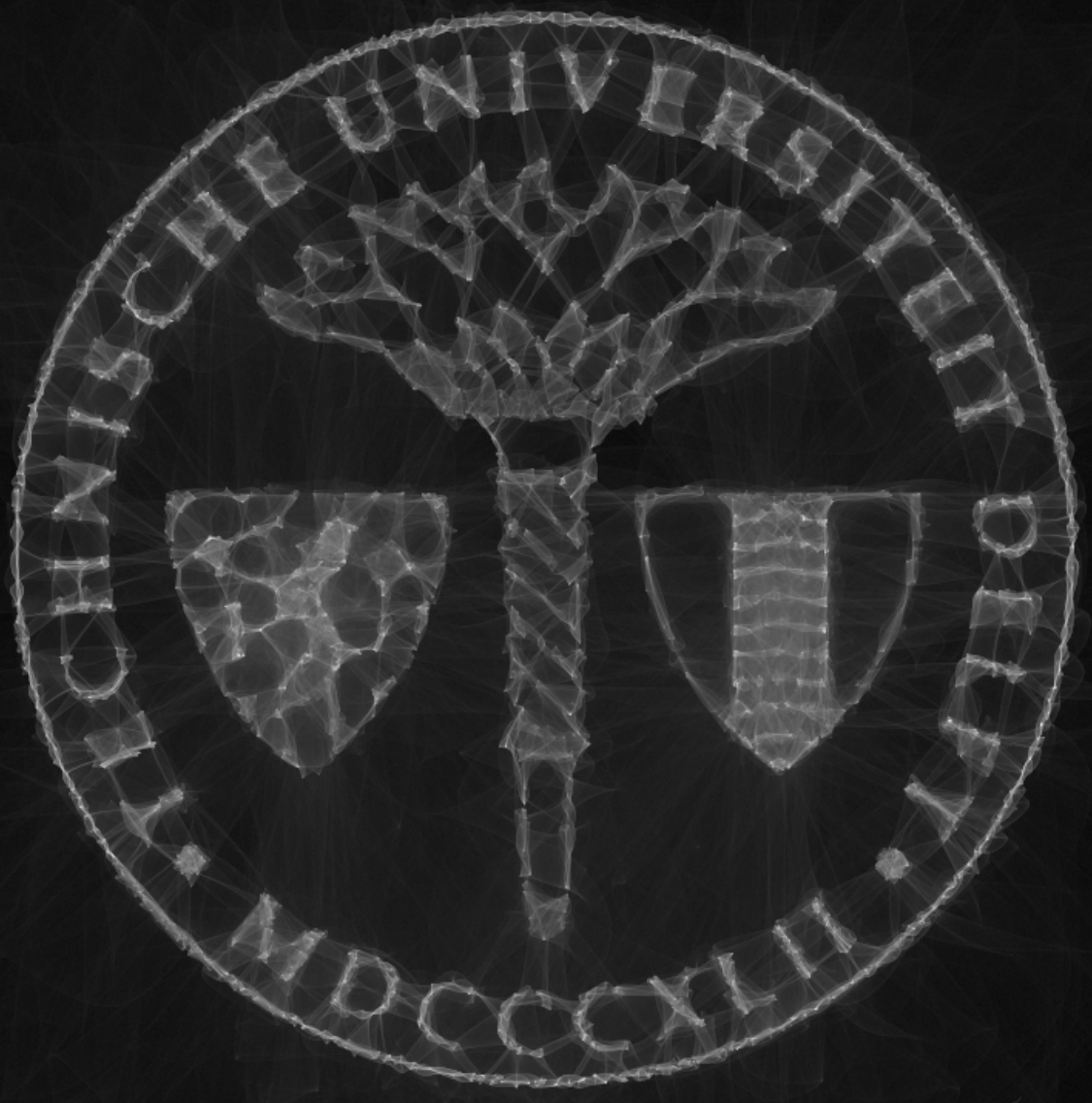
Designing free-form optics for multiple-source illumination using differentiable ray-tracing and neural networks

Bart de Koning



TU Delft

Designing free-form optics for multiple-source illumination using differentiable ray-tracing and neural networks

by

Bart de Koning

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday August 22, 2022 at 12:00 PM.

Student number:	4361040	
Project duration:	December 14, 2021 – August 22, 2022	
Thesis committee:	Dr. M. Möller,	TU Delft Numerical Analysis, supervisor
	Prof. Dr. ir. A.W. Heemink,	TU Delft Mathematical Physics
	Dr. A.J.L. Adam,	TU Delft Optics
	A.N.M. Heemels, MSc,	TU Delft Optics

TU Delft

Preface

This thesis describes the literature study and research performed by Bart de Koning as part of the final project of the Master Applied Mathematics, with specialisation Computational Science and Engineering, at the Delft University of Technology. The project is part of a collaboration between the Numerical Analysis and Optics groups. It offered a great opportunity to combine my interests in physics, mathematical (geometrical) modelling and 'making things that look nice', in a practical application.

The research started with the struggle to make use of Mitsuba 2, which turned out not to be ready for this project at the time. This incentivised me to implement a differentiable ray-tracer myself, which was a fun challenge, but also a lot of work. It certainly triggered my tendency to work things out as much as possible from first principles.

I also decided to code the NURBS implementation myself, as there was no good implementation available in an AD framework. This implementation is an improvement of the one by Lucas Crijns, who worked on caustic design with B-spline surfaces a few months before my project started. This project convinced me that geometric design with AD can be a very powerful tool, and I hope to work on this again in my upcoming career.

A special thanks goes out to my supervisors for this project: Dr. Matthias Möller helped me with insights in neural networks and B-splines/NURBS as well as giving very helpful feedback to polish the final draft of this thesis, Dr. Aurèle Adam shared his knowledge in optics, and PhD candidate Alex Heemels gave a lot of his time for specific guidance, as well as giving a lot of helpful feedback on this thesis. I also want to mention my fellow master students Marek Oerlemans and Dylan Everingham, who worked on master projects in optics design with AI and the same time I did. Their work gave me a more holistic picture of this field of research.

The project took place partly in a period of COVID regulations, which caused almost all meetings wit the supervisors and fellow master students to take place online. Despite this the supervisors have been able to provide very helpful guidance and feedback, so I thank them for helping me produce the document that lies before you.

Lastly I thank my parents and my sister, for sharing in my enthusiasm for this project despite often understanding very little of what I was talking about, and reminding me that there is more to the world than caustic design when I needed to hear it.

Bart de Koning
Rhoon, August 2022

Contents

Preface iii

Abstract vii

1 Introduction 1

2 Preliminary theory 3
2.1 Optics . . . 3
2.1.1 The nature of light . . . 3
2.1.2 From electromagnetic theory to geometrical optics . . . 3
2.1.3 Light-matter interactions . . . 4
2.1.4 Optical design . . . 7
2.1.5 Human perception of light . . . 8
2.2 Ray-tracing . . . 9
2.2.1 The basics of rendering using ray-tracing . . . 9
2.2.2 Differentiable ray-tracing . . . 11
2.3 Caustic design . . . 15
2.3.1 General . . . 15
2.3.2 Etendue . . . 16
2.3.3 Existing caustic design methods . . . 17
2.3.4 Caustic design methods with differentiable ray-tracing . . . 18
2.3.5 Previous results of neural networks in Caustic design . . . 18
2.4 B-splines & NURBS . . . 19
2.4.1 General . . . 19
2.4.2 B-spline and NURBS surfaces . . . 19
2.4.3 More generalized surfaces . . . 24
2.5 Physics-informed machine learning . . . 25
2.5.1 A brief history . . . 25
2.5.2 Neural network architectures . . . 25
2.5.3 Training/Learning and the loss function . . . 26
2.5.4 Machine learning in physics problems . . . 28

3 Analytical ray-tracing problem statement 31
3.1 General problem statement . . . 31
3.2 Derivation of the ray mapping for a plane wave sources . . . 32

4 The differentiable ray-tracer 33
4.1 The B-spline lens . . . 33
4.1.1 Lens definition . . . 33
4.1.2 Well-definedness . . . 34
4.2 Sources and sampling . . . 35
4.2.1 Plane wave . . . 35
4.2.2 Point source . . . 36
4.2.3 Ray sampling artefacts . . . 36
4.3 Ray-tracing . . . 37
4.3.1 Sequential . . . 37
4.3.2 B-spline intersection . . . 37
4.3.3 Intersection algorithm convergence . . . 42
4.4 Image reconstruction . . . 44
4.4.1 Gaussian reconstruction . . . 44
4.4.2 Differentiability . . . 46

5 The Optimization pipeline **47**
5.1 The neural network . . . 47
5.1.1 Architecture . . . 47
5.1.2 Control point freedom . . . 49
5.2 The loss function . . . 49
5.3 The complete pipeline . . . 50
5.4 Iterating . . . 50

6 Results **53**
6.1 Render derivatives with respect to a control point . . . 54
6.2 Sensitivity to initial state and neural network architecture . . . 55
6.2.1 Circular top hat distribution from plane wave . . . 55
6.2.2 TU flame and faceted ball from plane wave . . . 59
6.3 B-spline intersection algorithm . . . 63
6.4 Optimization with a point source and a grid of point sources . . . 64

7 Conclusion **67**
7.1 General . . . 67
7.2 Outlook . . . 67

A Efficient evaluation of the B-spline basis functions **71**

B Notes on the implementation **73**
B.1 LightTools_verification . . . 73
B.2 PINN based freeform design (branch: MEP_Geometric_Optics_Bart) . . . 73

C Computational graph **75**

D Simulation details **81**

E Mitsuba 2 **83**

Abstract

Differentiable ray-tracing is an exciting new development in computer graphics to approach all sorts of 3D scene design problems by obtaining gradients of renders produced by ray-tracing with respect to parameters that define the scene. These gradients can then be incorporated in a gradient-descent type optimization pipeline.

