g.*, 30 fps on consoles, 60 fps on computer platforms) to simulate a smooth experience where characters

can move and affect their environment. *Real-time rendering* is a completely different field in itself as it can only *approximate* the physics of the light to allow for other interactions to occur simultaneously. In real-time graphics, people strive for realism but are limited by important performance considerations. These two goals—realism and performance—are extremely difficult to combine in general. This is particularly true in the context of shadows, as we shall see.

We introduce the terminology for shadow casting and explain the distinction between *hard* shadows and *soft* shadows. We also highlight the difficulties encountered when generating realistic shadows in both offline and real-time settings. Next, we review the most preferred approaches to generating shadows in the context of direct illumination only. In particular, we present various techniques to either perfectly simulate soft shadows or approximate them efficiently. Finally, we discuss recent advances in real-time ray tracing and what this means for soft shadows synthesis.

3.1 Types of Shadows

Shadows are produced when an *occluder* is placed between a light source and a *receiver*. For infinitely small light source models like point lights, there is only one possible segment that connects the shading point with the emitter. Consequently, the shadows created by these so-called δ-lights are characterized as *hard* since their silhouettes are crisply defined and sharp. Area lights, on the other hand, have finite surface areas and, as such, provide uncountably many ways of connecting the shading point to the visible surface of the light. This results in *soft shadows* which are less harsh and thus gradated towards the edges. Each shadow can have a fully shadowed region, called the *umbra*, and partially shadowed regions, called the *penumbra*. Point lights only have umbra regions, whereas area lights give rise to penumbra for a more realistic look. This is illustrated in Figure 3.1.

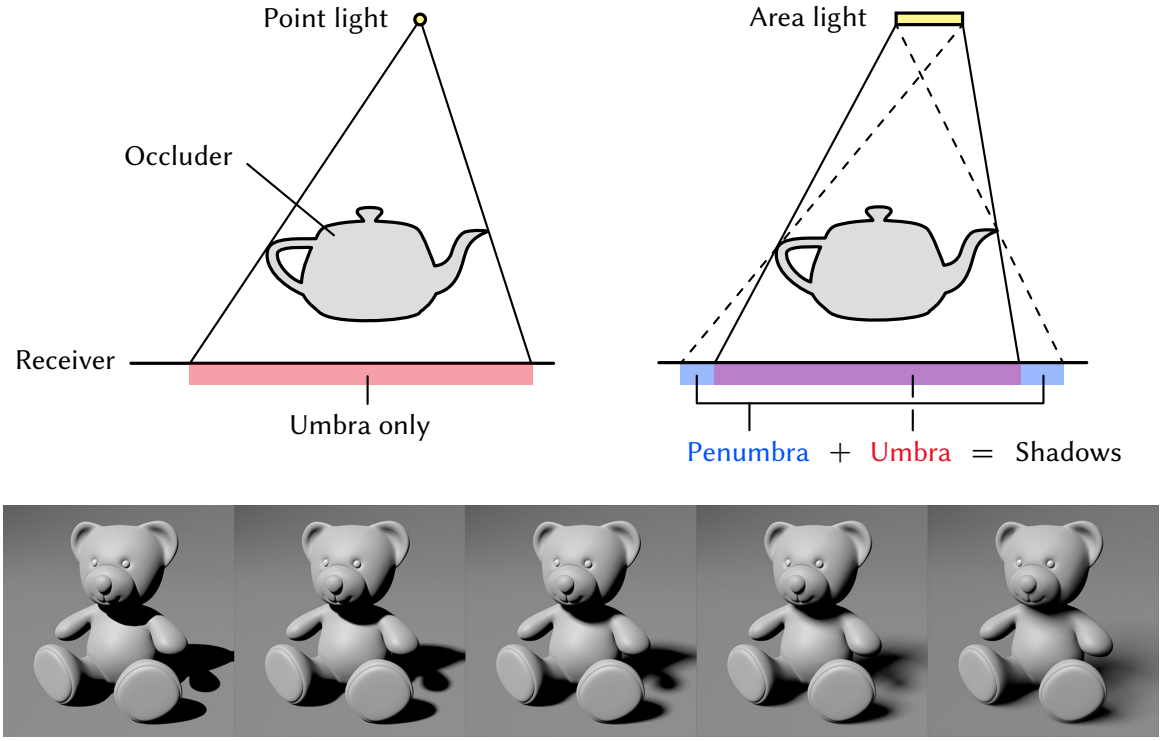


Figure 3.1. *Top left:* a teapot occludes the incoming light from an infinitely small point source, thus casting umbra on the receiving ground to create hard shadows. *Top right:* the teapot casts both umbra and penumbra to create soft shadows. *Bottom:* The distinction between hard and soft shadows is mainly noticeable at the silhouette of the shadows casted by the teddy bear. From left to right, the surface area of the light is increased in size while keeping its radiance-to-area ratio constant. The umbra becomes less and less perceivable (*Image source:* Tim May).

It is important to observe that *true* soft shadows are not just hard shadows blurred out by a low-pass spatial filter. Moreover, the umbra region of a soft shadow is not equivalent to a hard shadow generated by a point light source (assuming we fix the positions of the lights). Indeed, the umbra region of a soft shadow shrinks in size as we increase the surface area of an area emitter. For very large area lights that are placed far enough from the receiver, the umbra can be almost unperceivable or completely vanish. Soft shadows are usually preferable to avoid any visual ambiguities, such as misreading a shadow for an actual geometric feature (*e.g.*, a crease in a surface) [AMHH+18]. The main advantage of hard shadows produced by point lights or infinite directional lights is that they are faster to compute in general. These δ-light models are nonphysically-realizable as they are not attached to physical surface. This implies that, to

compute the radiance contribution, there is always a *unique* direction to pick between the point to be shaded and the light. This, in turn, significantly simplifies the visibility evaluation. Soft shadows correspond to a most accurate model of shadowing but require a light sampling step. In any case, soft shadows should be sought as much as often as possible to synthesize realistic images.

3.2 Ray Intersection Acceleration

We now turn our attention to offline techniques for generating soft shadows. Our main interests are *secondary rays*, those rays that evaluate the light contribution at a shading point $\mathbf{x}$. These rays will be referred to as *shadow rays* as they effectively determine if $\mathbf{x}$ either directly lies in shadow or not (Figure 3.2). Secondary rays are different from *primary rays* which are traced directly from the eye.

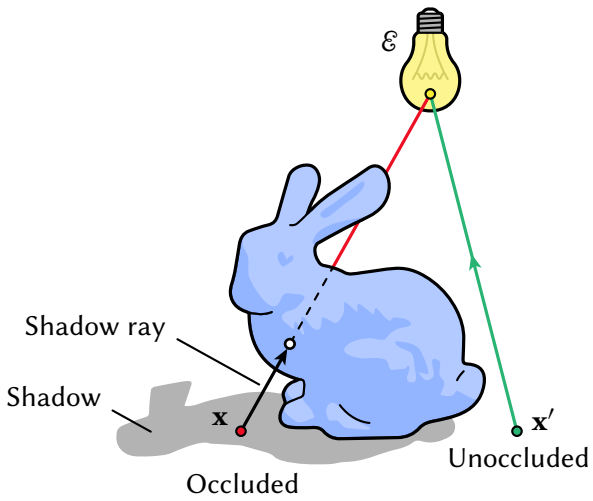


Figure 3.2. To determine if a point lies in shadow, we simply trace a ray to the light source. The shadow ray with origin $\mathbf{x}$ intersects the bunny before reaching the light $\mathcal{E}$, so the radiance contribution is zero. Conversely, the shadow ray with origin $\mathbf{x}'$ has a clear line of sight, yielding positive incoming radiance at that point.

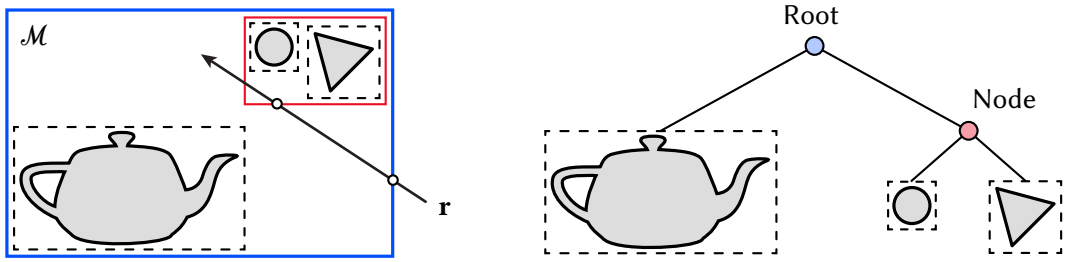
In computer graphics, scenes are typically represented as a union of polygon meshes indexed by triangles. Each mesh or object in a scene is entirely determined by the positions of its polygons along with their face and vertex normals. Determining the intersection of a line in $\mathbf{R}^3$ with an

analytic shape (*e.g.*, disk, plane, sphere) has closed-form solutions and can be evaluated efficiently in a software. Shapes made of millions of triangles, however, require a special type of data structure to search for triangle intersection. Such a spatial tree structure is used to trace primary and secondary rays in a scene to evaluate radiance contributions. An efficient implementation of such a data structure is crucial to a rendering engine as it allows for more paths to be computed for the same amount of time which, in turn, yields an estimate of the rendering equation with lower variance.

There are many types of such acceleration structures, the most prevalent being *bounding volume hierarchies* (BVH). Bounding volumes are used to enclose a collection of polygonal meshes. The idea is that these volumes should be *much* simpler geometry-wise than the objects they contain. BVHs are based on the premise that testing for intersection against trivial shapes—such as cubes or spheres—has a much lower computational cost than testing on complicated meshes with million of triangles. By partioning the primitives of a scene into a hierarchy of disjoint sets, we can test for ray intersection against these simpler shapes and quickly discard nonintersecting rays: if a ray does not hit the enclosing parent node, it will never hit its children and therefore can be skipped.

More formally, a BVH stores the primitives as the leaves of a tree and stores a bounding box of the primitives in the nodes beneath it. Determining the ray intersection point, if any, merely boils down to traversing the tree from the root until a leaf is found. If the ray does not hit the node's bounds, the subtree beneath that node can be skipped and the BVH can simply return nil (Figure 3.3). Hence, a bounding volume essentially acts as a proxy for the bounds of the enclosing objects to speed up the intersection computations.

The top-down construction of the scene's BVH is done before any rendering takes place. Building an efficient tree structure amounts to creating a partitioning of the scene geometry that does not cause too many overlaps. If the partition is computed carefully, more nodes of the tree can be visited by rays (on average), which can lead to unneccessary computations at render time. Most of the best algorithms for building ray acceleration data structures are based on the *surface area heuristic* (SAH), a strategy that essentially provides a cost model to determine which spatial subdivision scheme should occur next. SAH uses geometric probabilities to design a cost



(a) Scene representation **(b)** Tree representation

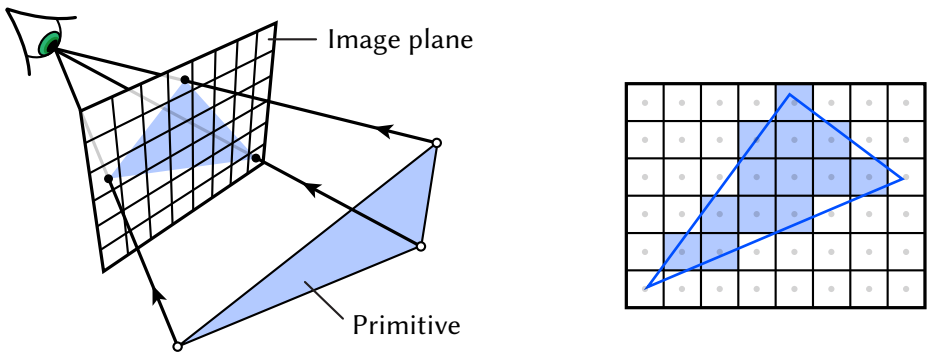
Figure 3.3. At construction, each object in the scene $\mathcal{M}$ is enclosed in its bounding volume in a recursive manner. When a ray **r** is traced, we first check that its trajectory is within the bounds of the scene (blue). As it is the case, we check against internal nodes (red box and teapot). No intersections could be found, so the algorithm returns nil. We only had to check a maximum of three ray-box intersections to discard the ray **r**.

function for splitting the space to minimize spatial overlapping and thus the expected intersection time. Before usage, the resulting binary tree is typically converted to a more compact linear representation to improve cache, memory, and overall system performance. Traversing this tree is logarithmic in the number of objects.

Intersection acceleration design is an entire subfield of computer graphics and a thorough overview of acceleration structures for rendering is out of the scope of this thesis. K-d trees are sometimes used as alternatives to scene subdivision but, albeit nonoverlapping and faster to traverse on CPUs, their performance is substantially affected on larger complex scenes. The main takeaway here is that, even if BVHs and k-d trees aim at accelerating ray-primitive intersection, they are typically considered too slow for real-time applications. Improving the quality of BVHs after construction is an active field of research but the computational budget available for interactive techniques is simply too low to allow its usage [PJH17]. Ray traversal on the GPU has made significant progress in the past few years [AL09, YKL17] but it is still too expensive to be performed on most industrial graphics cards.

3.3 Previous Work on Interactive Shadows

In real-time rendering, soft shadows can only be estimated from hypothetical point light sources. Since we cannot trace rays to perform Monte Carlo sampling, other techniques had to be developed to approximate shadows casted by area light sources. *Rasterization* is the method of choice for modern real-time applications. The idea of rasterization is to geometrically project primitives from the scene onto the image plane and then process the pixels it affects. In essence, it converts the triangles of the polygon models into pixels on the screen (Figure 3.4).