One such type of design problem is caustic design with free-form lenses. This is the process of obtaining the geometry of lenses in an optical system such that the light that passes trough this system forms a desired illumination distribution on a target screen. A typical application of this is distributing the light from streetlights or car headlights in a pleasing and efficient way over the street.

The non-imaging optics literature offers many methods to design free-form lenses for caustic design, but differentiable ray-tracing is largely unknown in this field. Therefore this thesis proposes a method of optimizing free-form lenses for caustic design with differentiable ray-tracing.

The optimization pipeline starts with a multi-layer perceptron neural network which outputs parameters that define one free-form lens side in the form of a B-spline surface. A specially implemented ray-tracer then produces a caustic render by tracing through this lens from either a plane wave or a (grid of) point source(s). Back-propagation and optimization takes place using automatic differentiation in PyTorch and the Adam optimizer.

The results, which are verified with LightTools, show great promise for this technique, especially the plane wave optimizations. The point source grid optimizations proved to be more challenging, but also here the optimization was able to achieve improvement. This shows that the proposed technique also has potential in positive etendue optimizations.

1 Introduction

Almost everyone has observed caustics, possibly without knowing the technical term; they are the light patterns that appear for instance on the bottom of a pool or as light travels trough a wine glass. These light patterns have several practical applications, such as in architecture, where a window casts a pleasant caustic pattern into a room [Kiser et al., 2021], or in security, where a hard-to-forge artefact can be 'read out' under normal daylight [Papas et al., 2011].

These applications call for *caustic design*; a collection of methods that solve the inverse problem of finding an optical system that produces the desired caustics. This thesis focusses on designing free-form lenses that generate particular caustics. The non-imaging optics literature offers variety of methods to design such free-form lenses [Papas et al., 2011; Schwartzburg et al., 2014; Yue et al., 2014]. This thesis explores the possibilities of performing caustic design using *differentiable ray-tracing*, a technique which is largely unmentioned in the field of optics design.

Differentiable ray-tracing

Ray-tracing is the process of computationally creating a 2-dimensional image (called a render) of a 3-dimensional digital scene, by discrediting the light transport trough the scene from sources to a detector into a finite set of rays, which are traced trough the scenes using (simplifications of) laws of optics.

Differentiable ray-tracing is an exciting new development in computer graphics to approach all sort of optimization problems of 3D scenes, by obtaining derivatives of a render with respect to scene parameters. One such class of optimization problems is free-form lens design for shaping a light beam to form a desired target caustic on a detector.

The strength of differentiable ray-tracing is its ability to be implemented in gradient-descent based optimization pipelines. A modern such pipeline structure is the *Physics Informed Neural Network* (PINN) [Raissi et al., 2019], where a neural network is trained to approximate the solution to a set of differential equations that form some physics model. This physics model however can also be implemented in other forms, for instance with differentiable ray-tracing in the case of an optics simulation, which brings us to the goal of this thesis.

Thesis goal

The goal of this thesis is to implement and test an optimization pipeline for caustic design with differentiable ray-tracing. The pipeline will consist of a neural network which outputs parameters that define a free-form lens surface in the of a B-spline surface. Ray-tracing will take place using implementations of B-spline surfaces and differentiable ray-tracing which will be developed specially for this thesis. Optimization will be governed by back-propagation of a suitable loss function and the Adam optimizer.

Outline

The thesis outline is as follows: chapter 2 treats the necessary preliminary theory on optics, ray-tracing, B-splines (and NURBS) and gradient-based optimization using automatic differentiation, and chapter 3 treats the analytical formulation of the lens optimization problem. Chapter 4 then explains the implemented differentiable ray-tracer, and chapter 5 explains the implemented optimization pipeline as a whole. Chapter 6 shows and discusses a selection of optimization results and chapter 7 concludes the thesis.

2

Preliminary theory

This chapter is a condensation of the literature study performed before executing the main research within the subsequent chapters of this thesis.

2.1. Optics

This section discusses the properties of light relevant to caustic design.

2.1.1. The nature of light

The nature of light has been debated heavily over the past centuries, from the particle perspective of Newton (1643-1727) and the wave perspective of Huygens (1629-1695) to the theory of electromagnetism by Maxwell (1831-1879) and the quantum theory pioneered in the beginning of the twentieth century [Fowles, 1975, ch.1]. As to the "true" nature of light, Fowles put it very succinctly:

> *Since electromagnetic theory and quantum theory also explain many other physical phenomena in addition to those related to electromagnetic radiation, it can be fairly assumed that the nature of light is well understood, at least within the context of a mathematical framework that accurately accounts for present experimental observations. The question as to the "true" or "ultimate" nature of light, although as yet unanswered, is quite irrelevant to our study of optics* [Fowles, 1975, ch.1].

In other words: optics makes use of models of light that are probably emergent from a "most fundamental" theory of light, but these models are sufficient for the goals of optics.

2.1.2. From electromagnetic theory to geometrical optics

There are two main ways to model light: *wave optics* and *geometrical optics*. The former models light as a wave as following from Maxwell's equations (see below), and the latter describes light in terms of a large collection of rays, essentially one-dimensional curves trough space. The Maxwell Equations describe light as a wave, but with these equations as starting point the validity of the geometrical model can be mathematically derived.

We consider an electric field $\mathbf{E} = \mathbf{E}(\mathbf{x}, t)$ and a magnetic field $\mathbf{H} = \mathbf{H}(\mathbf{x}, t)$, depending on space $\mathbf{x} \in \mathbb{R}^3$ and time $t \in \mathbb{R}$. Assuming the absence of electric charges and currents, the differential form of Maxwell's equations states the following relationships:

$$\nabla \cdot \mathbf{E} = 0, \tag{2.1.1a}$$

$$\nabla \cdot \mathbf{H} = 0, \tag{2.1.1b}$$

$$\nabla \times \mathbf{E} = -\mu \frac{\partial \mathbf{H}}{\partial t}, \tag{2.1.1c}$$

$$\nabla \times \mathbf{H} = \epsilon \frac{\partial \mathbf{E}}{\partial t}. \tag{2.1.1d}$$

Here ϵ is the electric permittivity and μ is the magnetic permeability of a medium. These equations can be rewritten as[1]

$$\frac{\partial^2 \mathbf{E}}{\partial t^2} = \frac{1}{\mu\varepsilon}\nabla^2 \mathbf{E}, \tag{2.1.2a}$$

$$\frac{\partial^2 \mathbf{H}}{\partial t^2} = \frac{1}{\mu\varepsilon}\nabla^2 \mathbf{H}. \tag{2.1.2b}$$

Here we recognize wave equations with wave velocity $v = 1/\sqrt{\mu\varepsilon}$. In vacuum this yields the speed of light $c = 1/\sqrt{\mu_0\varepsilon_0} \approx 3.00 \times 10^{-8}\,\mathrm{m/s}$, where μ_0 and ε_0 are the electric permittivity and the magnetic permeability of the vacuum, respectively. The *refractive index* of a medium is the ratio of the speed of light in a vacuum and the speed of light in the medium:

$$n := \frac{c}{v} = c\sqrt{\mu\varepsilon}. \tag{2.1.3}$$

In general it holds that the higher the mass density of a material is, the higher its refractive index is. The refractive index is also wavelength-dependent, since ε is wavelength-dependent.[2]

The electromagnetic waves described by eqs. (2.1.2a) and (2.1.2b) carry energy. This amount of energy also determines the wavelength λ of the waves, see section 2.2.1. The energy density U of such waves is defined as

$$U := \tfrac{1}{2}(\varepsilon|\mathbf{E}|^2 + \mu|\mathbf{H}|^2). \tag{2.1.4}$$

From Maxwell's equations also the following conservation law can be derived:

$$\nabla \cdot (\mathbf{E} \times \mathbf{H}) + \frac{1}{2}\frac{\partial}{\partial t}(\varepsilon|\mathbf{E}|^2 + \mu|\mathbf{H}|^2) = 0. \tag{2.1.5}$$

This law can succinctly be written as

$$\frac{\partial U}{\partial t} + \nabla \cdot \mathbf{S} = 0, \tag{2.1.6}$$

where $\mathbf{S} := \mathbf{E} \times \mathbf{H}$ is the Poynting vector named after John Henry Poynting (1852-1914). The above equation indicates that the energy of an electromagnetic wave is carried in the direction of the Poynting vector and thus that $\mathbf{S}$ can be interpreted as the energy flux. The source of electromagnetic radiation is always a non-uniformly moving charge, for example the ions in the suns plasma.

Formally geometrical optics can be defined as the limit of Maxwell's equations in terms of the electromagnetic wave frequency going to infinity. In this context a *light ray* is a trajectory through space in the direction of the Poynting vector, indicating the propagation of electromagnetic energy [Romijn, 2021, sec 2.1]. A ray also has a *ray weight* associated with it, denoting the power carried by that ray.