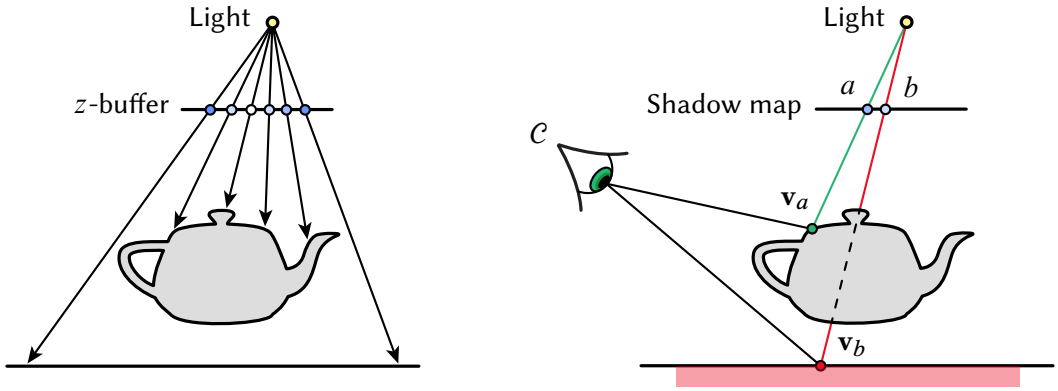
(a) Project to screen space **(b)** Find pixel coverage and interpolate

Figure 3.4. In rasterization, we first project the vertices making up triangles onto the screen, then we determine the pixels they cover. Linear interpolation of varying variables and depths is then performed for all covered pixels.

Several methods were developed in this particular framework to evaluate the occlusion profile of arbitrary meshes. In what follows we discuss the most relevant approaches for generating soft shadows at interactive framerates.

3.3.1. Classic Shadow Mapping. Considered one of the oldest technique for simulating shadows, *shadow maps* use a z-buffer to efficiently determine shadow regions casted from static light sources. Introduced by Williams in 1978 [Wil78], shadow mapping renders the scene a first time from the position of the light and stores the output into a z-buffer: whatever the light "sees" is illuminated. This shadow depth buffer is then used to determine if a point lies in shadow or not during a second pass, this time rendered with respect to an arbitrary view point. As each primit-

ive is drawn, we compare with the computed shadow map at each pixel. If a point to be rendered is farther away from the emitter than the corresponding value in the z-buffer, it is occluded and thus lies in shadow, otherwise it is visible. This is illustrated in Figure 3.5.



(a) Computing the shadow map **(b)** Determining shadows at run-time

Figure 3.5. In shadow mapping, the scene is first rendered from the point of view of the light and z-depths are calculated and stored in a buffer. To determine if a vertex $\mathbf{v}$ is in shadow, we compare its z-value with the depth buffer's corresponding texel. Here, the distance from $\mathbf{v}_a$ to the light is less than the value in texel a, so $\mathbf{v}_a$ is visible. On the other hand, $\mathbf{v}_b$ is farther away from the light than the depth stored in texel b and therefore lies in shadow (red).

In practice, shadow mapping is implemented via texture map lookups. The cost of building this texture is $O(n)$ for n primitives to be rendered, and accessing each texel is constant. The main benefit of shadow maps is that they can be computed once per light beforehand and they can then reused across frames, assuming the light sources are static.

The shadow mapping model described so far is ambiguous because the direction of lighting is unclear for different light profiles. For infinite directional light sources (*e.g.*, the sun) or distant spotlights (*e.g.*, a lighthouse), we can suppose that the emitter is not part of the scene per se, which heavily constrains the possible directions of illumination. The light's view frustum is set to encompass the collection of all possible occluders in the camera's view frustum and some optimizations can be made to select these occluding candidates. If the light is within the boundaries of the scene then a popular method is to use a cube environment map to project and capture all directions. This approach is often referred to as *omnidirectional shadow mapping*.

The pixel resolution of the shadow buffers and their numerical precision needs to be carefully chosen, otherwise shimmering edges, perspective aliasing, and other precision issues can occur. *Shadow acne* is a typical problem that is caused when a triangle is incorrectly consider to shadow itself. Adding a small bias when comparing depths is one possible solution, but this value usually needs to be angle-dependent to capture subtleties at contact points. A bias that is too high will result in *Peter Panning*, a light leaking artifact that makes objects with missing shadows appear to be detached and to float above the surface, leaving an unwanted gap inbetween (Figure 3.6). There exists many variants of shadow depth maps that addressed these issues (*e.g.*, *cascaded shadow maps* (CSM)), but they all provide an approximate solution for *hard* shadows. We now look at techniques that can be used to soften shadow edges for a smoother look at shadow silhouettes.

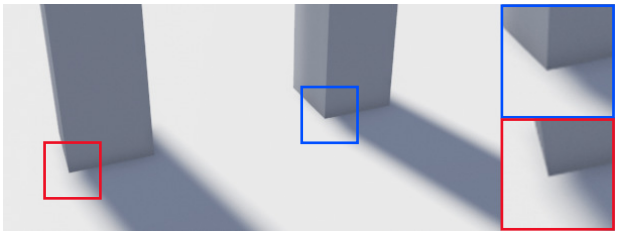


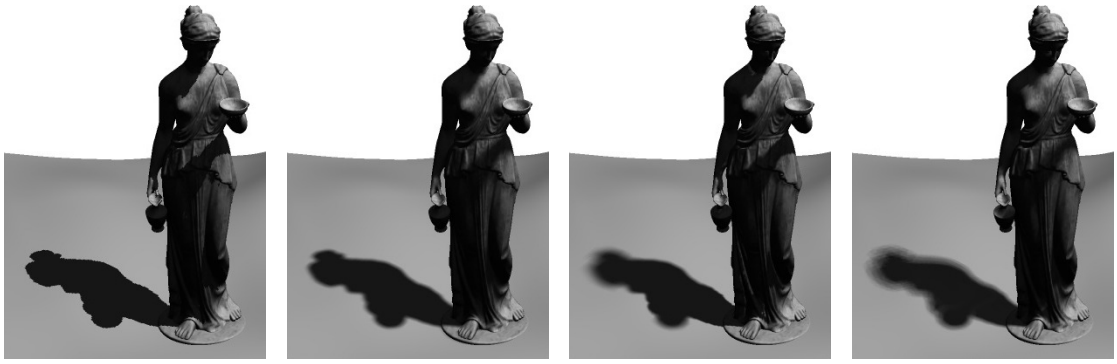
Figure 3.6. Peter Panning (or light leaking) occurs at contact points and can negatively impact the realism of soft shadows. *Image source:* Unreal Engine 4.

3.3.2. Percentage-Closer Filtering. Shadow maps can be extended to approximate penumbra to generate pseudo-soft shadows. The main goal of this method is to blur out the shadow edges that may look blocky and unnatural. This aliasing artifact occurs when a large number of screen pixels are covered by a single texel in the shadow map. To avoid this problem, *percentage-closer filtering* (PCF) [RSC87] retrieves multiple samples from a shadow map and blends the results. More precisely, PCF attempts to approximate the proportion of samples in some fixed regions around the receiver shading point that are visible from the light. Many shadow map comparisons are made per pixel and then filtered via some screen-space kernel to create artificial soft shadows. Unfortunately, self-shadowing problems and light leaking artifacts can still occur with this technique. Perhaps the main issue with percentage-closer filtering is that shadows will appear uniformly soft. This is because the width of the area to be sampled is typically held constant, which results in penumbra regions with equal width. While this is an important improvement

over hard blocky edges, penumbra usually become softer as it moves away from umbra. This is particularly apparent when occluders are in contact with the receiver.

Percentage-closer soft shadows (PCSS) [Fer05] resolves this problem by increasing the width of the surface area to be sampled. In particular, this area grows as the average occluder gets farther from the receiver and closer to the emitter. PCSS essentially works by finding an average depth of nearby occluders to rescale, assuming the average blocker is a reasonable estimate of the size of the penumbra. While this generally results in perceptually-accurate soft shadows, this strong assumption can be violated in simple scenarios where two close and distant objects occlude the same surface at a pixel. GPU-based techniques such as ones using *backprojection* are viable options but their high per-pixel cost is not suitable for interactive rendering.

3.3.3. Other Shadowing Techniques. There are several other techniques to generate soft shadows in real-time. For instance, *variance shadow maps* (VSM) [DL06] stores the depth in one buffer and the depth squared in another, which allows for statistical prefiltering of the shadow map textures. VSM have significant performance benefits over PCF methods as the shadow depths can be pre-filtered with a separable blur kernel. As a result, light bleeding when there is high depth complexity and/or variance is handled remarkably well by VSM. Other approachs based on *irregular z-buffers* (IZB) permit sampling of the scene at arbitrary points on the image plane. This technique allows for one or more receiver locations to be stored at each map texel. Cascaded shadow-map approaches can also be used in conjunction with filtered maps. This hybrid solution encapsulates the idea that we generally need shadows to be more accurate near the object rather than farther. Sharp shadows from IZB blended with soft from PCSS can generate compelling penumbra regions, hence their popularity in modern video games. Some of the techniques discussed above are illustrated in Figure 3.7. A complete overview of shadow mapping techniques can be found in [ESAW11].



(a) Shadow maps **(b)** PCF **(c)** PCSS **(d)** VSM

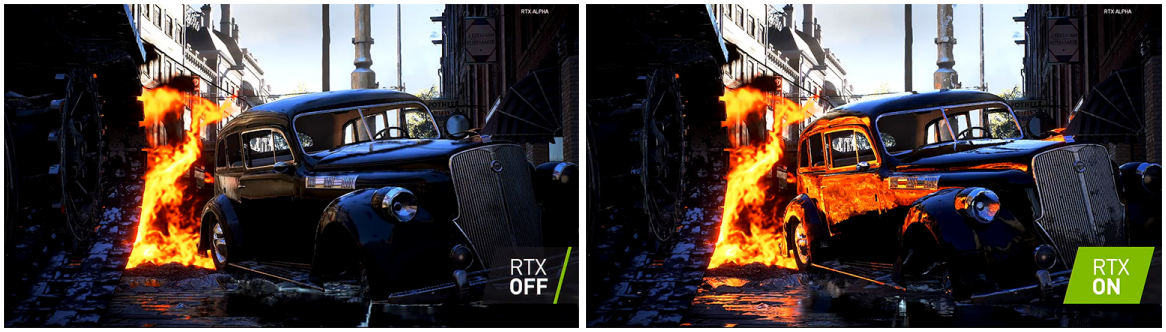
Figure 3.7. Approximate soft shadows methods. *Image source:* [Mik08].

3.4 Real-Time Ray Tracing

All interactive techniques for generating soft shadows are approximate solutions to the true visibility problem. Querying a BVH in real-time is generally considered too costly to be worthwhile, but this is subject to change in the near future. While rasterization has ruled the video game industry ever since its invention, its limitations are quite severe. The so-called "AAA" video game titles can be visually stunning and immersive, but underneath the hood, many corners are being cut to allow for dynamic, interactive environments. It is clear that ray tracing is in every way superior to rasterization to resolve complex lighting phenomena, but the heavy machinery needed to achieve this is simply not suited for today's GPUs. Or is it?

Recently, hardware-based solutions for ray tracing has started to emerge with impressive results [YKL17, Hil18]. As pure ray tracing is still impossible to perform at interactive framerates, adopting a hybrid approach where rasterization is used for primary rays and ray tracing is used for secondary ones seems like a powerful strategy. Combined with recent advances in Monte Carlo denoising [ZJL+15], real-time ray tracing has the potential to generate both realistic true reflections and shadows, something the field of computer graphics has sought for decades. This approach has proved to be very successful thus far. For instance, [HHM18] obtains close-to-exact soft shadows by first stochastically tracing visibility rays and then denoising the result. With improved denoising algorithms and GPU architectures optimized for tracing rays, it is undeniable

that this hybridity will provide significant performance boosts to real-time applications. Recent games such as EA's *Battlefield 5* (2018) and Eidos' *Shadow of the Tomb Raider* (2018) have already started to embrace this hybrid rendering approach with impressive results (Figure 3.8).



(a) Rasterized **(b)** Ray-traced

Figure 3.8. Effect of RTX technology on reflections in *Battlefield 5*. Traditional rasterization-based approaches cannot handle light effects such as a flame reflecting off a shiny car door. *Images* © NVIDIA®.

Chapter 4

Learning Visibility

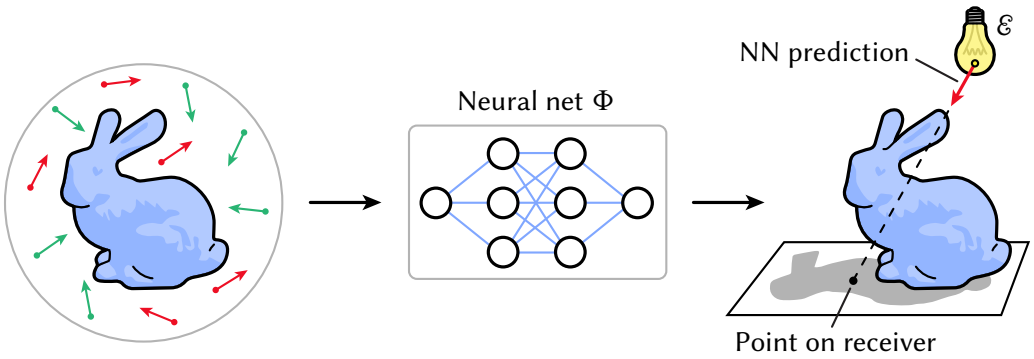
We present a learning algorithm to approximate the visibility function of simple meshes to compute soft shadows. The main goal of our algorithm is to alleviate some of the cost associated with shadow ray queries by training a deep neural network $\Phi_{\mathcal{O}}$ in an offline setting for a given object $\mathcal{O}$. Once the model is trained, we use the network to guide the shadow ray tracing for visibility testing in an augmented environment with receivers. By sampling the global occlusion profiles of a mesh, we aim to learn per-object visibility by treating it as a *binary classification problem*. The premise is that, since the visibility function is a binary operator, the visibility problem can be thought of as classification task with two categories: {nonvisible, visible} $\equiv \{0, 1\}$. We suggest to learn a binary classifier in a supervised manner using a neural network that would return, after training, a value close to the true occlusion bit from any point, in any direction.

A priori, the visibility problem is a challenging one: visibility is full of discontinuities but it also exhibits a fair amount of spatial coherence. Most importantly, it is nonsmooth and not well-behaved at grazing angles. Neural networks, however, have proven to be particularly robust for resolving similar problems, such as 3D shape recognition [SMKL15]. Given the recent success of deep learning on classification tasks [KSH12], considering deep nets to approximate visibility seems like a natural and sensible thing to investigate. The use of neural networks in rendering is particularly recent and, as such, has seen few applications until 2017.

Using deep nets to solve the full visibility problem has, to the best of our knowledge, never been tried. Closest to our work is [DK17], in which the authors train an artificial neural network

to approximate the probability distribution of *selecting* light sources with high contribution (and thus includes visibility). Their algorithm is based on a voxelized representation of the scene where many neural nets are learned for each voxel to determine which emitter should be sampled in explicit path tracing. Our approach differs from theirs as we want to use a single neural network to estimate the visibility function for secondary rays.

In this chapter, we develop a simple algorithm for replacing the shadow ray query with a neural network query. We experiment with low-complexity shapes to learn their visibility profile. We show that, in the context of soft shadows, the network does not need to be perfectly accurate to generate shadows that are close to ground truth. Indeed, for large area lights, the penumbra dominates the umbra. A moderately accurate neural net can thus take advantage of this phenomenon, since the incorrect regions will be attenuated and less visible due to smoothing. At a high level, our visibility pipeline can be decomposed into three stages: sampling, learning, and prediction. (Figure 4.1). We detail each stage in the following sections.



(a) Sampling **(b)** Learning **(c)** Prediction

Figure 4.1. Visibility learning pipeline. First, we sample (a) the visibility to gather training examples. We then train (b) a model to learn the visibility profile of the mesh. Finally, we use this model to predict (c) the visibility for unseen data points to generate shadows.

4.1 Sampling the Visibility

We start by presenting a few heuristics to sample the visibility profile of a mesh. The sampling scheme should ideally be representative of the overall visibility of the object, but this is difficult to

enforce for arbitrary meshes. Instead, we opt for a more conservative approach. Before describing these methods in detail, we first discuss the input representation for our neural network. This representation will then guide our visibility sampling algorithm.

4.1.1. Input Features. Before constructing an artificial neural network for classifying visibility, we need to specify the input format for the model. Recall that evaluating visibility can be thought of as taking two points $\mathbf{p}, \mathbf{p}' \in \mathcal{M}$ and checking if an occluder is present along the line segment $l = \mathbf{p} + t(\mathbf{p}' - \mathbf{p})$ for $t \in (0, 1)$. Since both $\mathbf{p}, \mathbf{p}' \in \mathbf{R}^3$, we can concatenate these two points to obtain a 6-dimensional input. The problem with this representation is that it only captures visibility along a finite line segment, meaning we lose information for the visiblity of points along l that go *beyond* $\mathbf{p}'$ (*i.e.*, when $t > 1$). A better approach is to specify a single point $\mathbf{p}$ and a direction ω to query along, in which case the input is given by

$$\mathbf{x} = (p_x, p_y, p_z, \omega_x, \omega_y, \omega_z)^\top \in \mathcal{X} \subseteq \mathbf{R}^6. \tag{4.1}$$

This representation agrees with how shadow rays are being evaluated in a renderer. Indeed, at each shading point, a ray is traced towards an emitter to determine the visibility bit.

We can reduce the dimension to $\mathbf{R}^5$ by using spherical coordinates to represent the direction vector, that is, $\omega = (\cos\phi \sin\theta, \sin\phi \sin\theta, \cos\theta)$. This representation is convenient as we can rewrite the binary visibility function as $V : \mathcal{M} \times \mathcal{S}^2 \to \{0, 1\}$. This function thus answers the question: "*Given I am at position* $\mathbf{p}$ *looking in the direction* ω*, am I seeing something* (1) *or am I looking into the void* (0)*?*". Note that this representation needs to be sufficient to determine visibility: other values such as depth cannot be encoded in the input features since we do not have this value unless we trace a ray, which is precisely what we want to avoid. In our experiments, we test with both representations. We can now address the visibility sampling component of the algorithm.

4.1.2. Naïve Visibility Sampling. To simplify the problem, we can assume that the object $\mathcal{O}$ to be learned is centered at the origin $\mathbf{0} = (0, 0, 0)$. To sample the visibility of a mesh, we need to provide bounds on the environment surrounding the object. This bounded region needs to

encompass all potential light positions in the final, augmented scene. A natural choice is to use the bounding box of the final scene that includes the receivers. Since we still have access to the BVH for tracing primary rays, the dimensions of this bounding box is readily available to us. It seems more natural, however, to use the bounding sphere of the final scene $S_{\mathcal{M}}$ to sample the positions. Unfortunately, sampling this volume naïvely can result in points *inside* the mesh. To avoid this issue, we can sample positions $\mathbf{p}$ *between* the tight bounding sphere of the object $S_{\mathcal{O}}$ and the larger bounding sphere of the final scene $S_{\mathcal{M}}$. We shall refer to this region as the *sampling region* $S = S_{\mathcal{M}} \backslash S_{\mathcal{O}}$ (Figure 4.2). This further constrains the possible positions of the emitter in the final scene, as we will effectively miss all points that are too close to the mesh. Nonetheless, for distant light sources this assumption is reasonable and avoids introducing an unnecessary cost related to inside-outside determination when generating our datasets.

We now need to sample directions ω in the bounding volume S. Again doing so naïvely, we can uniformly sample the unit sphere for a direction. By sampling $\xi_1, \xi_2 \sim \mathcal{U}(0, 1)$, we can compute angles $\theta = 2\pi\xi_1$ and $\phi = \cos^{-1}(2\xi_2 - 1)$ to obtain a random spherical direction ω. This approach is completely oblivious to the actual position of the mesh, meaning many samples will return unoccluded if their direction points outward. The farther away the sampled position is to the center of the mesh, the more likely it is to yield no occlusion, as many rays will be traced into the void. In other words, $\mathbf{P}[r_{\mathcal{M}}(\mathbf{p}, \omega) = 1] \to 0$ as $\|\mathbf{p}\| \to \infty$ because the solid angle subtended by the object shrinks to zero with the square of the distance. While this is technically a valid technique for sampling the visibility, shadow ray queries are typically more organized if we assume that the receiver is close to the object. We thus need a better sampling scheme.

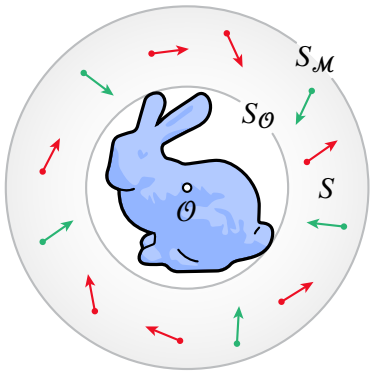


Figure 4.2. Naïvely sampling the visibility can result in very few positive training examples. Sampling positions in the gray region $S = S_{\mathcal{M}} \backslash S_{\mathcal{O}}$ with arbitrary spherical directions can hurt the generalization capacity of a learning algorithm.

4.1.3. Improved Sampling Scheme. By definition, occluded coverage of an object is the solid angle subtended by that object as viewed from the light, projected onto the receivers. Therefore, sampling an area that resembles this solid angle once the origin of the ray is determined should promote a better label distribution of visibility. Perfectly sampling the solid angle for visibility points will always yield a visible bit $y = 1$, leading to complete class imbalance. In addition, the computational cost of determining the triangles of a mesh that are visible from an arbitrary view point is linear in its number of triangles. Considering we wish to sample as many points as possible to have a diverse and balanced training set, we want to limit the time it takes to generate a single example. Determining a tight bounding sphere for an arbitrary mesh requires additional machinery, but we can use the minimum bounding box structure already in place after the BVH construction to have a rough estimate of this bounding sphere. Setting the diagonal of the bounding box as the diameter of the sphere $S^\star_\mathcal{O}$, we are guaranteed that the object is contained in it. Since the bounding box is inscribed by this sphere, we can sample directions in the cone subtended by $S^\star_\mathcal{O}$ to ensure that the visible and nonvisible samples are more or less balanced. Put differently, we effectively bias the directions towards the object $\mathcal{O}$ (Figure 4.3).

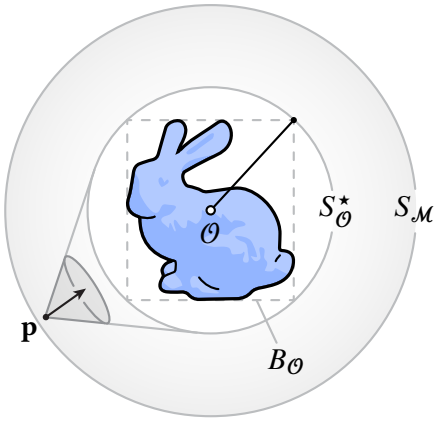


Figure 4.3. Determining the boundind box $B_\mathcal{O}$ (dotted) is relatively cheap after the BVH construction, so we can use it to obtain a loose bounding sphere $S^\star_\mathcal{O}$ around the object.

Note that objects that are tightly enclosed in their bounding box can cause problems. Indeed, if we wish to sample the visibility of a cube then $\mathcal{O} = B_\mathcal{O}$. This implies that all 8 vertices of the inscribed cube will be on $S^\star_\mathcal{O}$. If we sample directions towards this sphere, subtleties at corners

will be missed. A simple solution is to expand the sphere by a small, optional value $\alpha \in [0, 1)$ to cover these regions by increasing the radius $R(S^\star_\mathcal{O}) \mapsto (1 + \alpha)R(S^\star_\mathcal{O})$.

Using this method, we can have better control on the training set label distribution by varying only a single hyperparameter α. We can thus guarantee a perfectly balanced dataset for positive and negative occlusion points, if needed. However, doing so will bias the distribution to be learned. This essentially puts an equal prior probability of positive and negative occurences, which is not necessarily true for the true visibility distribution and may well give bad predictions by over-predicting the unoccluded class. Since we do not know the true visibility distribution a priori, we set $\alpha = 0.15$ heuristically for most meshes we experiment with. Our final sampling scheme is summarized in Algorithm 1 below.

Algorithm 1 SAMPLEVISIBILITY($\mathcal{O}$, R_{max}, α, N)

Require: Object $\mathcal{O}$, max scene radius R_{max}, bounding sphere expansion factor α, number of training examples N

$\mathcal{O} \leftarrow$ CENTERTOORIGIN($\mathcal{O}$) ▷ Move mesh to origin
$B_\mathcal{O} \leftarrow$ GETBOUNDINGBOX($\mathcal{O}$)
$R_{\text{min}} \leftarrow$ GETDIAGONAL($B_\mathcal{O}$)/2 ▷ Get object's bounding sphere radius
if $\alpha > 0$ **then** ▷ Increase smaller sphere size (*mesh-dependent*)
    $R_{\text{min}} \leftarrow (1 + \alpha)R_{\text{min}}$
$(\mathcal{X}, \mathcal{Y}) \leftarrow (\emptyset, \emptyset)$ ▷ Initialize training set
$i \leftarrow 0$
while $i < N$ **do**
    $\xi \leftarrow$ RAND(0, 1)
    $R \leftarrow R_{\text{min}} + \xi(R_{\text{max}} - R_{\text{min}})$ ▷ Sample position in $S = S_\mathcal{M} \backslash S_\mathcal{O}$
    $\mathbf{p} \leftarrow R \cdot$ SAMPLEUNIFORMSPHERE( )
    $\theta_{\text{max}} \leftarrow$ GETCONEANGULARSPREAD($\mathbf{p}$, R_{min}) ▷ Sample direction towards $\mathcal{O}$
    $\omega \leftarrow$ SAMPLEUNIFORMCONE(θ_{max})
    $\omega \leftarrow$ TOWORLD(ω) ▷ Align cone along vector $-\mathbf{p}$
    $y \leftarrow r_\mathcal{M}(\mathbf{p}, \omega)$ ▷ Ray trace
    $\mathbf{x} \leftarrow$ CONCAT($\mathbf{p}, \omega$)
    $(\mathcal{X}, \mathcal{Y}) \leftarrow (\mathcal{X}, \mathcal{Y}) \cup \{\mathbf{x}, y\}$ ▷ Add to training set
    $i \leftarrow i + 1$
return $(\mathcal{X}, \mathcal{Y})$

A few lines in Algorithm 1 deserve further explanation, starting with Line 12. Computing the angular spread of the subtended cone can be done by trigonometry: since $\mathcal{O}$ is centered at the origin, $R_{\min}/\|\mathbf{p}\| = \sin\theta_{\max}$. It would be tempting to simply invert to retrieve $\theta_{\max}$, but inversion is costly. Instead, we note that SampleUniformCone( ) utilizes the term $\cos\theta_{\max}$ to sample a direction. Indeed, a random vector in the cone of directions aligned with the z-axis requires to compute $\cos\theta = (1-\xi_1)+\xi_1\cos\theta_{\max}$ for $\xi_1, \xi_2 \sim \mathcal{U}(0,1)$. We can then retrieve $\sin\theta$ by applying the identity $\sin^2\theta + \cos^2\theta = 1$. The final direction is given by $\omega = (\cos\phi\sin\theta, \sin\phi\sin\theta, \cos\theta)$. Line 14 of the above algorithm orients the canonical cone along the vector $\mathbf{0} - \mathbf{p}$ so the sampled direction points towards $S^\star_{\mathcal{O}}$. This is illustrated in Figure 4.4 below.

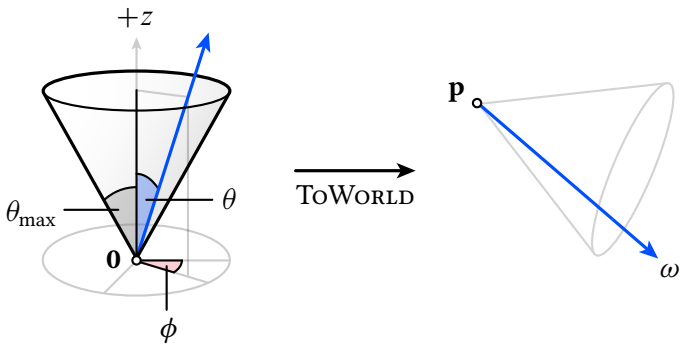


Figure 4.4. To sample a direction towards $\mathcal{O}$, we first sample the uniform cone of directions and then rotate it.

Algorithm 1 was designed with the mindset that it should be general enough to cover a wide variety of shapes. Tailoring the visibility sampling problem to the complexity of the shape can quickly become a convoluted task, especially if the object exhibit many concave regions. Moreover, this process can be easily parallelized by leveraging a parallel implementation of ray tracing. Having an infinite generator for the training set is also convenient since it allows us to generate as many examples as we want. In particular, we can sample a fresh training set for each epoch of our training algorithm. This is highly desirable to reduce the generalization error of our learned model. With an efficient and dynamic visibility sampling routine, we now turn our attention to designing a classifier that learns the underlying visibility distribution of this dataset to predict shadows.