2.1.3. Light-matter interactions

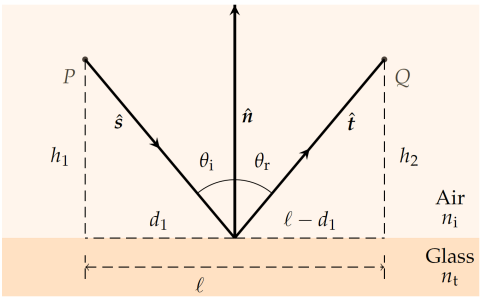
For the proper modelling of optical phenomena, various interaction types between light and matter need to be taken into account. For instance, at an optical interface between two media discontinuities in the refractive index n occur. When a ray hits a surface that acts as such an interface, some light is scattered backwards which is called *reflection*, and some light continues into the new medium, which is called *refraction*.

We only consider *specular* reflection and refraction, meaning that bundles of rays that get reflected or refracted stay in a bundle. This is in opposition to diffuse reflection or refraction, where a rough surface reflects or refracts a bundle into many different directions. The *Fresnel equations* are also discussed, which model which portion of light is reflected and which portion is refracted in the case that refraction occurs.

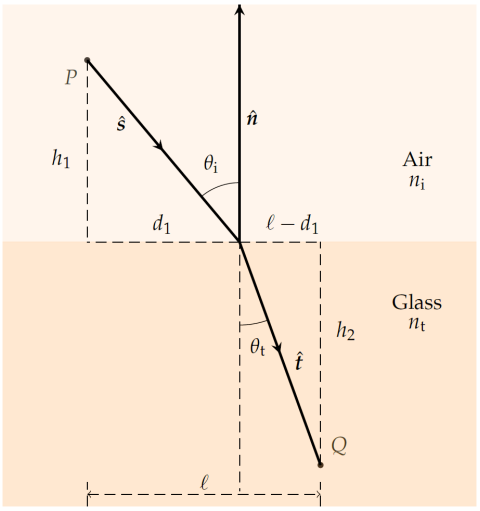
Both reflection and refraction can be derived from the *Principle of Fermat*, which states that a light ray is a curve on which the travel time is minimal. For these derivations see Romijn, 2021, sec. 2.4-2.5.

[1] for the full derivation see Romijn, 2021, sec. 2.1.

[2] Often a single value for the refractive index of a material is reported, which corresponds to a wavelength of $533\,\mathrm{nm}$; the wavelength of a helium-neon laser.



(a) An illustration of the law of reflection. Figure figure taken from [Romijn, 2021, fig. 2.4].



(b) An illustration of the law of refraction (Snell's law). Figure taken from [Romijn, 2021, fig. 2.6].

Figure 2.1.1: Light matter interactions: reflection (left) and refraction (right).

Reflection

The relationship between the direction of a ray before and after specular reflection is given by the *law of reflection*, which states that the angle of incidence θ_i is equal to the angle of refraction θ_r, as seen in fig. 2.1.1a. As a formula this is simply

$$\theta_i = \theta_r. \tag{2.1.7}$$

Following fig. 2.1.1a, if we let $\hat{\mathbf{s}}$ be a unit vector in the direction of the incoming ray, $\hat{\mathbf{n}}$ be the unit normal vector to the surface, and $\hat{\mathbf{t}}$ be the unit vector in the direction of the reflected ray, then

$$\hat{\mathbf{t}} = \hat{\mathbf{s}} - 2\langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle \hat{\mathbf{n}} \tag{2.1.8}$$

is the vectorial form of the law of reflection [Romijn, 2021, sec. 2.4], with $\langle \cdot, \cdot \rangle$ denoting the standard inner product between any two vectors in $\mathbb{R}^3$.

Refraction

On a (sub-)atomic level, an electromagnetic wave that passes trough a medium causes particles in that medium such as electrons to oscillate. Due to this oscillation these particles emit their own electromagnetic waves, which interact with the original wave. The resulting wave travels at a speed slower than the (vacuum) speed of light, as discussed in section 2.1.2. At the macroscopic level this becomes visible by the abrupt change of a light ray's direction when it passes through a boundary between media of different refractive indices. When the refractive index varies continuously throughout a medium, this results in a smoothly bending ray path (see section 2.3.1).

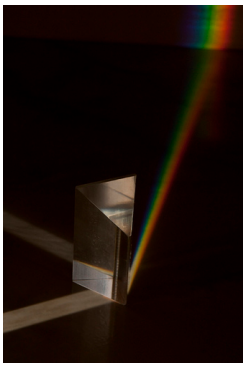
As also mentioned in section 2.1.2, the refractive index of a medium varies with the wavelength of light, which is another effect of the interaction between electromagnetic waves and the electrons in a medium. This effect is called *dispersion*, and a prime example of this is how a prism splits white light that consists of a spectrum of wavelengths by bending each wavelength by a different amount, as seen in fig. 2.1.2a.

The bending of a ray at an optical surface between two media of different refractive indices is given by *Snell's law*. In terms of angles this law reads

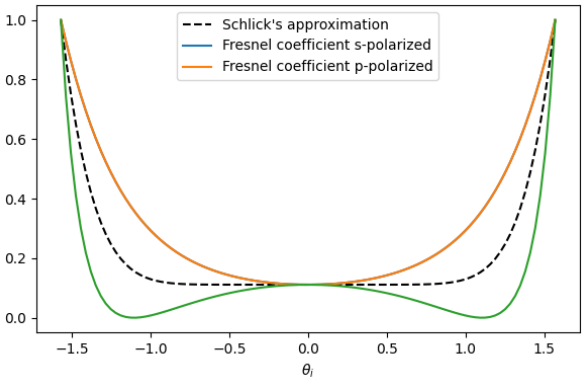
$$n_i \sin\theta_i = n_t \sin\theta_t, \tag{2.1.9}$$

and its geometrical meaning is indicated in fig. 2.1.1b. Here n_i and n_t are the refractive indices of the materials on the incident and transmittance side respectively, and θ_t is the angle of the refracted ray to the surface normal $\hat{\mathbf{n}}$. In vectorial form this is given by

$$\hat{\mathbf{t}} = r\hat{\mathbf{s}} - \left(r\langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle + \sqrt{1 - r^2(1 - \langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle^2)} \right) \hat{\mathbf{n}}, \tag{2.1.10}$$



(a) Dispersion of white light by a prism. Figure taken from https://www.real-world-physics-problems.com/prism.html.



(b) Reflection coefficient dependent on the incident angle θ_i for $r = 0.5$, given by the Fresnel equations and Schlick's approximation.

Figure 2.1.2: Dispersion example and reflection coefficients.

where $r := \frac{n_i}{n_t}$. Eq. 2.1.10 can be derived from eq. (2.1.9) by using

$$\cos\theta_i = -\langle \hat{\mathbf{n}}, \hat{\mathbf{s}} \rangle, \quad \cos\theta_t = -\langle \hat{\mathbf{n}}, \hat{\mathbf{t}} \rangle. \tag{2.1.11}$$

Note that this law yields a physical impossibility when $r^2(1-\langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle^2) > 1$, since we can only interpret the result if $\hat{\mathbf{t}}$ is real-valued. This 'problem' only appears when $r > 1$, so when light is going from a medium with a higher refractive index to a medium with a lower refractive index. The *critical angle*, the maximum incidence angle at which refraction occurs, is then given by

$$r^2(1 - \langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle^2) = 1 \quad \Rightarrow \quad \cos\theta_c = \sqrt{1 - \frac{1}{r^2}} \quad \Rightarrow \quad \sin\theta_c = \frac{1}{r}, \tag{2.1.12}$$

where we use that $\cos\theta_i = -\langle \hat{\mathbf{n}}, \hat{\mathbf{s}} \rangle$. When $\theta_i > \theta_c$ we speak of *total internal reflection* (TIR) [Romijn, 2021, sec. 2.5].

The Fresnel equations

The Fresnel equations describe, in the case of refraction, which portion of the energy of the incoming light is reflected instead of refracted. This depends on the *polarisation* of the light, a property that describes the orientation of the electric and magnetic waves that make up the light. The Fresnel reflectance equations make a distinction between s-polarized and p-polarized light [Fowles, 1975, sec. 2.7.1]:

$$R_s = \left| \frac{\cos\theta_i - r\cos\theta_t}{\cos\theta_i + r\cos\theta_t} \right|^2, \tag{2.1.13a}$$

$$R_p = \left| \frac{\cos\theta_t - r\cos\theta_i}{\cos\theta_t + r\cos\theta_i} \right|^2. \tag{2.1.13b}$$

Fig. 2.1.2b shows the reflection coefficients for both polarisation types for the case $r = 0.5$. It also shows Schlick's approximation [Schlick, 1994, eq. 24] of these coefficients, an approximation often used in computer graphics:

$$R = R_0 + (1 - R_0)(1 - \cos\theta_i)^5, \quad R_0 = \left(\frac{r-1}{r+1} \right)^2. \tag{2.1.14}$$

This approximation is useful for simplified optics simulations that do not take polarization into account. The ray-tracer implemented for our experiments also uses Schlick's approximation, as discussed in section 4.3.2.

Gradient index

The discussions of specular reflection and refraction above only describe the behaviour of rays of light when they intersect a surface of discontinuity in the refractive index in space, and assume that the refractive index is homogeneous outside this surface where the ray goes in a straight line. There are however also materials with a gradient refractive index field through space: $n = n(\mathbf{x})$. Light traveling trough such a material effectively undergoes continuous refraction, where the path of the ray $\mathbf{r} = \mathbf{r}(t)$ is given by the Eikonal equation expressed as a second order ODE, relating the change in position (direction) of the light ray to the refractive index gradient [Sharma et al., 1982]:

$$\frac{\mathrm{d}^2\mathbf{r}(t)}{\mathrm{d}t^2} = n(\mathbf{r}(t))\nabla n(\mathbf{r}(t)) \tag{2.1.15}$$

Optimization of gradient index optics is not further mentioned in this thesis, but could be an interesting future direction of research.