4.2 Designing a Visibility Classifier

Recall that the main idea of constructing a visibility classifier is to shift the burden of shadow ray tracing to an offline setting. If the model is small enough, we can hope to use it instead of the scene BVH to speed up the computations of shadow regions. What we want is a neural network trained for a specific occluder *without any receiver*. The premise is that, if a sufficient number of training examples regarding the occluder visibility profile are given to a network, it could internally reconstruct a spatial representation of the mesh. This learned representation could then be used on an augmented scene with receivers to predict soft shadows generated by arbitrarily-placed light sources.

4.2.1. Network Architecture. As previously mentioned, the network complexity should be kept as simple as possible to avoid introducing any major overhead during shadow inference. The network shall be used at each shading point, meaning propagating the input through the network should be fast. In what follows we describe our choice of architecture.

The universal approximation theorem states that a single layer net can, in theory, approximate any function arbitrarily well. However, we also know that a deeper model usually performs better on classification tasks, so our approach is to use few layers with very few neurons. We opted for a two-layer neural network with 128 neurons at each layer as a tradeoff between simplicity and efficiency. We use ReLU activation units for each layer to obtain well-behaved gradients. Since we need the network to predict a binary value, we use a sigmoid activation function at the output layer to squash the final layer value into the unit interval $[0, 1]$. A simple 0.5-thresholding is then employed to determine the occlusion category. To increase training stability, we use batch normalization [IS15]. To train the model, we use the Adam optimizer [KB14] with a learning rate of $\eta = 0.01$ and exponential decay rates $\beta_1 = 0.9, \beta_2 = 0.999$. We train using mini-batch with a batch size of $m = 128$.

4.2.2. Loss Function. The choice of loss functions is critical for learning algorithms. A loss function that is not well adapted to the problem will result in poor performance of the estimator. Due to its ubiquity in deep learning for classification problems, we choose to use binary cross-

entropy as our objective loss function:

$$\begin{aligned}\mathcal{J}(\boldsymbol{\theta}) &:= -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)}\log\hat{p}^{(i)} + (1-y^{(i)})\log\Big(1-\hat{p}^{(i)}\Big)\Big] \\ &= -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)}\log\big(\Phi(\mathbf{x}^{(i)};\boldsymbol{\theta})\big) + \big(1-y^{(i)}\big)\log\big(1-\Phi(\mathbf{x}^{(i)};\boldsymbol{\theta})\big)\Big],\end{aligned} \tag{4.2}$$

where m is the batch size, $y^{(i)}$ is a true label, $\hat{p}^{(i)}$ is a predicted belief on the label, and Φ is our model. This cost function has a simple interpretation. Indeed, if we see the output of Φ as a probability $\hat{p}$ of being either visible or nonvisible then we can write $\Phi(\mathbf{x}^{(i)};\boldsymbol{\theta}) = \mathbf{P}\big[y^{(i)} = 0 \,|\, \mathbf{x}^{(i)}\big]$ so that $\mathbf{P}\big[y^{(i)} = 1 \,|\, \mathbf{x}^{(i)}\big] = 1 - \Phi(\mathbf{x}^{(i)};\boldsymbol{\theta})$. Writing this as the probability of a Bernoulli trial, we obtain

$$\mathbf{P}\big[y^{(i)} \,|\, \mathbf{x}^{(i)};\boldsymbol{\theta}\big] = \big[\Phi(\mathbf{x}^{(i)})\big]^{y^{(i)}}\big[1-\Phi(\mathbf{x}^{(i)})\big]^{1-y^{(i)}}. \tag{4.3}$$

Assuming that the data is independent and identically distributed, we can write the likelihood by simply taking the product across the sampled mini-batch:

$$\ell\big(\boldsymbol{\theta} \,|\, \mathbf{x}^{(i)}\big) = \prod_{i=1}^{m}\big[\Phi(\mathbf{x}^{(i)})\big]^{y^{(i)}}\big[1-\Phi(\mathbf{x}^{(i)})\big]^{1-y^{(i)}}. \tag{4.4}$$

Binary cross-entropy (4.2) is then derived by taking the logarithm and rearranging terms. From an information-theoretic point of view, this loss function is particularly interesting for classification problems since it essentially minimizes the KL-divergence between two distributions: the true visibility distribution of the mesh and the visibility distribution approximated by the network.

4.2.3. Other Hyperparameters. Our training set consists of $N_t = 8 \times 10^6$ visibility points that can be resampled at every epoch. We use a fixed validation set of $N_v = 2 \times 10^6$ examples to assess the generalization capacity of the network. To initialize the weights and biases, we adopt the technique from He *et al.* [HZRS15] specifically tailored to ReLU-like activations. This initialization heuristic uses the size of the previous layer to sample the weight entries according to a Gaussian $w_{ij}^{(\ell)} \sim \mathcal{N}\big(0, \sqrt{2/N_{\ell-1}}\big)$ for $1 \leqslant \ell \leqslant d$. This procedure is similar to Xavier/Glorot

initialization [GB10] and keeps the variance of the input gradient and output gradient the same. We set the bias to zero for layers, as recommended by [HZRS15].

Below is a table summarizing our choice of hyperparameters for our model.

Hyperparameter		**Value**
Architecture	$\Phi_{\mathcal{O}}$	$5/6 \to 128 \to 128 \to 1$
Activation functions	g	ReLU $\to$ ReLU $\to$ Sigmoid
Regularizer	—	Batch normalization
Parameter initialization	$\boldsymbol{\theta}$	He *et al.* (weights), biases $\equiv 0$
Loss function	$\mathcal{L}$	Binary cross-entropy (log-loss)
SGD Optimizer	—	Adam ($\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$)
Training set	N_t	8 000 000 (dynamic resampling)
Validation set	N_v	2 000 000
Batch size	b	128
Number of training epochs	K	30
Minimum sampling radius	R_{min}	1.0
Final scene radius	R_{max}	7.0
BSphere expansion factor	α	0.15

Table 4.1. Summary of hyperparameters for training.

4.3 Methodology

4.3.1. Implementation. To test our algorithm, we implemented a ray tracer in C++11 with different emitter models, such as point lights and area lights, and a Lambertian reflectance model. We also implemented a simple direct illumination Monte Carlo integrator that can handle both types of lighting. The mesh can be loaded in Wavefront OBJ format and an SAH-based BVH is constructed at run-time. We use the construction of [Wal07] to quickly build the tree in parallel. Sampling is done in the renderer, where data is serialized and saved to file for training.

Our neural network is implemented in Python with Keras [C+15] with a TensorFlow backend [AAB+15]. A bridge is created between the C++ renderer and the Python trainer to fetch new training data on demand to allow for different data to be used at each epoch. The model is saved as a checkpoint at the end of each epoch. To load the weights back into the renderer, we implemented a custom wrapper that loads each weight and bias matrices into memory. We then replace any

shadow ray queries with our model.

Implementing the training and inference under a unified language turned out to be relatively tricky and unnecessary. Given the limited support for prototyping deep learning models in C++ and the wide variety of learning packages available in Python, it is not worth training in the renderer itself. Since this is a one-time cost that needs to be done before any rendering takes place, we opted for the approach that gave us the most freedom for experimenting. A Python implementation is robust and supports training on the GPU with CUDA [NBGS08], which enables rapid iterations.

4.3.2. Experimental Setup. Our experimental setup consists of various meshes from low to moderate complexity. We start with two simple shapes, namely the unit sphere and the unit cube. These objects—albeit trivial—are particularly valuable for testing. The sphere is perfectly rotationally invariant, while the cube has several rotational symmetries. We then move to more difficult meshes such as the Stanford bunny, which is a highly nonconvex object, and the torus, which is a genus-1 manifold.

For all tests, we use a scene radius of $R_{\max} = 7$ and scale down the geometry so that $R_{\min} = 1$. We monitor loss and accuracy at each epoch during training. For each object, we evaluate the ground truth visibility using a point light source to have a crisp representation of the umbra. The network is then evaluated at different positions in the sampling region S. Finally, we can replace the point light with a moderately small spherical light source to obtain soft shadows. In this scenario, the solid angle formulation of the rendering equation is used to evaluate direct illumination for diffuse surfaces:

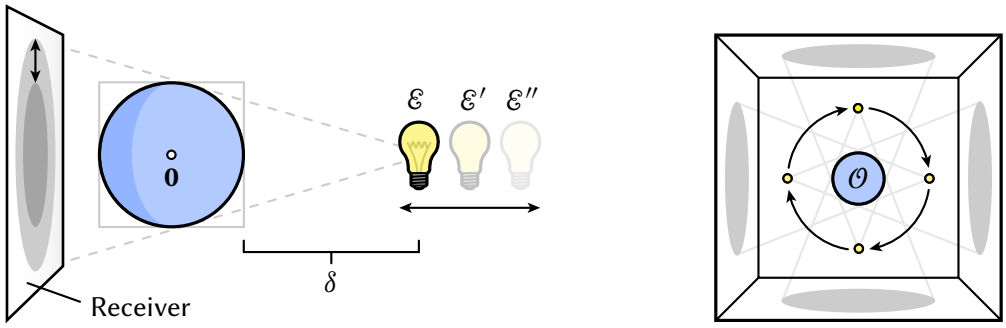
$$\begin{aligned} L_o(\mathbf{x}, \omega_o) &= L_e(\mathbf{x}, \omega_o) + \int_{\mathcal{S}^2} f_s(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) |N(\mathbf{x}) \cdot \omega_i| \, \mathrm{d}\sigma(\omega_i) \\ &\approx \int_{\mathcal{H}^2} \frac{\rho}{\pi} L_i(\mathbf{x}, \omega_i) \Phi_{\mathcal{O}}(\mathbf{x}, \omega_i) |N(\mathbf{x}) \cdot \omega_i| \, \mathrm{d}\sigma(\omega_i) \\ &= \frac{\rho}{\pi} \int_{\mathcal{H}^2} L_i^{\Phi}(\mathbf{x}, \omega_i) \, \mathrm{d}\sigma^{\perp}(\omega_i), \end{aligned} \tag{4.5}$$

where ρ is the surface albedo and L_i^{Φ} is the combined radiance and learned visibility. We approximate Equation 4.5 using a Monte Carlo estimator with low sample count. In our experiments,

we use 4 samples per pixel to avoid aliasing artifacts.

4.3.3. Learned Visibility Visualization. To visualize the learned visibility in three dimensions, we need to design a tool to assess how well the model generalizes to unseen data, namely a small collection of points that were not part of the training set. To do so, we first note that moving the emitter along a particular dimension should produce shadows that are structurally similar. For instance, moving an emitter along the direction parallel to the receiver surface normal, we can increase or decrease the size of the shadow (Figure 4.5a).

To see how the learned model produces shadow regions from varying positions, we can project the occlusion profile on all six walls of an axis-aligned box. This allows us to vary one dimension at a time to see how the model behaves for predicting shadows on receivers with different angles (Figure 4.5b). For symmetric shapes like the sphere and the cube, the shadows should be the same for all projections, assuming the distance to the occluder and receiver is held constant. Combining both (a) and (b), we thus have a relatively robust and intuitive visualization tool for our network to compare with ground truth shadows. In the case of a sphere for instance, we expect all learned shadows to resemble a circle. Taking the mean squared error between true and predicted can then be used to measures visual similarity.



(a) Moving along one dimension

(b) Box projection

Figure 4.5. To visualize our model, we can either vary the position of the light $\mathcal{E}$ in one dimension or project on receivers with different angles.

In our tests, we use a $5 \times 5 \times 5$ box and project using the best model on the validation set from varying light depth. This depth is defined as the shortest distance between $\mathbf{p}$ and the object's

bounding box $B_{\mathcal{O}}$. More formally,

$$\delta := \delta(\mathbf{p}, \mathcal{O}) = \inf\{t > 0 : \mathbf{p} - t\,\mathbf{p}/\|\mathbf{p}\| \in B_{\mathcal{O}}\}. \tag{4.6}$$

Secondly, we can visualize the pre-activation output of our model. Since we are classifying points by thresholding, we can simply disregard all terms in the rendering equation and return the approximate visibility spectrum $V(\mathbf{x}, \omega) \approx \widetilde{\Phi}_{\mathcal{O}}(\mathbf{x}, \omega)$, where $\widetilde{\Phi}_{\mathcal{O}} : \mathcal{X} \to [0, 1]$ is the continuous output of the model *before* thresholding. This visualization on receivers is particularly interesting as it allows to assess gray regions where the network might be uncertain of its predictions. In classification problems, we effectively separate the space into two subregions. Borrowing intuition from support vector machines (SVM) in machine learning, we can think of visualizing the raw output as the margins of the learned hyperplane. We expect the network to be highly confident in clearly visible (0) and occluded (1) zones, but unsure in transition regions, *i.e.*, $\widetilde{\Phi}_{\mathcal{O}}^{-1}\big([0.5 - \epsilon, 0.5 + \epsilon]\big)$ for some $\epsilon > 0$. We thus foresee that the learned model will produce uneven shadow silhouettes while retaining the overall shape of the umbra region.

Chapter 5

Results

We present our experimental results. Each scene network was trained offline and we show the accuracy and loss plots on their respective training and validation sets. The accuracy corresponds to the proportion of true results among the total number of cases observed in the dataset: it measures how successful our classifier is at correctly identifying occluded and unoccluded samples. We are mostly interested in the validation error of our models. We display the evolution of our binary classifiers as they are trained and compare with ground truth, ray traced images. The augmented test scenes are constructed by adding an infinite receiving ground plane at $z = -1$ on which shadows can be cast. We vary the position of the light source in the positive hemisphere above the object to visualize the hard and soft shadows in different lighting scenarios. These positions correspond to $p_x \in \{0, \pm 1.5, \pm 3\}$ with fixed $p_y = 1$ and $p_z = 2$. We also show the raw output of our best models (on the validation set) using the box projection test. The pre-activations are colored to better visualize the output values of our neural network classifiers.

5.1 Sphere

We start with the simplest possible shape, namely an analytical unit sphere. We first note that the model reached a very high level of validation accuracy (0.984) early on in the training, as shown in Figure 5.1 below. We also observe that the model reaches a higher level of accuracy on the validation set than on the training set on 2/3 of the epochs, which might be due to class imbalance. We shall refer to our model at epoch i as $\Phi^{(i)}_{\text{Sphere}}$.