2.1.4. Optical design

Optical design is a vast area of research. In general the goal is to create a system, consisting of mirrors and lenses and other optical components, that transform incident light (or more generally, electromagnetic radiation) in a desired way, for instance in the human eye (section 2.1.5). We distinguish two types of application areas of optical design:

- *imaging optics*: this involves optical elements that allow humans and machines to see, like in the human eye and cameras. It consists of transforming light that leaves an object at a point back to a point on the detector. This will not be discussed further as the subject matter of this report falls in the second category.
- *non-imaging optics* or *caustic design*, discussed in section 2.3.1. The goal of non-imaging optics is to transform the incident illumination from light sources into a desired target distribution [Meyron et al., 2018].

Light sources: point sources and Light Emitting Diodes

Non-imaging optical design requires the modelling of light sources. One of the simplest light sources to model is the *point source*, which emits light from a single point in space in all directions equally.

Point sources, however, do not have a real physical equivalent. One light source that can easily be inserted into an electrical circuit and is used often in industries is the *light emitting diode* (LED). A LED consists of a semiconductor that emits light and heat when a voltage is applied to it. The color of this light depends on the semiconductor material and the resultant energy strength [Romijn, 2021, sec. 1.1]. Every real world source has some extend, *i.e.* some surface area from which light is emitted. An extended source however can be approximated by a collection of point sources on its surface.

2.1.5. Human perception of light

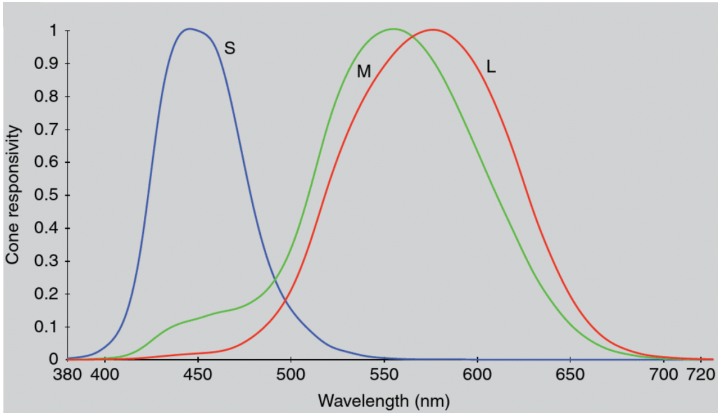
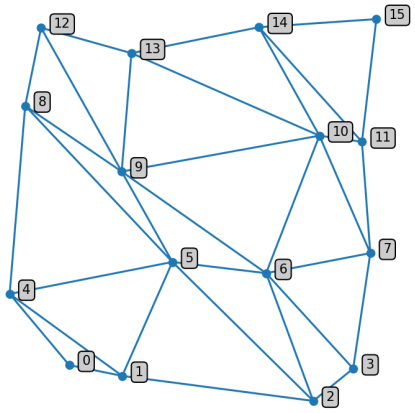


Figure 2.1.3: The responsivity of the different types of cones in the human eye as a function of wavelength. The three types of cone are labeled S, M, L for small, medium and large wavelengths respectively. Figure taken from [Fairchild, 2013, fig. 1.4a.]

The human eye is an interesting optical system on its own, but it is mainly discussed here because it's function could inform efficient evaluation of the quality of caustics for visual purposes. In this case it is no use trying to get rid of imperfections in a caustic if they are not perceptible by the naked eye.

The part of the eye that is sensitive to light is called the cornea, and it contains various types of light-sensitive cells. The cells that are most sensitive to light are called *rods*, and the cells that are used to perceive colors are called cones, of which most people have three different types. These different types have differing sensitivities to the parts of the electromagnetic spectrum, as indicated in fig. 2.1.3. These cones are labeled S, M, L, refering to the (relatively) small, medium and large wavelengths that the cone is sensitive to. As indicated in the figure these cones very roughly correspond to the colors blue, green and red, but the perception of color is a complicated process following from the neural processing of the cone signals [Fernandez-Maloigne and Robert-Inacio, 2013, sec. 1.1].

The *Stevens effect* is also very relevant here; this effect says that perceived contrast increases with brightness, so contrast does not simply depend on relative brightness [Fairchild, 2013, p. xxi].



(a) Triangle mesh with vertex numbers.

faces	
(0, 1, 4)	(4, 5, 8)
(8, 9, 12)	(1, 2, 5)
(5, 6, 9)	(9, 10, 13)
(2, 3, 6)	(6, 7, 10)
(10, 11, 14)	(1, 5, 4)
(5, 9, 8)	(9, 13, 12)
(2, 6, 5)	(6, 10, 9)
(10, 14, 13)	(3, 7, 6)
(7, 11, 10)	(11, 15, 14)

(b) Triangle faces defined by vertex index triplets.

Figure 2.2.1: Example of a triangle mesh in 2 dimensions.

2.2. Ray-tracing

Rendering is the computational process of producing a 2-dimensional image from a description of a 3-dimensional scene. Most photo-realistic rendering systems are based on ray-tracing, which is in its algebraic essence a very simple algorithm; it involves computing the path of rays of light trough a scene as it interacts with objects in a scene in various ways. The following sections give some algebraic and computational background to ray-tracing and differentiable ray-tracing.

2.2.1. The basics of rendering using ray-tracing

A ray-tracer consists at least of the following components:

Cameras or detectors The camera records an image of the scene on a sensor with provided size and pixel density.

Ray-object intersections To trace a ray it must be computed accurately where a ray intersects a given geometrical object. Furthermore, at this point the object's surface normal often has to be computed (as explained in section 2.1.3).

The way such an intersection is computed depends on the data structure with which the geometrical object is stored. For simple geometries, like a sphere, intersections are easily derived analytically. Let us look at the intersection of a ray and a sphere of radius $\rho > 0$ and center $\mathbf{c} \in \mathbb{R}^3$. The ray is given by the line segment $\mathbf{r}(t) = \mathbf{o} + \hat{\mathbf{d}}t, t \geq 0$, with an origin point $\mathbf{o}$ and direction $\hat{\mathbf{d}} \in \mathbb{R}^3$ such that $\left\|\hat{\mathbf{d}}\right\| = 1$. Then the intersection can be derived as follows:

$$\begin{cases} \mathbf{r}(t) = \mathbf{o} + \hat{\mathbf{d}}t, \ \|\hat{\mathbf{d}}\| = 1 \\ \|\mathbf{r} - \mathbf{c}\|^2 = \rho^2 \end{cases} \quad \Rightarrow \quad t^2 + \langle \hat{\mathbf{d}}, \mathbf{o} - \mathbf{c} \rangle t + \|\mathbf{o} - \mathbf{c}\|^2 - \rho^2 = 0$$

$$\Rightarrow \quad t = \frac{1}{2}\langle \hat{\mathbf{d}}, \mathbf{c} - \mathbf{o} \rangle \pm \frac{1}{2}\sqrt{\langle \hat{\mathbf{d}}, \mathbf{c} - \mathbf{o} \rangle^2 - 4\left(\|\mathbf{c} - \mathbf{o}\|^2 - \rho^2\right)},$$

where the smallest real and positive solution t yields the intersection, if such a solution exists. If no such solution exists, the ray does not intersect with the sphere.

For more complex geometries a *triangle mesh* is often used. This consists of an ordered list of point coordinates and an ordered list of triplets of point indices that each defines a triangle. Triangles do not intersect, but can share vertices or edges. Often a triangle mesh is a piece-wise linear approximation of a more smooth surface.

An example triangle mesh of this is shown in fig. 2.2.1. Note that the faces are all defined with the vertices in counter-clockwise order. This is important since this ordering defines the orientation

of the triangle,[3] for instance defining what is the inside and the outside of the mesh in 3D.

For a detailed description of a ray-triangle intersection algorithm from the literature see Pharr et al., 2017, section 3.6.2. For our experiments a triangle mesh intersection algorithm is implemented as well, which is covered in section 4.3.2.

Light sources/emitters A ray-tracer models the distribution of light throughout a scene, based on the source location and their energy distribution over the available emission directions.

There are many different models for the different sources of light in the real world. The spectrum of wavelengths emitted can go from very simple (monochromatic or a setup that is not 'wavelength-aware' at all) to for instance *Planck's law* for black body radiation [Pharr et al., 2017, section 12.1.1], which prescribes the radiance of an object at a certain wavelength as a function of its temperature.

The type of emitters we discuss here are *point* [Pharr et al., 2017, section 12.3] and *area* [Pharr et al., 2017, section 12.5] sources. A point source shines in all directions uniformly from a single point in space. An area source is a surface that shines light in all directions pointing to the outside of the surface (given an orienting normal field on the surface). Of particular interest is the directional area emitter, which only emits light perpendicular to the surface. For a flat surface we call this emission a *plane wave*, which can be approximated by a far away point source.

Surface scattering Each surface is provided with information about its appearance, which describes how light interacts with the surface (section 2.2.1). This is also called *material* information. This is either uniform on the surface, or can be specified locally using *texture coordinates* [Pharr et al., 2017, section 10.2].

An important aspect of rendering scenes with the help of ray-tracing is the choice of which rays to trace. Many ray-tracing algorithms use *ray-casting*: they trace rays from the camera to a light source in stead of the other way around, since many rays leaving a light source never hit a camera or detector.

The rays to be traced are sampled from some probability distribution. This process is vulnerable to *Monte-Carlo noise*, the sort of grain you see for instance in fig. 2.2.5b when the number of traced rays is relatively low. These problems can be tackled by choosing a good probability distribution and a sufficient amounts of rays to trace, *i.e.* an amount of rays for which the Monte-Carlo noise is suppressed to an acceptable level, where in general the amount of noise is proportional to $\frac{1}{\sqrt{N_{\text{rays}}}}$ [Pharr et al., 2017, sec. 13].

Physics derived quantities

A good way to quantify the transport of light though a scene is by keeping track of (the conservation of) energy. The smallest unit of light is the *photon*, carrying a wavelength λ-dependent amount of energy

$$Q_\gamma = \frac{hc}{\lambda}, \tag{2.2.1}$$

where h is *Planck's constant* and c is the speed of light.

The rate at which a source emits energy in the form of light can then be described as the time derivative of the radiant energy, called the *radiant flux* or *power*:

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}. \tag{2.2.2}$$

The energy flow can also further be specified as:

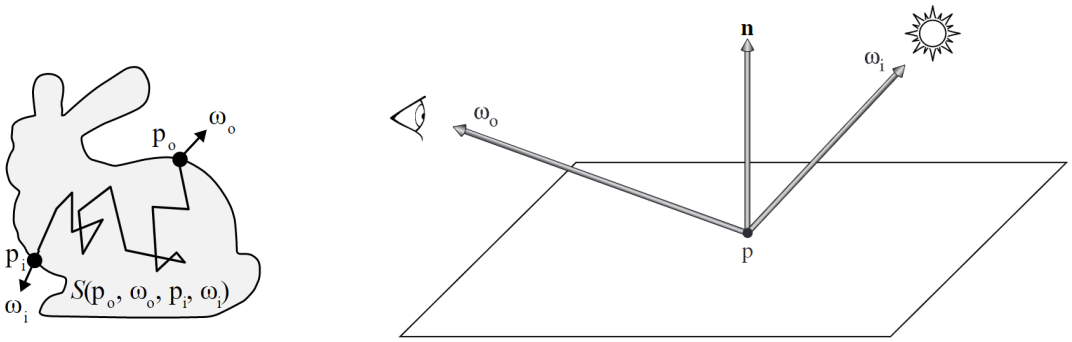
Irradiance E is the flux per unit area, dependent on location:

$$E(x, y) = \frac{\mathrm{d}\Phi}{\mathrm{d}A}. \tag{2.2.3}$$

For instance if a point source has a power of Φ and it is at the center of a sphere of radius r, then on that sphere the uniform irradiance is

$$E = \frac{\Phi}{4\pi r^2}.$$

[3] For instance by the right hand rule using two edges of the triangle. Say a triangle consists of the points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$. Then $\mathbf{x}_1 - \mathbf{x}_2$ and $\mathbf{x}_1 - \mathbf{x}_3$ denote two of the edges of the triangle, and $(\mathbf{x}_1 - \mathbf{x}_2) \times (\mathbf{x}_1 - \mathbf{x}_3)$ is a normal vector to the triangle.



(a) Schematic of a light-ray undergoing sub-surface scattering. Figure taken from [Pharr et al., 2017, fig. 15.6].

(b) An example of the input of the BSDF function. Figure taken from [Pharr et al., 2017, fig. 1.6].

Figure 2.2.2: Schematics of types of ray-tracing computations.

Intensity I denotes power per unit solid angle or *steradian* Ω, dependent on direction (here in spherical coordinates):

$$I(\theta, \phi) = \frac{\mathrm{d}E}{\mathrm{d}\Omega}. \tag{2.2.4}$$

For the same situation as above we have a uniform intensity of [Pharr et al., 2017, section 10.2]

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

Bidirectional scattering distribution functions

In a 3D rendering scene each object is provided with a *material*; information on how light interacts with the object and thus how it appears in a render. Such models vary greatly in complexity, from only reflection to *sub-surface scattering*, a complex process that enables for instance realistic human skin rendering. A schematic depiction of sub-surface scattering is shown in fig. 2.2.2a. In more complex models there can also be a dependence on wavelength and sometimes even on polarisation of the light.

What we are interested in here for lens modelling is the *bi-directional scattering distribution function* (BSDF): a function $f(p, \omega_o, \omega_i)$ that describes how much light reflects or refracts at point p in the direction of ω_o when incident light hits p from direction $-\omega_i$, See fig. 2.2.2b. If the function is indeed dependent on p it is called *spatially varying* (SVBSDF).

For the lenses modelled in our research we will assume *specular* reflection and transmission [Pharr et al., 2017, sec. 8.2]. This assumes that a surface is smooth, and thus light interactions are governed purely by reflection and refraction as discussed in section 2.1.3.

BSDF's are not an explicit part of the ray-tracer implemented for our experiment, but could be part of a future version of this implementation.

2.2.2. Differentiable ray-tracing

In order to understand the specific topic of differentiable ray-tracing, first the broader concept of automatic differentiation will be discussed.

Automatic differentiation

Gradient-descent type optimization methods (discussed in section 2.5.3) like the ones involving differentiable ray-tracing often require derivatives of highly composite functions[4] $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$. Hard-coding the computation of these derivatives is laborious and thus error-prone. Luckily several other methods exist to compute these derivatives.

One such class of methods is called *finite difference* [Vuik et al., 2017], based on approximating derivatives by using Taylor expansions. These methods however require multiple calls to $\mathbf{F}$ and thus are computationally inefficient, and always introduce a truncation error.

[4] functions that consist of many more elementary operations like (e.g. $+, \times, \sin, \exp$), such as neural networks (discussed in section 2.5.2).

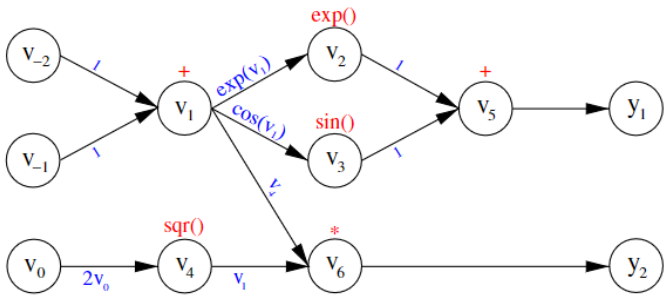


Figure 2.2.3: The computational graph of the function $\mathbf{F}(x_1, x_2, x_3) = (\exp(x_1 + x_2) + \sin(x_1 + x_2), \sqrt{x_3}(x_1 + x_2))$. v_{-1}, v_0, v_{-2} denote the input variables x_1, x_2, x_3 respectively, $v_1, \ldots v_6$ denote intermediate variables, and y_1, y_2 denote the outputs. In red the operation on the inputs of the node are shown, and in blue the partial derivatives are shown (of the right side of the arrow with respect to the left side of the arrow). Figure taken from [Gebremedhin and Walther, 2020, fig. 2].

As a good alternative *Automatic differentiation* (AD) or *algorithmic differentiation* [Gebremedhin and Walther, 2020] has been developed. The main idea of AD is to make systematic use of the chain rule of calculus for the various component operations of $\mathbf{F}$. This incentivizes to compose $\mathbf{F}$ out of simple functions for which the derivative is cheap to compute. Popular machine-learning frameworks like TensorFlow [Abadi et al., 2005] and Torch [Collobert et al., 2002] all make use of AD.
Automatic differentiation can be divided into two different modes:

- *Forward mode*: here one column of the Jacobian $J\mathbf{F}(\mathbf{x})$ is computed (corresponding to a single input of $\mathbf{F}$ and all outputs);

- *Reverse mode* (also known as *back-propagation* in the context of neural networks): here one row of the Jacobian is computed (corresponding to all inputs of $\mathbf{F}$ and a single output).

Note that for minimising a loss function in an optimization procedure, the output loss yields $m = 1$ and the number of inputs n is potentially very large, so reverse mode is the optimal choice. This is why software packages like PyTorch and Tensorflow, optimized for machine learning, use reverse mode automatic differentiation.

A *computational graph* is often used as a representation of how a function is built up from its component elementary functions. Such a graph can also be constructed in the background of computations and traversed when a certain gradient is requested. An example graph is shown in fig. 2.2.3 for the function $\mathbf{F}(x_1, x_2, x_3) = (\exp(x_1 + x_2) + \sin(x_1 + x_2), x_3^2(x_1 + x_2))$ that maps from $\mathbb{R}^3$ into $\mathbb{R}^2$.

There are two main ways to implement automatic differentiation:

- *Source transformation*: the automatic differentiation algorithm takes the source code for a computation as input and outputs the code transformed in such a way that it produces the derivatives of the computation instead, for instance rewriting `sq(x)` to `2*x` [Gebremedhin and Walther, 2020];

- *Operator overloading*: This refers to a general concept in programming where one (re-)defines the behaviour of an operator, like $+, -, \times$ for a certain data-type. To apply this to automatic differentiation, the idea is to create a two-valued number data object containing both a value and a derivative value, and overload the operators in such a way that if they are called with this new number object both the new value and the new derivative value are computed [Gebremedhin and Walther, 2020].

Both PyTorch and TensorFlow use operator overloading [Gebremedhin and Walther, 2020].

Differentiable ray-tracing
As stated in section 2.2.1, rendering in computer graphics constitutes generating images of 3D scenes defined by geometry, materials, light sources and camera properties. This is a complicated process

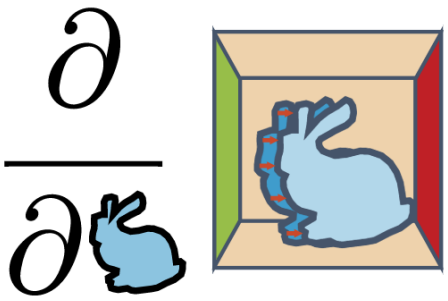


Figure 2.2.4: Simple schematic representation of a geometry derivative. Figure taken from [Jakob et al., 2019, Getting started].

whose differentiation is not uniquely defined[5], preventing a straight forward computation of gradients. Chapter 4 explains the differentiable ray-tracer implemented for our experiments.