(a) Sphere Accuracy **(b)** Sphere Loss

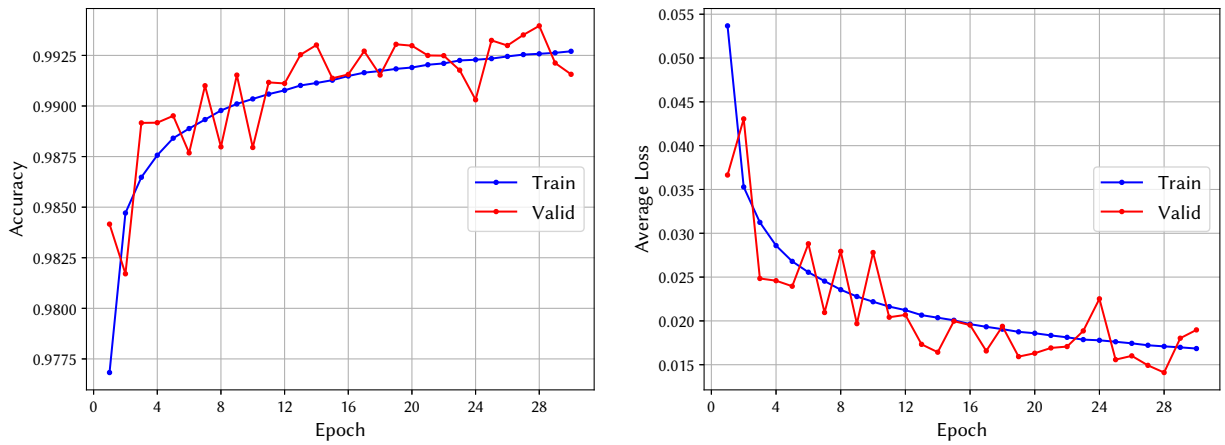


Figure 5.1. Training curves for Sphere scene.

The top row of Figure 5.2 shows how the model evolves through time as it is trained by back-prop. The high validation accuracy early in the training process is directly reflected in the quality of the shadows, as $\Phi^{(5)}_{\text{Sphere}}$ already closely resembles the ray traced (RT) ground truth. While the neural network is quickly able to reconstruct the visibility profile of the ball, this process rapidly stagnates. The difference between the worst and best learned (L) models for this particular setup is almost unnoticeable. The middle second and third row of the same figure compares ray tracing with our approach for different point light positions. Here, the best model $\Phi^{(28)}_{\text{Sphere}}$ is used. Even if the visibility was sampled in the bounding region with $R_{\text{max}} = 7$, the model fails to capture the shadows beyond $p_x = \pm 3$, even if it can approximate point closers to the mesh relatively well. In the last row of the figure, we replaced the point light in the top row with an area light with radius $R_{\mathcal{E}} = 0.5$. Visually, both penumbra regions look similar but the MSE heatmap shows that our learned algorithm has difficulties resolving penumbra at the front of the object.

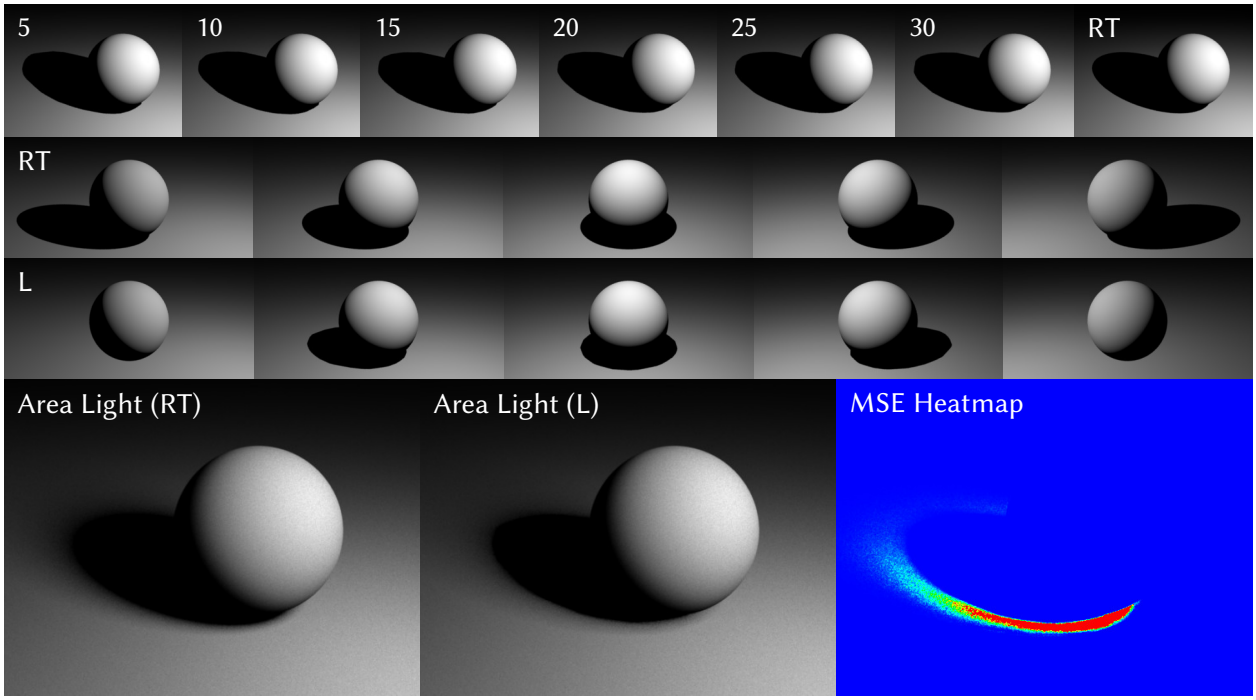


Figure 5.2. Visualization of the learning process for the SPHERE scene.

Figure 5.3 shows the results for the box projection test using our best model. Each column corresponds to a different position **p** for the light, and the raw pre-thresholded neural network output is plotted for two different light depths δ. All projections correctly identify the cast shadow as a circle, with the exception of the top projection (third column) having more difficulties. The binary (thresholded) shadow for both depths are superposed and compared with the ray traced umbrae. This region should be identical for all projections as the sphere is rotationally invariant.

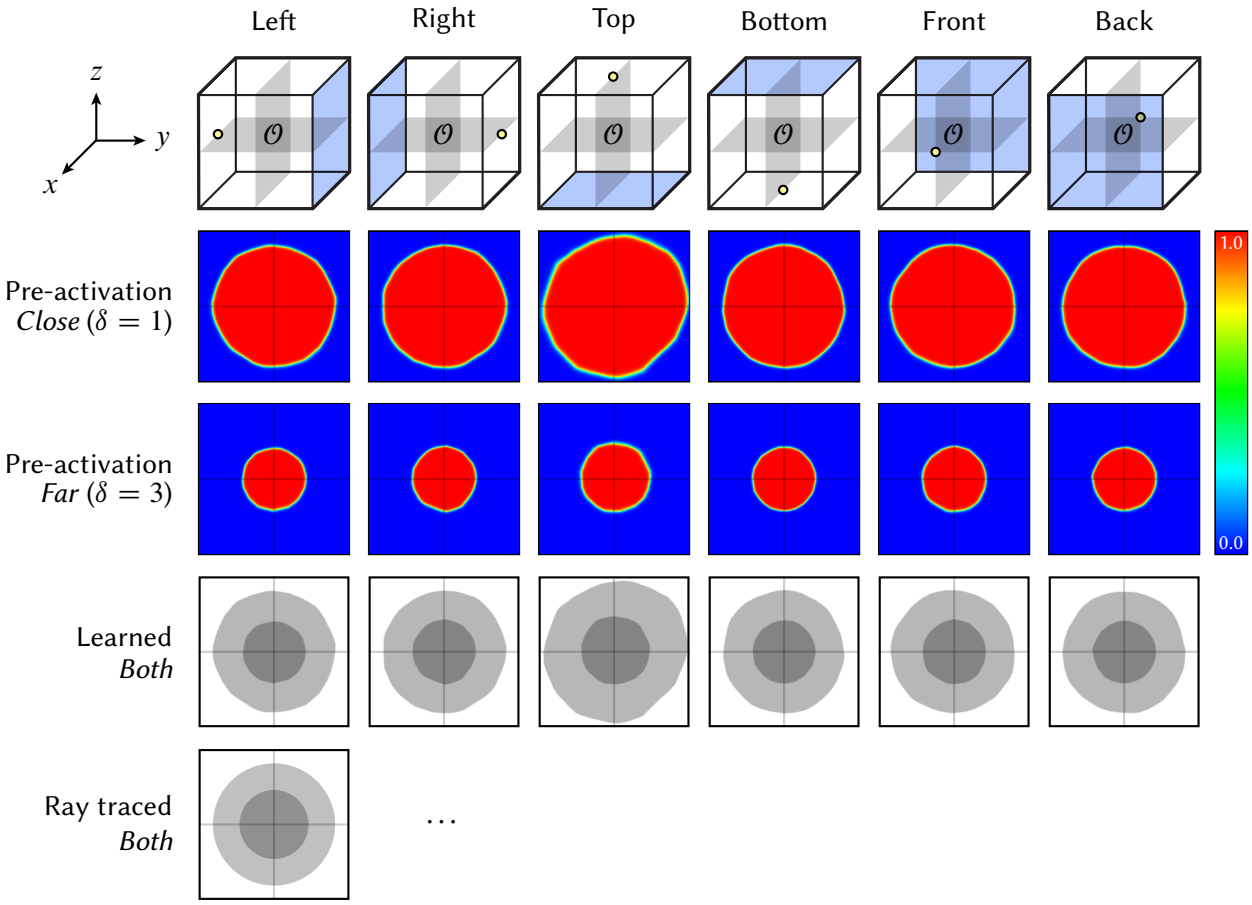
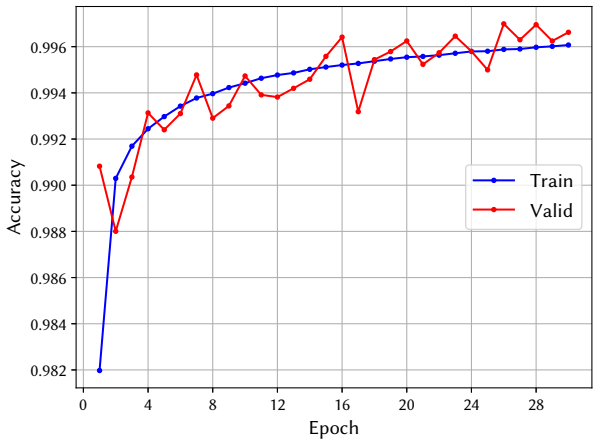


Figure 5.3. Box projection of SPHERE scene.

5.2 Cube

The second scene we tested is the unit cube. While it has several rotational symmetries, the unit cube has many sharp corners that could be difficult to resolve for the neural network with a smooth loss function like BCE. Figure 5.4 depicts the accuracy and loss of the network during training. Again, our model obtains a remarkably high accuracy early on in the training process, with a score of 0.991 after the very first epoch. Here, the model mostly performs better on the training set, which is to be expected.

(a) CUBE Accuracy



(b) CUBE Loss

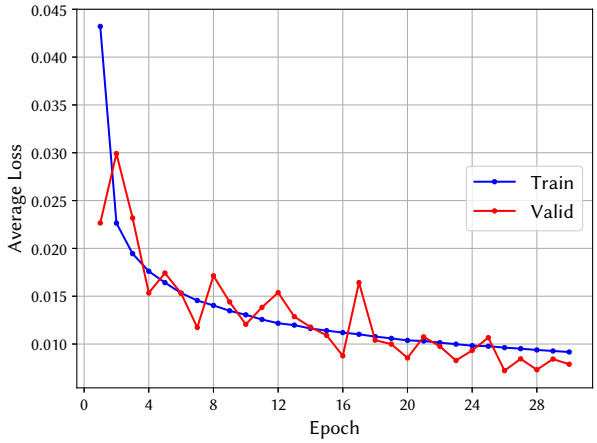


Figure 5.4. Training curves for CUBE scene.

Figure 5.5 similarly shows the training process, with little to no improvements after the first few epochs. The learned shadow region for the cube is much smaller than the ray traced umbra region and also suffers from Peter Panning. One important difference with the unit sphere is that our best model $\Phi^{(26)}_{\text{Cube}}$ is able to generalize to farther light positions, as shown in the third row, even though the shape of the shadow is not structurally accurate. The network has many issues resolving sharp corners and oversmooth these regions. This especially severe when the light is directly above the cube (third row, third column). Replacing the point light with an area light, the shadow is almost inexistant, as displayed by the MSE heatmap.

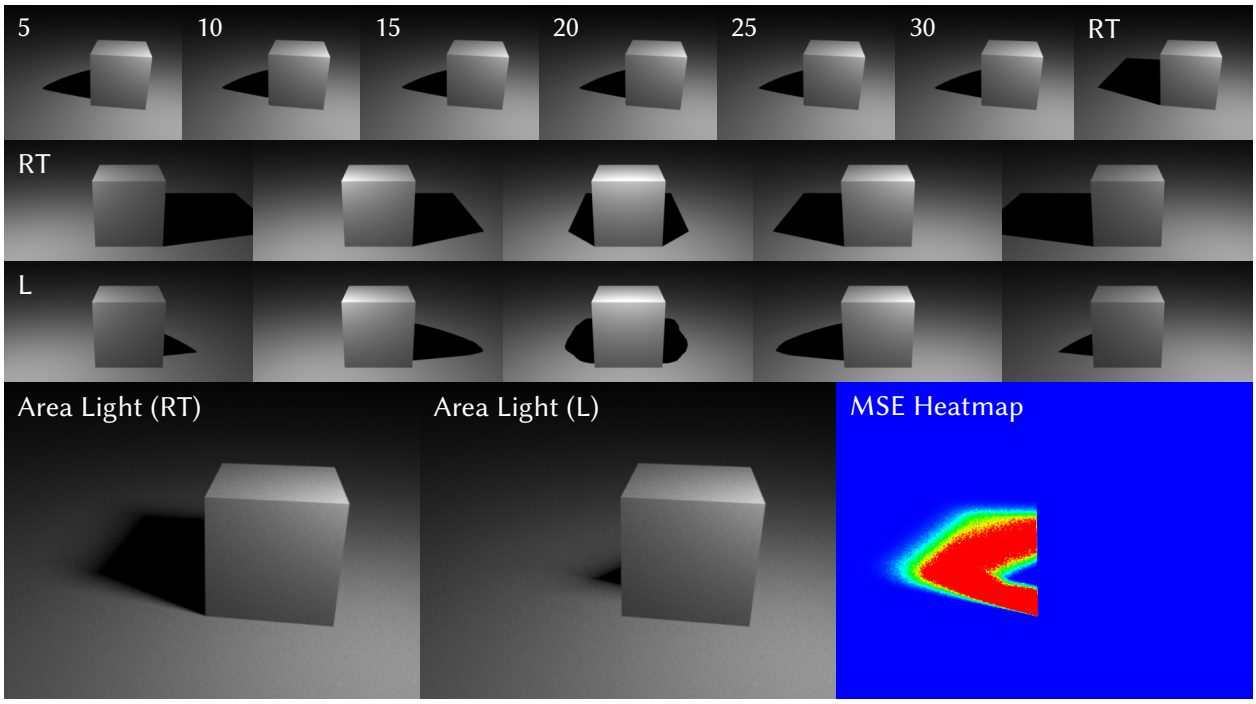


Figure 5.5. Visualization of the learning process for the CUBE scene.