Differentiable rendering (DR) refers to a family of techniques that make such integration possible by obtaining gradients of the render with respect to scene parameters, see for instance fig. 2.2.4. By differentiating the rendering, DR bridges the gap between processing methods in 2D and 3D, allowing (possibly with the help of neural networks) to optimize 3D entities (*e.g.* their vertex positions or underlying parameters) while operating on 2D projections (*i.e.* renders). Back-propagation of the gradients of some render-based loss function with respect to the scene parameters makes this optimization possible. Self-supervision pipelines are often constructed this way by computing a loss that compares the rendering output to some input image in various ways. The applications of this are broad, including 3D object reconstruction [Yan et al., 2016], hand pose estimation [Pavlakos et al., 2018] and face reconstruction [Genova et al., 2018]. A simple application is shown in fig. 2.2.5.

There are many different scene parameters which can be differentiated, but for the current application we will only focus on differentiation with respect to parameters that define geometry. How the differentiation with respect to these parameters occurs depends on the geometry data representation used by the rendering software.

The rendering software *Mitsuba 2* [Nimier-David et al., 2019] is discussed here, as using this was the original plan for this project and it might be used in future research. Mitsuba 2 supports the following geometry data representations:

- *Triangle mesh* (section 2.2.1);
- *Custom*: This renderer also supports integrating your own geometry data representation. In principle an implementation of the parametric surfaces $\mathbf{S} : [0,1]^2 \to \mathbb{R}^3$ as described in section 2.4.2 could serve as such a custom integration, provided with a ray-surface intersection algorithm and surface normal calculation:
$$\hat{\mathbf{n}}(u,v) = \frac{\mathbf{S}_u(u,v) \times \mathbf{S}_v(u,v)}{\|\mathbf{S}_u(u,v) \times \mathbf{S}_v(u,v)\|}.$$

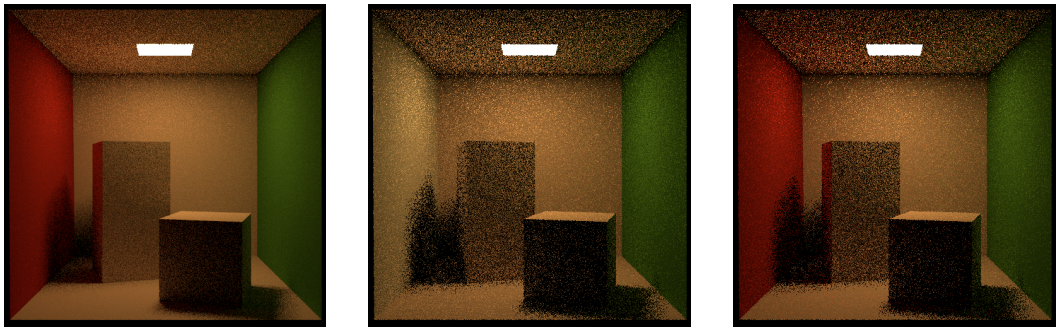
Creating your own custom geometry data representation yields a lot of freedom, but is probably quite technically challenging to implement, therefore we choose to represent our lens with a triangle mesh [Kato et al., 2020].

A key concept in determining the illumination of a pixel in physically based rendering is the collection of *pixel and shading integrals*, of the form

$$I = \int_{\mathcal{X}} f(\mathbf{x}, \boldsymbol{\theta}) \, \mathrm{d}\mathbf{x}, \tag{2.2.5}$$

which sums up all the light that hits that pixel coming from all possible directions, hence the domain $\mathcal{X}$ is often the unit sphere. The integrand f denotes the density of light coming from the direction given by $\mathbf{x}$. $\boldsymbol{\theta}$ denotes the set of all scene parameters: $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_N)$. Differentiable rendering yields a

[5]For instance in section 4.4 a particular choice for image reconstruction is made which is differentiable with respect to geometric scene parameters. The naive bincount reconstruction is not differentiable in this way.



(a) Reference image (b) Initial render (c) Final render

Figure 2.2.5: An example of optimization using differentiable rendering with Mitsuba 2. The Adam optimizer is used to make the right wall match the red color of the reference image. Made using the instructions on Jakob et al., 2019, Differentiable rendering.

pixel-wise derivative with respect to scene parameters, say for instance θ_j:

$$\frac{\partial I}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \int_{\mathcal{X}} f(\mathbf{x}, \boldsymbol{\theta}) \, \mathrm{d}x. \tag{2.2.6}$$

By the Leibniz integration rule, the partial derivative can be taken into the integral if $\mathcal{X}$ does not depend on θ_j (which it never does) and if both f and $\frac{\partial f}{\partial \theta_j}$ exist and are continuous. This is of course never the case for variables that denote integer amounts, but is often the case for variables that can vary continuously, like positions or certain material properties. The main difficulty with these computations is that often the integrand f is not differentiable at every location with respect to certain scene parameters, for instance due to visibility changes.[6] Therefore the key concept in the paper by Loubet et al., 2019, describing the differentiable ray-tracing that Mitsuba 2 uses, is constructing a coordinate transform applied to the integral in eq. (2.2.6) which makes taking the partial derivative into the integral possible.

The integrals in eqs. (2.2.5) and (2.2.6) are evaluated by Monte-Carlo integration [Caflisch, 1998]. For eq. (2.2.5) this yields

$$I \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(\mathbf{x}_i, \boldsymbol{\theta})}{p(\mathbf{x}_i)}, \tag{2.2.7}$$

and for eq. (2.2.6)

$$\frac{\partial I}{\partial \theta_i} \approx \frac{1}{N} \sum_{i=1}^{N} \frac{1}{p(\mathbf{x}_i)} \frac{\partial f(\mathbf{x}_i, \boldsymbol{\theta})}{\partial \theta_j}. \tag{2.2.8}$$

Here p is a probability distribution on $\mathcal{X}$ where the points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are sampled from. It is assumed that p does not depend on $\boldsymbol{\theta}$, which is not always the case. Scene-dependent probability distributions are not used, the interested reader can find the expression for the partial derivative approximation in this case in Loubet et al., 2019.

[6] This can be understood as follows: say you move your hand in front of your eyes to block the sun. Then the movement of your hand causes certain parts of your retina to suddenly receive a lower intensity of light in a discontinuous way.

Figure 2.3.1: 'Caustic Brain': An example of caustic design. The transparent acrylic slab is designed to refract the incoming light into the shape of a human brain. Figure taken from [Schwartzburg et al., 2014].

2.3. Caustic design

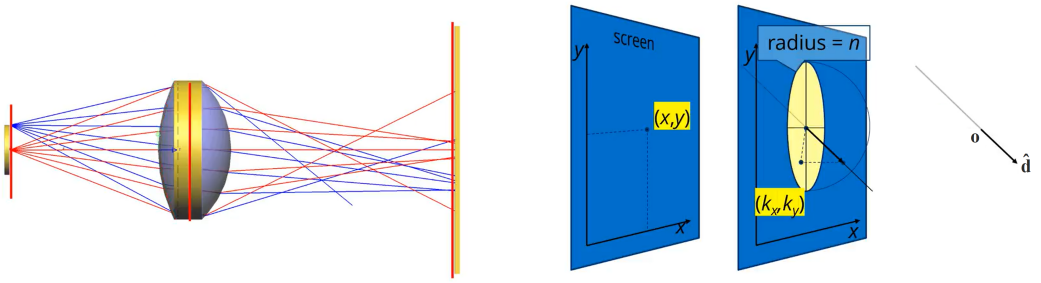
This section discusses the subject of caustic design.

2.3.1. General

Caustic design, also known as *Irradiance Tailoring* or *Beam Shaping*, is a part of *non-imaging optics* concerned with the geometrical design of optical components to transform the incident illumination from light sources into a desired target distribution [Meyron et al., 2018]. An example of this can be seen in fig. 2.3.1.

Caustic design can be done with either reflective surfaces (*i.e.* mirrors) or refractive surfaces (*i.e.* lenses). Possible applications of refractive caustic design are in architecture where a window casts a pleasant caustic pattern into a room [Kiser et al., 2021], or in security, where a hard-to-forge artefact can be 'read out' under normal daylight [Papas et al., 2011].

A more recent development in caustic design is the development of*gradient index optics* (see section 2.1.3), where the optimized variables do not (only) determine the surface geometry, but (also) the local refractive index. Fabrication of materials which such a varying refractive index is an active area of research [Nguyen et al., 2017].



(a) Possible screens (red) for defining a phase space for the light bundle traveling trough an optical system. The screen must not be at an optical surface, since the refractive index and the ray directions must be well defined on the screen.

(b) The relationship between a ray $\mathbf{r}(t) = \mathbf{o} + \hat{\mathbf{d}}t$ in 3-dimensional space and its representation (x, y, k_x, k_y) in phase-space (adapted).

Figure 2.3.2: Explanations of etendue. Figures taken from [OSA, 2019].

2.3.2. Etendue

The etendue of a light beam can be thought of as the 'disorderliness' or 'degree of spread' of a beam of light, and is in fact a form of entropy. As such it is also a quantification of how difficult it is to shape the beam of light in a desired way, which is explained after the definition of etendue below.