This poor learned representation is demonstrated in Figure 5.6 where we used farther depths to avoid projecting on other surrounding walls (*i.e.*, the shadow would be too large to be contained on a single 5×5 panel). At $\delta = 2$ the network completely fails to capture the square shape of the shadow in all but the bottom projection. For a slightly farther light position at $\delta = 3$, we note that the network is somewhat capable of detecting the edges of the umbra but this representation lacks any form of symmetry. Moreover, the network learns two components for the umbra in most projections. For the top projection (third column), the learned model does not distinguish between the two light positions and predicts occluded for the entire plane. This is reminiscent of the sphere scene that learned a less accurate top representation of the visibility. The only success story for the unit cube is the bottom projection at $\delta = 3$, where our model learns a crisp representation of the side of the object

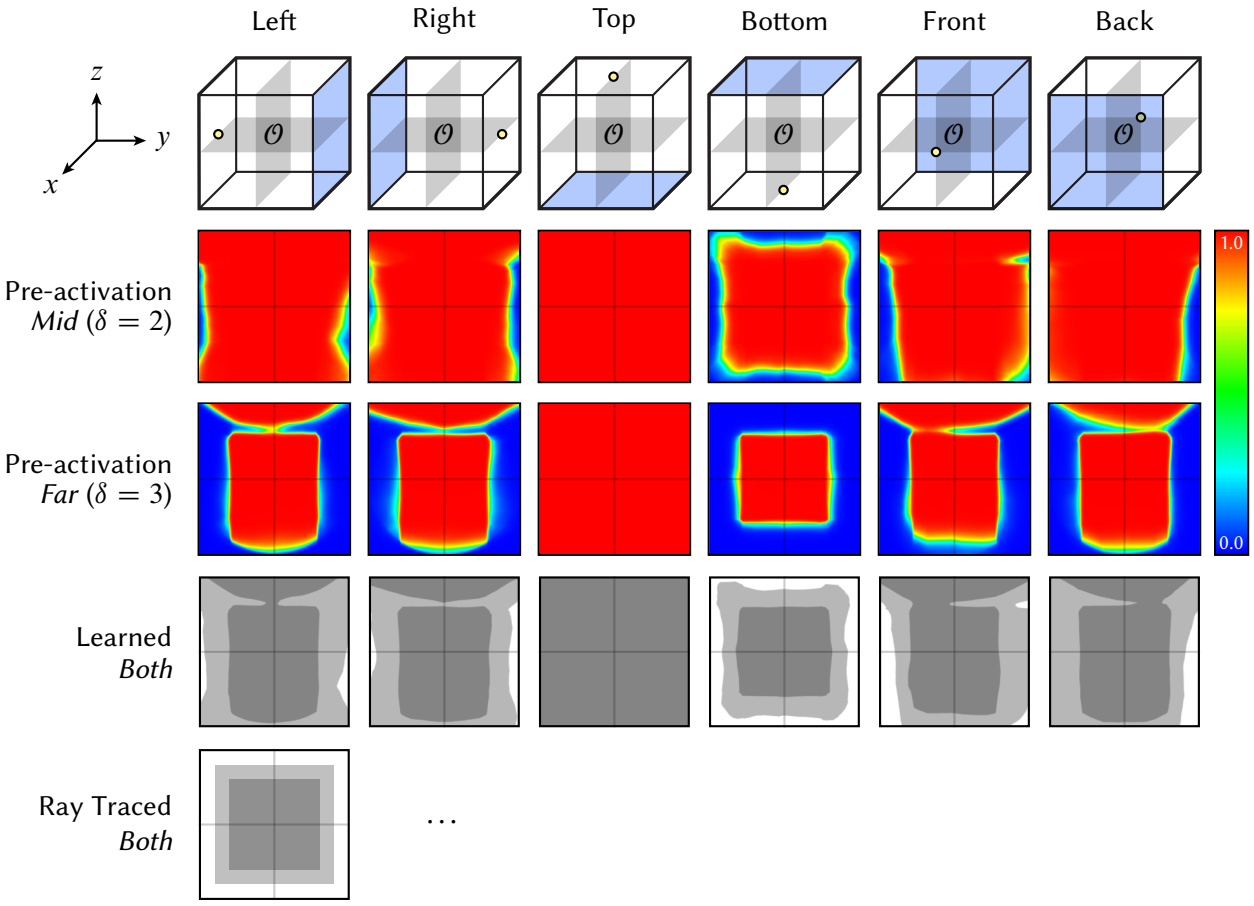
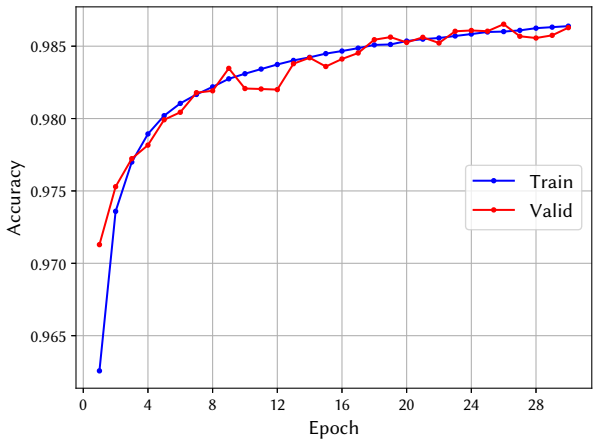


Figure 5.6. Box projection of CUBE scene.

5.3 Stanford Bunny

The sphere and the cube are both convex objects with many axes of symmetries. We next tested on a nonconvex shape that has no axial symmetries, namely the classical Stanford bunny. This mesh is particularly important has it has both convex (body) and concave (ears) components. While the body is more or less smooth, the ears are sharp at their tip, which justifies this choice of object as it can be seen as a concave mixture of the previous two objects. Figure 5.7 shows once again the training and validation accuracy/loss over time. Most surprising is the fact that two curves are very similar and show little variance in the second half of training. The validation accuracy is lower than the previous examples at the beginning, starting from 0.973 and reaching 0.987 at the end of training.

(a) Stanford Bunny Accuracy



(b) Stanford Bunny Loss

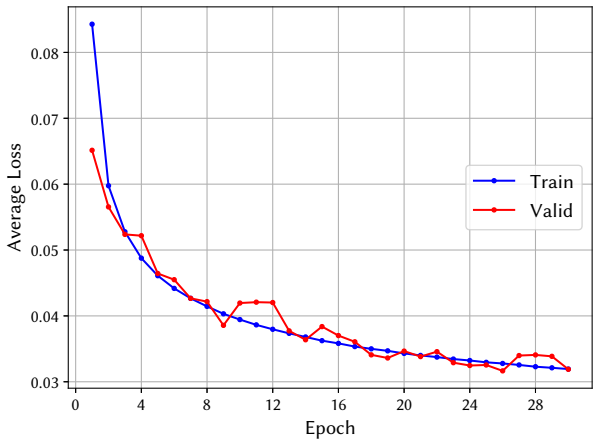


Figure 5.7. Learning curves for the Stanford Bunny scene.

To test our best model $\Phi^{(26)}_{\text{Bunny}}$, we placed the camera in a position that would allow to clearly see the shadow cast by the bunny ears. Rendering the bunny at every 5 epochs (Figure 5.8) revealed that the model's internal representation of the mesh did evolve, which is not something that was observed with previous objects. However, the neural network overcompensates for the right ear (left on the figure), as shown by a increase in length of the shadow. While the model adapts surprisingly well to the concavity of the head of the bunny, it still fails at displaying shadows when the light is too far. When the point light is replaced with an area light, the model generates

arguably realistic soft shadows, but has issues at the base of the object where the umbra dominates as shown by the MSE heatmap.

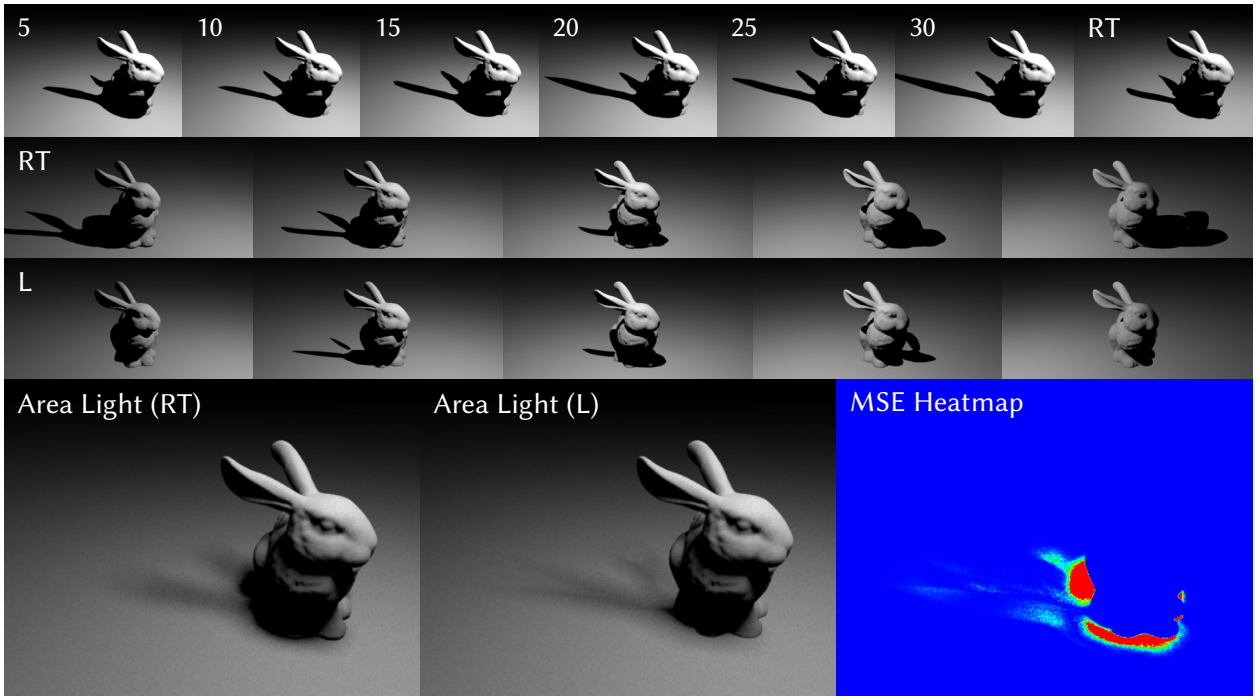


Figure 5.8. Visualization of the learning process for the STANFORD BUNNY scene.

The box projection of the Stanford bunny in Figure 5.9 was only performed for $\delta = 2$, as a smaller values produced shadows that were too large to be contained on a 5×5 wall, and larger values all collapsed to the unoccluded class. We observe that most predictions are structurally correct, but they lack fine details. The model's exaggeration on the bunny right ear previously noted is very well captured by the back projection. The neural network again shows poorly formed top projected shadows, as the learned model could not learn this view even at a short distance.

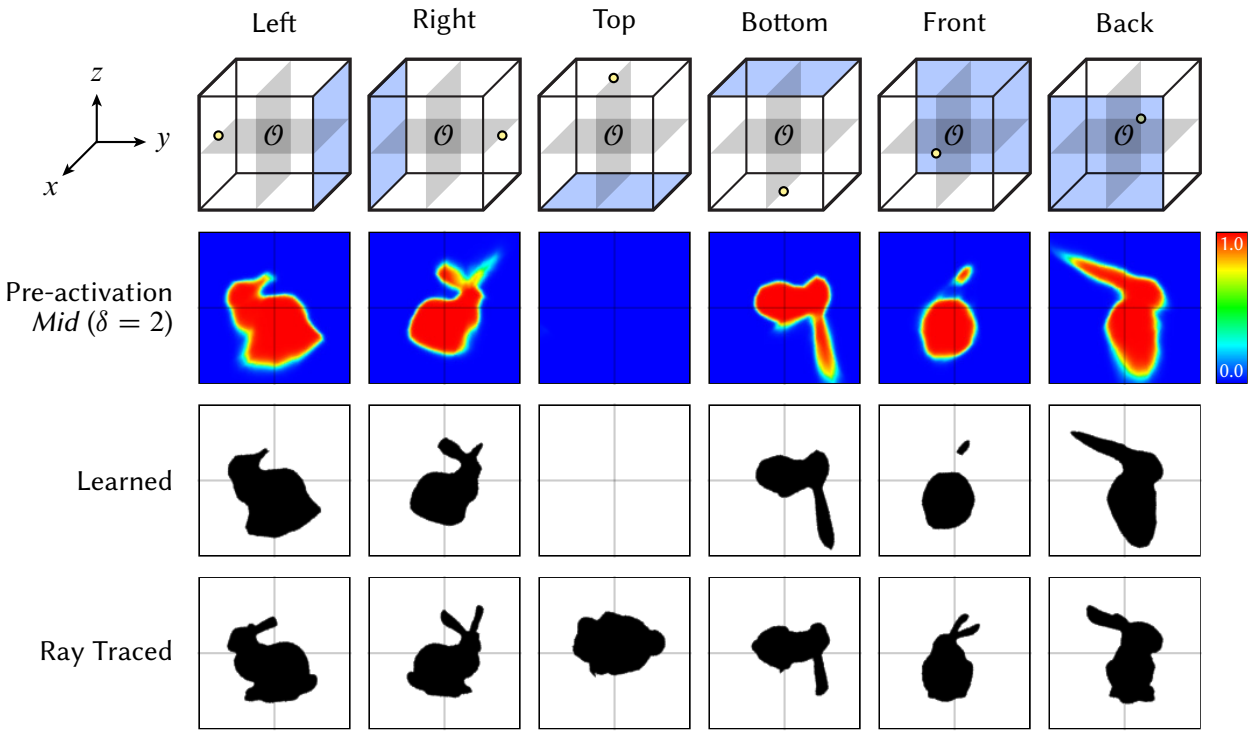


Figure 5.9. Box projection of Stanford Bunny scene.

5.4 Torus

Our last object is a torus to test nonzero genus shapes. This torus is angled at 45° in the y-axis to break symmetry. The goal of this mesh is to determine if our neural network can learn a representation of the hole while retaining a single spatial component. The curves in Figure 5.10 shows stability in the training process, eventually reaching an accuracy of 0.993 at epoch 29.

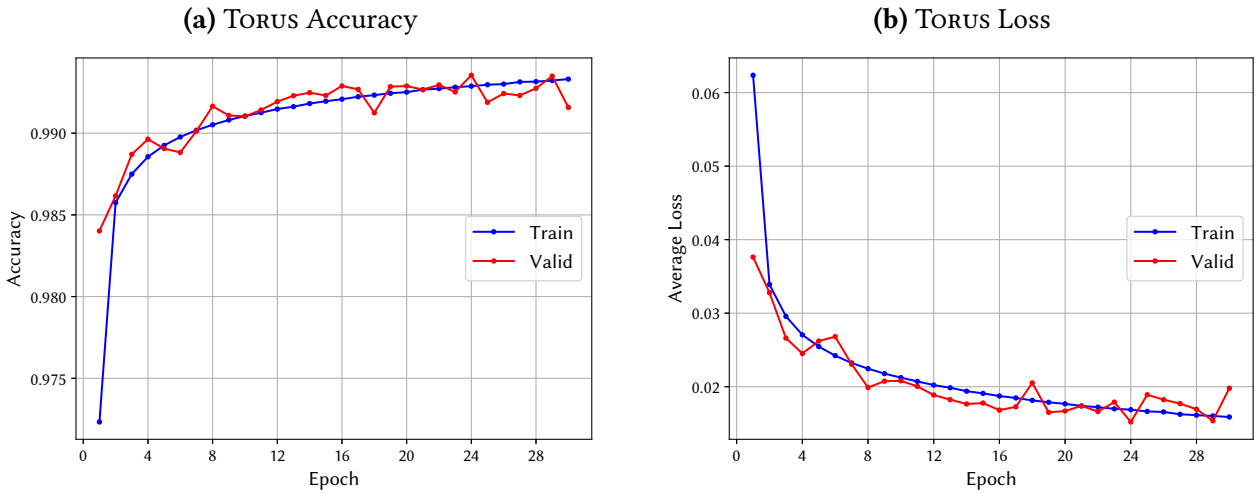


Figure 5.10. Learning curves for the Torus scene.

Figure 5.11 shows that the visual accuracy of the shadow stabilizes early on in the training. Our best network $\phi^{(29)}_{\text{Torus}}$ is not capable of capturing the entiery of the donut shape, which results in a banana-shaped umbra that is far from correct. When moving the light position, the network completely fails at identifiying any shadow region and collapses to the unoccluded class, thus rendering the network useless for generating shadows. The last row of the figure shows the result of replacing the point light with an area light, but this case is already a lost cause as most of the torus visibility profile cannot be represented by the model.

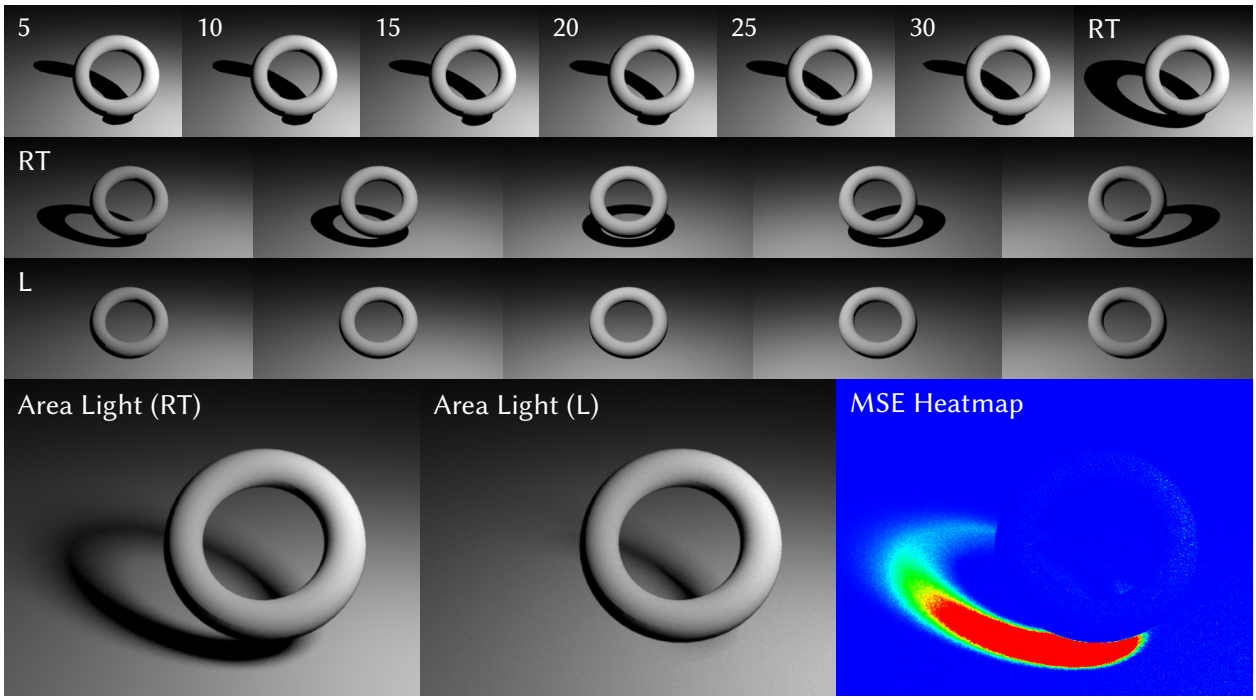


Figure 5.11. Visualization of the learning process for the Torus scene.

Finally, the box projection of the torus is shown in Figure 5.12. The network partially succeeds at reconstructing the visibility of the object at mid-range, as shown by all but the top projection. The network is able to understand the presence of a hole in the geometry, but reverts to a genus-0 shadow for a farther light position, indicating that it did not, in fact, learn the true geometry of the manifold. This recurrent structural error is indeed shown by the bottom, front and back projections, where the shadow resembles a banana instead of a donut. The problem with the top projection persists, as it fails at "closing" the shadow for mid-range and completely disregards the presence of occluded points at far-range. We did not superpose the learned binary shadows in the figure for clarity.

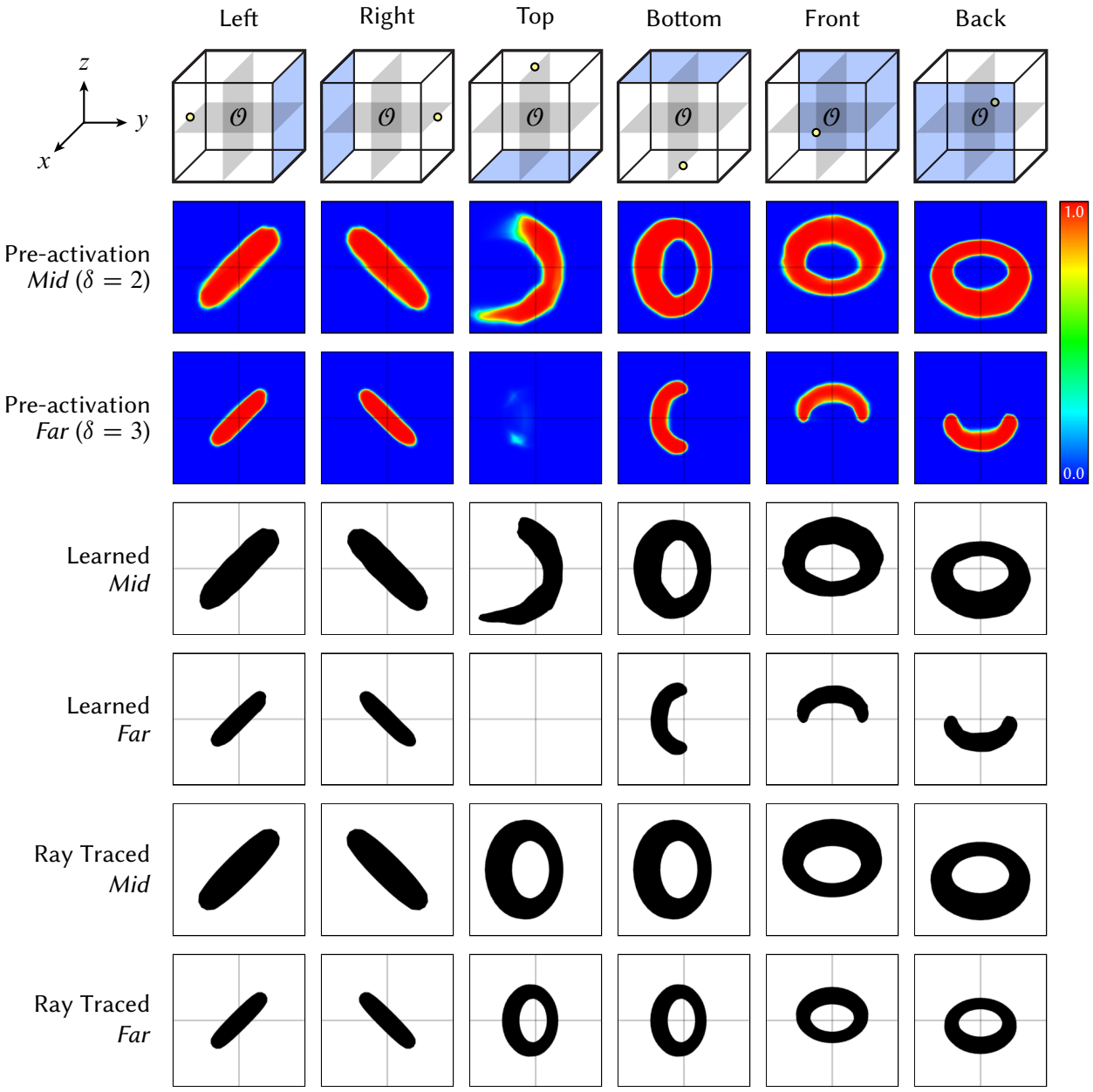


Figure 5.12. Box projection of TORUS scene.

Chapter 6

Discussion and Conclusion

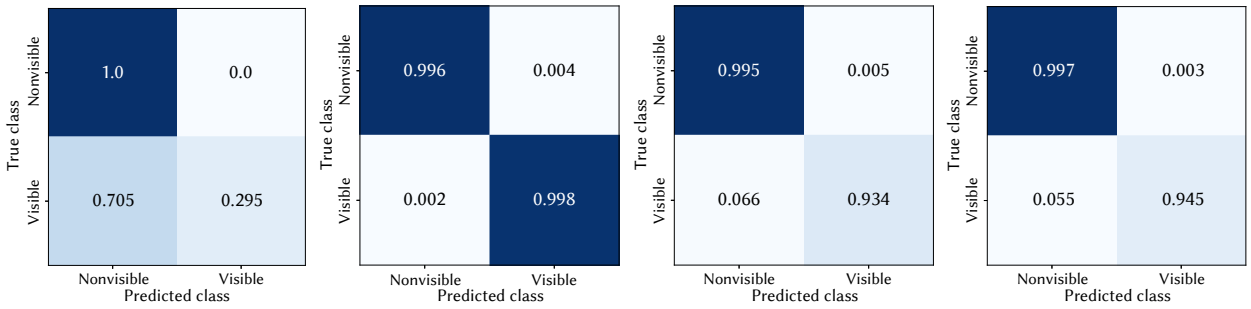
In view of the results, we unfortunaly cannot claim to have achieved our initial goal of using a deep net for accurate shadow generation. In what follows, we discuss our approach and describe the main limitations of our model. We then present several avenues of future work for learning visibility.

6.1 Method Analysis

6.1.1. Generalization Capacity. It is clear that the network capacity is not high enough to capture all subtleties in the geometries to learn. The fact that the network is able to vaguely recognize the object at close to mid-range is encouraging, but most shadows predicted have a tendency to be too smooth in sharp regions. In all test cases, the network was unable to generalize past a bounding sphere of radius 3, which is less than half the trainable radius that was set to $R_{\max} = 7$. The reasons for this are unclear, since the sampling routine was specifically designed to be agnostic to the distance to the mesh. Indeed, the expected number of hit/miss rays along a direction should be constant as we are sampling the subtended cone. One possible explanation is that two layers are simply not enough to capture the visibility depth. Past a certain point, the network prefers to predict the unoccluded class for all points. While this is visually better than always predicting occluded, it significantly constrains the potential light positions. For more complex meshes, we observed that this is particularly severe as the shape of the learned visibility

rapidly degenerates until it collapses to a single occlusion category.

Figure 6.1 shows the normalized confusion matrices on the validation set for all scenes. Here, the color reflects the distribution of data points (*e.g.*, a darker color means more examples; see Figure 6.2 for the training data distributions). We observe that the number of false positive and false negative is relatively low for the last three scenes. However, the SPHERE scene has a very high false negative rate and a zero false positive rate. In addition, it has perfect positive error and poor true negative error. This is an interesting behaviour as the SPHERE scene was arguably the most successful case visually. This is symptomatic of bigger problems with the visibility sampling scheme as this error was not immeditately noticed by our rendered test images and the box projection visualization.



(a) SPHERE **(b)** CUBE **(c)** STANFORD BUNNY **(d)** TORUS

Figure 6.1. Normalized confusion matrices for all scenes.

A deeper model could potentially alleviate the problems discussed thus far, but it would also come at the cost of longer evaluation time which is already an important limitation. We did try with many different architectures, from a single layer perceptron to a deep neural network with up to 6 layers. Empirically, we saw that fewer neurons in the first layer significantly hurt the quality of the shadows the model can produce. Deeper models, besides being much longer to train, did not seem to generate more accurate shadows overall, at least not as good as we would like them to be. This further suggests that the multilayer perceptron's capacity might simply be too constrained for something as complex as visibility.