To formally introduce the concept of etendue, we first must show how a bundle of light can be represented as a volume in a 4-dimensional space, the *phase-space*. In order to construct this representation, the bundle needs to be intersected with a (not necessarily flat) screen, such as in fig. 2.3.2a. Now the ray $\mathbf{r}(t) = \mathbf{o} + \hat{\mathbf{d}}t$ can be represented by a point $(x, y, k_x, k_y) \in \mathbb{R}^4$ as follows:

- The point (x, y) in the local distance preserving coordinates on the surface defines an origin point of the ray;
- To define the direction of the ray, a disk of radius n (the refractive index of the medium the screen is in) is placed tangent to the screen centered at (x, y), see fig. 2.3.2b. Then each point on the hemisphere defined by this disk defines a possible direction of this ray. This point is uniquely defined by its orthogonal projection on the disk in the coordinates of the surface: (k_x, k_y).

The etendue $\mathcal{U}$ of the beam is defined as the volume of the set $\mathcal{P} \subset \mathbb{R}^4$, the bundle of rays represented in phase-space [OSA, 2019]:

$$\mathcal{U} := \int_{\mathcal{P}} \mathrm{d}x\ \mathrm{d}y\ \mathrm{d}k_x\ \mathrm{d}k_y. \tag{2.3.1}$$

In certain situations the etendue is simple to compute. For instance a point source is *zero-etendue*, since when we place the screen through the source we see that $\mathcal{P}$ has no extent in the x and y directions. A collimated (parallel) beam of light is also zero-etendue, because in this case $\mathcal{P}$ has no extent in the k_x and k_y directions. For a flat surface $A \subset \mathbb{R}^3$ with area $|A|$ that emits light in all directions to one side of the surface, the etendue is

$$\mathcal{U} = \int_{\mathcal{P}} \int_{\mathcal{P}} \mathrm{d}x\ \mathrm{d}y\ \mathrm{d}k_x\ \mathrm{d}k_y \tag{2.3.2a}$$

$$= \int_A \int_{\|(k_x,k_y)\|_2 < n} \mathrm{d}(k_x, k_y)\mathrm{d}(x, y) \tag{2.3.2b}$$

$$= |A| \int_{\|(k_x,k_y)\|_2 < n} \mathrm{d}(k_x, k_y) = |A|\pi n^2. \tag{2.3.2c}$$

Here it is used that in eq. (2.3.2b) the inner and outer integrals are independent from each-other, and πn^2 is the surface area of the tangent circle.

As a beam of light passes through an optical system of lenses and mirrors, the etendue can never decrease, under the condition that no light is lost. Light can be ejected from the system to decrease the etendue, but this method of reducing etendue decreases the efficiency of the system in terms of

making use of the available light. Etendue is only preserved under scattering on an optical surface if that surface is perfectly specular. Such surfaces are infeasible in practice due to, *e.g.* , not being perfectly smooth and causing internal reflections. The non-decreasing nature of etendue is a powerful tool in non-imaging optics to determine bounds on parameters that define certain optical elements.

The complicating property of a finite etendue beam is that if it hits a surface, it does so from a whole solid angle range at a positive area subset of that surface. This makes the relationship between the surface geometry and the light scattered on the surface more complex that with a zero-etendue light beam. Intuitively it can be understood as follows: for a zero etendue beam each point on the refractive surface refracts at most one ray, but for a positive-etendue beam a significant amount of points on the surface can refract a whole range of rays.

In this thesis the term etendue is mainly used to differentiate between zero-etendue point sources and positive-etendue extended sources.

2.3.3. Existing caustic design methods

The non-imaging optics literature offers a variety of methods to design free-form lenses for caustic design, below a few of them are listed.

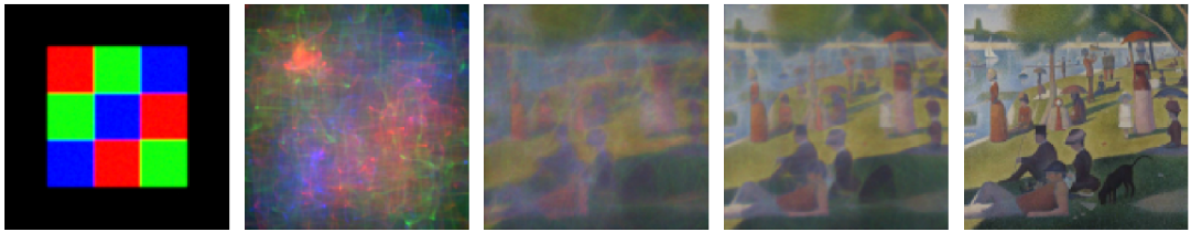
Caustic design methods without differentiable ray-tracing

In what follows we summarize a couple of existing caustic design methods from the literature that do not use differentiable ray-tracing.

- Papas et al., 2011: Here a system for designing and manufacturing surfaces that produce desired caustic images when illuminated by a light source is proposed. The target image is decomposed into a set of (possibly overlapping) anisotropic Gaussian kernels. This decomposition is then used to construct an array of continuous surface patches, each of which focuses light onto one of the Gaussian kernels, either through reflection or refraction. The paper proposes a way to construct these patches and how to arrange them on the lens surface by matching these patches to the Gaussian kernels. The shape of each patch is obtained by deriving and integrating a normal field for the patch. This normal field is a collection of normal vectors that implicitly define a surface.

- Yue et al., 2014: This paper proposes a technique for computing the shape of a free-form lens that generates user-defined caustic patterns. The generated refractive surface of a lens is smooth, and thus the resulting caustic pattern is smooth. The approach is as follows: first differential geometry is used to obtain a smooth mapping between the distributions of the incoming light and the light reaching the screen. Then this mapping is used to compute the surface of the lens. Both these steps are achieved by solving the Poisson equation.

- Schwartzburg et al., 2014: In the work described here also the construction of free-form lenses to create a desired target caustic is discussed. The proposed pipeline is as follows: first a light source, an input lens with one variable surface and a screen are defined. Then rays are traced from the source through the input lens to the screen. Subsequently an optimal transport mapping is obtained on all rays to determine how the screen irradiance must be changed to achieve the target distribution. This then yields a normal field for the variable lens surface as above. This does assume that for each point on the free-form lens surface rays hit it from only one direction, which puts restrictions on the light source. For an example result see fig. 2.3.1.



(a)

(b)

Figure 2.3.3: The progress (a) and the setup (b) of a caustic design process in Mitsuba 2 using differential ray tracing, approximating the painting *A Sunday Afternoon on the Island of La Grande Jatte* by Georges Seurat. The rightmost figure in (a) shows the original painting. Figures taken from Nimier-David et al., 2019, figs. 10c, 1.c.

2.3.4. Caustic design methods with differentiable ray-tracing

Fig 2.3.3 shows an example of a caustic design process using differentiable ray tracing with Mitsuba 2 as discussed in section 2.2.2. Here the lens is a glass slab described by a triangle mesh, where the height of each vertex in the mesh is optimized individually. The optimization takes place using gradient descent (section 2.5.3) based on gradients of the renders with respect to these vertex positions. The incident light consists of parallel beams of light of primary colors, as shown in the left-most sub-figure of fig. 2.3.3a [Nimier-David et al., 2019].

2.3.5. Previous results of neural networks in Caustic design

The research in this thesis described in chapter 1 in some sense follows up on the work of Lucas Crijns [Crijns, 2021], also performed at the Optics group of the Delft University of Technology. In this work Crijns also investigates free-form design using B-spline surfaces using wave optics instead of geometric optics to simulate light, using a technique called *Fraunhofer diffraction*. In this technique the B-spline surface does not describe a surface of a lens, but the *wave phase shift* applied to light passing through a lens.

Crijns had some success with this method, but found that his renders depended too sensitively on the input parameters of his machine learning method for proper optimization. He furthermore found that Fraunhofer diffraction introduces aliasing errors, to which he proposed ray-tracing as a possible alternative.

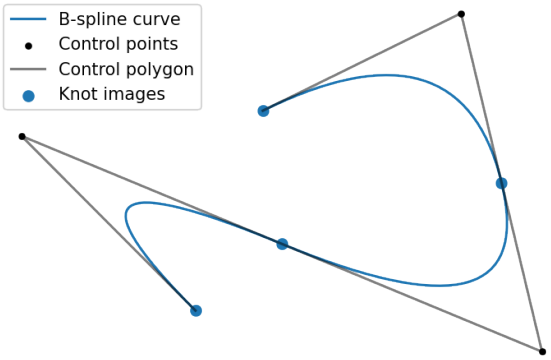


Figure 2.4.1: An example of a B-spline curve of degree two.

2.4. B-splines & NURBS

This section discusses the mathematical formalism of B-splines and NURBS which define parametric geometries.

2.4.1. General

The field of optical design has its own standards for saggable[7] surface descriptions, for instance asphere for circularly symmetric surfaces [Fischer et al., 2008, ch.7] or Zernike polynomials for free-form surfaces [Noll, 1976] (in optics the term *free-form* refers to a surface that has a high level of complexity and is often not circularly symmetric). For our application however we want a surface description which offers a wide variety of representable surfaces for a relatively small amount of parameters, and we want the effect of these parameters to be local. This yields a more tractable object for optimization, because light scattering depends on the local geometry of the surface.

Thus we focus on B-splines and Non-uniform Rational B-splines (NURBS). Traditionally B-spline/NURBS were mainly used by the *computer aided design* (CAD) industry, as a standard for curve and surface descriptions. More recently the use of these descriptions has reached many other fields, like visual entertainment industries and sculpture [Piegl and Tiller, 1997, foreword]. Section 4.1 covers the definition of the free-form lens surfaces used for our caustic design experiments in the form of a B-spline surface.