Although this evaluation can be made efficient by moving it onto the GPU, the vanilla neural network model still treats all training examples independently. Recasting the visibility problem

in the convolutional framework to take advantage of correlated data is not a straightforward task, as convolutions naturally act on array-like representations. However, recent work in spherical convolutional neural networks [CGKW18] has shown that it is possible to use ConvNets for spherical signals and can possibly be applied to improve visibility learning. Combining this idea with recent regularization techniques for neural networks, such as penalizing low entropy output distributions [PTC+17], could potentially produce important improvements over our method.

6.1.2. Dataset Generation. We observed that the same architecture can produce highly varying results depending on the mesh. For instance, even if the cube mesh has a relatively simple piece-wise linear analytical form for ray intersection, the network could only reconstruct one of the six sides of the mesh correctly using a large amount of training examples. One possible explanation for such poor performance is that the bounding sphere heuristic for sampling this object is nonoptimal. Even if we increased $S_{\mathcal{O}}$ by a small factor α, few visibility points actually captured the corners of the cube. While this argument makes sense for the cube, it cannot be applied to the bunny mesh as many sharp traits were well estimated at close range. On the other hand, the sphere mesh was very well approximated but the bounding region was also spherical, which could suggest that a sampling volume that is adapted to the mesh to learn could be beneficial. However, modifying our sampling scheme to sample a larger enveloppe around $\mathcal{O}$ did not seem to improve results, at least for the same architecture.

The difficulty of designing a one-size-fits-all sampling scheme stems from the fact that we know so little a priori about the visibility of the mesh. Not enforcing an equally balanced training set can make the accuracy measure highly unrelieable as it could become heavily biased towards one class and misrepresents the true distribution.[1] Figure 6.2 below shows the average class distribution for all scenes trained. Surprisingly enough, training on a 1:1 ratio of visible and nonvisible samples yielded a model that could not produce any shadow *at all*. This indicates that not altering the training distribution was crucial in controling the mode of the learned distribution and it could also explain why the model could not capture shadows from farther light sources. This is also reflected by the poor performance of the Cube scene, where the average

[1] For instance, a binary classifier achieving 0.90 accuracy on a heavily skewed test set with a 90% class A and 10% class B is essentially useless.

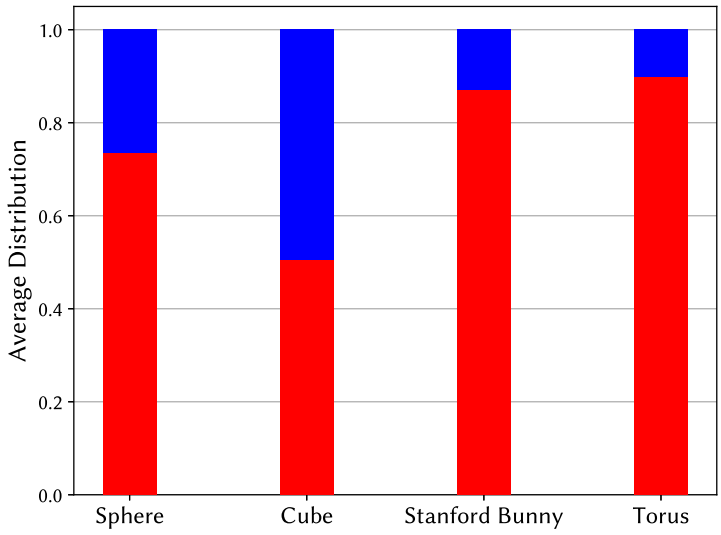


Figure 6.2. Average class distribution for all scenes over 30 epochs of dynamic visibility resampling. In blue, visible (1) and in red, nonvisible (0).

class distribution was the closest to a 1:1 ratio among all other scenes.

Moreover, training with and without dynamical resampling at each epoch did not produce better shadows overall. In fact, both methods stabilized early in the training and barely improved over time. This is an interesting observation, as one would expect better performance from a more diverse training set. Online resampling does decrease the generalization error of the model, but it is more likely that the shallowness of the network lead to a visibility approximator with high bias. In other words, the model was too simple to explain all discontinuities present in the visibility profile of the mesh and was only able to resolve some of them. Using online learning in regions where the model completely failed is one intuitive way of improving the visibility sampling scheme, but it is unclear if this would actually enhance the quality of the shadows.

All renders were obtained with an input representation in Euclidean coordinates in $\mathbf{R}^6$. Using spherical coordinates to decrease the number of input dimensions to $\mathbf{R}^5$ turned out to be problematic and often did not produce any shadow at all. This might be explained by the fact that using two different coordinate systems for positions and directions forced the network to learn the mapping from one space to another on top of learning the visibility distribution, which is already difficult for such a small net.

6.1.3. Choice of Minimizer. While the binary cross-entropy loss is a natural choice for classification problems, it is clearly not enough for target distributions as complex as visibility. The principal issue with maximizing log-likelihood in this scenario is that it does not use any geometrical feedback to guide the learning process, even though the visibility problem is intrinsically geometric. Gradients do not care how far off the prediction is in Euclidean space, but this information could possibly be exposed by adding an additional geometric penalty term. The issue with this approach is that this term needs to be differentiable for backpropagation, and it is not clear how to incorporate this measure efficiently. In fact, it is an open problem to design a trainable objective function that would integrate both the notions of information and geometry in a well-defined and differentiable manner. There are other loss functions that do no rely on a probabilistic interpretation, but in the context of classification they arguably make less sense.

6.1.4. Choice of Metric. Accuracy turned out to be poor metric to assess how well the model would perform in the presence of receivers. A near perfect accuracy on the validation set did not translate to better shadows by any mean; in fact the per-epoch rendered scenes proved that small increases in validation accuracy did not contribute to a lower error when compared to ray traced shadows. In other words, our choice of loss function did not correlate with overall quality of the final shading. An arguably improved metric would be to measure an expected structural error on random projection planes. Using the box projection test to measure this error could have undesired and pervasive effects, such as forcing the network to learn axis-aligned shadows. Using randomly oriented walls for projections would ensure a uniform coverage of the visibility profile, but this approach would require invoking the renderer on every training batch. While this is feasible under a unified framework, this introduces a large amount of overhead that is difficult to accommodate for separate training/inference platforms.

6.2 Other Limitations

6.2.1. Self-shadowing. One limitation of using a neural network for shadows is that it cannot resolve shadows casted directly onto the occluder. This is not an issue for convex shapes, but self-

shadowing can occur for concave objects. For instance, the ears of the bunny can cast a shadow on its back if the light is placed above the object. In our model, this was handled by reverting to ray tracing if a primary ray hit the identified occluder. It is possible to use precomputed visibility methods such as dynamic height fields [SN08] to capture this effect.

6.2.2. Static Occluders. Another important limitation of our model is that it assumes that the occluder is static and centered at the origin. However, it would be possible to change the occluder's position, provided we also modify how we use the neural network to determine visibility. This would require transforming the shape back in local frame, which can be costly if this needs to be done at every frame.

6.2.3. Evaluation Cost. Our implementation did not fully take advantage of parallelism since we applied the neural network at every shading point. Deep neural networks are particularly good at processing large amounts of data in batch, but in our tests we called our model whs times, where w, h are the width and height of the image plane, and s is the number of samples per pixel. As a result, our method was 5–10× slower than using a BVH for secondary rays. Feeding many inputs (say for each pixel) as tensors to the model could provide important speedups and could allow for more complex architectures to be used without impacting inference time.

6.3 Future Work

In hindsight, it is clear that our goal of learning arbitrary visibility profiles was far too ambitious. Artificial neural networks are powerful approximation tools but they also tend to oversmooth in blurry regions, which is problematic for shadows even when they are produced by area light sources. While it is true that we do not necessarily need shadows to be perfectly accurate to provide a sense of realism, small defects in the structure of the shadows are rapidly noticed even if they get blurred. Our experiments have shown that attempting to solve the visibility problem with a small capacity neural network simply seems to be unfeasible, even for simple shapes. By sampling a spherical volume around the occluder, we assumed that the light positions could be anywhere in this region. In practice, moving light sources are usually constrained into much

tighter regions (*e.g.*, the trajectory of the sun in the sky). For the neural network approach to be efficient, we would need to dramatically reduce the number of potential positions when sampling, but by doing so, we also heavily weakens the usefulness of our model.

Nonetheless, there are some interesting avenues for future research. Given visibility is a general problem that frequently appears in rendering, it would be worth investigating other problems that suffer from poor visibility approximations. Our experiments has demonstrated that training a neural network to estimate visibility can be done in only a handful of epochs and, as such, could be used as a preprocessing step to bootstrap other rendering algorithms. For instance, recent work in Markov chain Monte Carlo for rendering [OHHD18] has shown that mutating paths to avoid nonzero path contributions due to occlusion can significantly accelerate convergence. Another possibility is to extend our framework to cone tracing by estimating the average visibility. By sampling the visibility using small beams of directions instead of single rays, a different model could be learned. This approach is particularly intriguing as a network would probably have more facility learning visibility over a discretized set of directions than over the entire sphere of directions. Combining this approach with a voxelized scene representation in a divide-and-conquer manner could potentially lead to better visibility approximation overall. Finally, the idea of only using the network in high confidence zone could be explored more. By computing a segmentation mask of confidence zones produced by the network, it would be possible to invoke the model only in uncertain regions and ray traced the rest. The network would then only be used to resolve partially occluded regions to retain the structural fidelity of the shadows. This could be achieved by using Bayesian neural networks to quantify the incertainty for instance. This segmentation idea has recently proved to be successful in the context of adaptive temporal antialiasing [MSG+18] and is worth investigating further in the context of learned visibility.

6.4 Conclusion

In this thesis, we have investigated the possibility of using a deep neural network to approximate the visibility profile of simple objects for shadow generation. After reviewing the fundamentals of light transport and deep learning, we have presented offline and real-time techniques to estimate

shadow regions in virtual scenes. We then introduced a method for sampling visibility points around an object that correspond to potential light positions to evaluate. We have shown that using this data generating tool, we could dynamically create a binary training set of points and directions in ray space. We used this dataset to train a simple neural network classifier in a supervised manner to approximate the underlying visibility distribution of the object.

We tested our model on meshes of varying complexity (*e.g.*, convex, concave and nonzero genus objects) by replacing the visibility query with our neural network to generate learned shadows. Our results demonstrated that for a multilayer perceptron architecture, the visibility function is too complex to be learned. Our simple two-layer feedforward neural network failed at capturing all subtleties in the visibility landscape, but it was somehow able to resolve some umbra regions. Our model could not generalized to further light positions despite having been trained on such data, suggesting that the network capacity was too low. Out of the four meshes we tested, only one (the sphere) was successful at correctly approximating the shadows. Other mesh objects had varying artifacts that produced unrealistic shadow regions.

Our experiments indicate that neural networks are not magic black boxes that work flawlessly in all scenarios: their power is limited and they do not always behave as one would expect. With real-time ray tracing now a first-class citizen in modern rendering pipelines, we believe hybrid strategies combining rasterization and ray tracing is the next step for solving the interactive relighting problem. While bruteforce learning of visibility could not be done efficiently, there are many other problems in rendering that could benefit from supervised learning to improve performance. The complexity of these problems need to be taken into account and carefully studied in order to design intelligent algorithms that would be both fast and precise enough to justify replacing traditional rendering techniques. With the ever-growing effort of the AI community to make deep learning programming frameworks accessible to other fields of research, we anticipate that many ideas attempting to bridge learning and rendering algorithms will be explored in the near future.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[AMHH+18] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition.* A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[BLPL06] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, pages 153–160, Cambridge, MA, USA, 2006. MIT Press.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[CGKW18] Taco S. Cohen, Mario Geiger, Jonas Köhler, and Max Welling. Spherical CNNs. In *International Conference on Learning Representations*, 2018.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning*, 12:2121–2159, 2011.

[DK17] Ken Dahm and Alexander Keller. Machine learning and integral equations. *CoRR*, abs/1712.06115, 2017.

[DL06] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 161–165, New York, NY, USA, 2006. ACM.

[ESAW11] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. *Real-Time Shadows*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition, 2011.

[Fer05] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[HHM18] Eric Heitz, Stephen Hill, and Morgan McGuire. Combining analytic direct illumination and stochastic shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '18, pages 2:1–2:11, New York, NY, USA, 2018. ACM.

[Hil18] Sébastien Hillaire. Real-time ray tracing for interactive global illumination workflows in Frostbite. Game Developers Conference, march 2018.

[Hin12] Geoffrey E. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, July 1989.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034, Washington, DC, USA, 2015. IEEE Computer Society.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456. JMLR.org, 2015.

[JKRL09] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *ICCV*, pages 2146–2153. IEEE, 2009.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[Mik08] Miroslav Miksik. Generating soft shadows in real-time. Master's thesis, Czech Technical University in Prague, Prague, Czech Republic, May 2008.

[ML18] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018.

[MSG+18] Adam Marrs, Josef Spjut, Holger Gruen, Rahul Sathe, and Morgan McGuire. Adaptive temporal antialiasing. In *ACM SIGGRAPH / Eurographics High Performance Graphics*, page 4, August 2018.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[OHHD18] Hisanari Otsu, Johannes Hanika, Toshiya Hachisuka, and Carsten Dachsbacher. Geometry-aware Metropolis light transport. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)*, 37(6):278:1–278:11, 2018.

[PJH17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering, Third Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2017.

[PTC+17] Gabriel Pereyra, George Tucker, Jan Chorowski, Lukasz Kaiser, and Geoffrey E. Hinton. Regularizing neural networks by penalizing confident output distributions. *CoRR*, abs/1701.06548, 2017.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 283–291, New York, NY, USA, 1987. ACM.

[SHK+14] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[SMKL15] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik G. Learned-Miller. Multi-view convolutional neural networks for 3D shape recognition. In *Proc. ICCV*, 2015.

[SN08] John Snyder and Derek Nowrouzezahrai. Fast soft self-shadowing on dynamic height fields. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, pages 1275–1283, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[Vea98] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[Wal07] Ingo Wald. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978.

[YKL17] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. Technical report, 2017.

[ZJL+15] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and Sung-Eui Yoon. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports)*, 34(2):667–681, may 2015.