2.4.2. B-spline and NURBS surfaces

As David F. Rogers writes in his foreword of *The NURBS book*:

> *B-spline curves and surfaces grew out of the pioneering work of Pierre Bézier in the early 1970s. Perhaps one can consider B-spline curves and surfaces the children of Bézier curves and surfaces, and nonuniform rational B-splines, or NURBS, the grandchildren. The timing is about right; they have certainly come of age* [Piegl and Tiller, 1997, foreword].

B-spline curves

A p-th degree B-spline curve in d-dimensional Euclidean space is a parametric curve

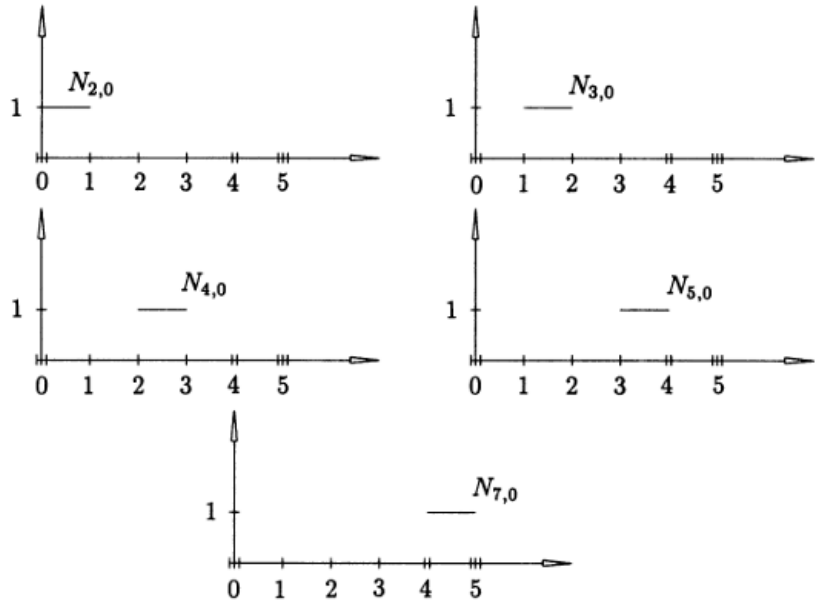
$$\mathbf{C} : [a, b] \to \mathbb{R}^d, \tag{2.4.1}$$

and in the following it will be assumed that $[a, b]$ is normalized to $[0, 1]$. An example of a B-spline curve in $\mathbb{R}^2$ is shown in fig. 2.4.1.

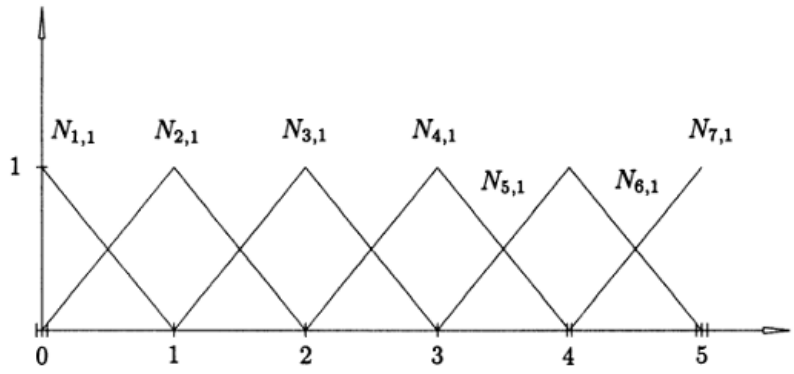
In order to fully specify a B-spline curve, the following are needed:

- A set of *control points* $\{\mathbf{P}_i\}_{i=0}^n \subset \mathbb{R}^d$, together called the *control polygon*;
- A set of B-spline basis functions $\{N_{i,p}\}_{i=0}^n$ of degree p;

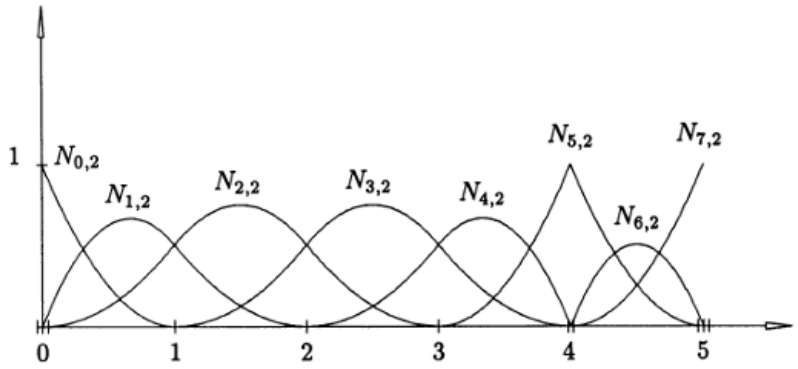
[7] A surface is called saggable if it can be expressed as a function $z = f(x, y)$, where z is called the sag.



(a) Basis functions of order 0.



(b) Basis functions of order 1.



(c) Basis functions of order 2.

Figure 2.4.2: Examples of B-spline basis functions with knot vector $U = \{0, 0, 0, 1, 2, 3, 4, 4, 5, 5, 5\}$. Figures taken from [Piegl and Tiller, 1997, figs. 2.4-2.6].

- A set of *knots* collected in the *knot vector* $\mathcal{V} = (u_1, u_2, \dots, u_m)$ which are non-decreasing real numbers in the B-spline domain $[0, 1]$: $u_1 \leq u_2 \leq \dots \leq u_m$.

The curve is defined as a linear combination of the basis functions in terms of control points:

$$\mathbf{C}(u) := \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i. \tag{2.4.2}$$

The basis functions of increasing degrees are defined recursively and in terms of the knots [Piegl and Tiller, 1997, eq. 2.5]:

$$N_{i,0}(u) := \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1}, \\ 0 & \text{otherwise,} \end{cases} \tag{2.4.3a}$$

$$N_{i,p}(u) := \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u). \tag{2.4.3b}$$

This is called the Cox-de Boor algorithm. Some basis functions of various degrees and a particular knot vector are shown in fig. 2.4.2. Appendix A gives some notes on implementing an efficient computation of these basis functions.

Below important properties of these B-spline basis functions are listed.

- The quotient $\frac{0}{0}$ can occur, which is in this context defined as $\frac{0}{0} := 0$.
- The basis function $N_{i,0}$ has support on the interval $[u_i, u_{i+1})$ (which is called a *knot-span*), and in general the basis function $N_{i,p}$ has support on the interval $[u_i, u_{i+p+1})$. This is called *local support*. This also means that for a well defined B-spline we need a number of $n + p + 1$ knots in total.
- The basis function $N_{i,p}$ is a piece-wise polynomial function of degree p on the knot spans.
- Knot vectors are often in *open* (also called *non-periodic* or *clamped*) form, which means

$$\mathcal{V} = (\underbrace{0, \dots, 0}_{p+1}, u_{p+1}, \dots, u_{r-p-1}, \underbrace{1, \dots, 1}_{p+1}). \tag{2.4.4}$$

  In this case $N_{0,p}(0) = N_{n,p}(1) = 1$.
- Consecutive knots can be identical, in which case we call them *repeated* knots. The amount of the same knot in a knot vector is called its *multiplicity* m. A basis function of order p is C^{p-m} across a knot of multiplicity m and C^p elsewhere.
- The basis functions have the *partition of unity* property: for all $u \in [0, 1]$ and $i = 0, 1, \dots n$ we have that $N_{i,p}(u) \geq 0$ and

$$\sum_{i=0}^{n} N_{i,p}(u) = 1. \tag{2.4.5}$$

B-spline surfaces

The concept of B-spline curves can be extended to parametrizations of sets of arbitrary dimension $\leq d$ (the dimensionality of the embedding space). In this thesis however we are only interested in describing (saggable lens surfaces), *i.e.* 2-dimensional surfaces in $\mathbb{R}^3$, as depicted in fig. 2.4.3. To construct such a B-spline surface, the following is needed:

- A *control net* of points $\{\mathbf{P}_{i,j}\}_{i,j=0}^{n_1,n_2} \subset \mathbb{R}^3$;
- The B-spline basis functions $\{N_{i,p}\}_{i=0}^{n_1}$ and $\{N_{j,q}\}_{j=0}^{n_2}$, which do not necessarily have the same degree (p does not have to be equal to q);
- Two knot vectors $\mathcal{V}_1, \mathcal{V}_2$.

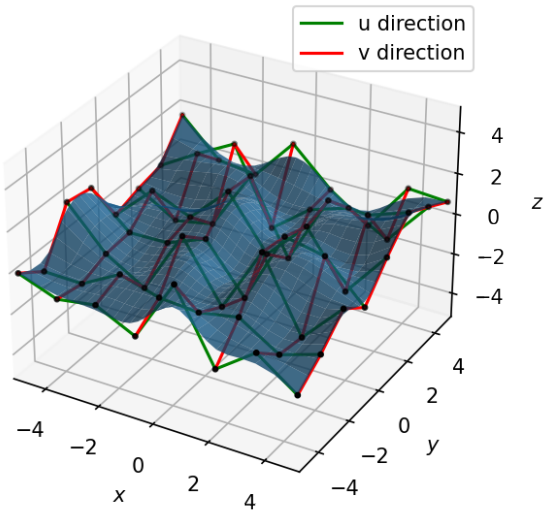
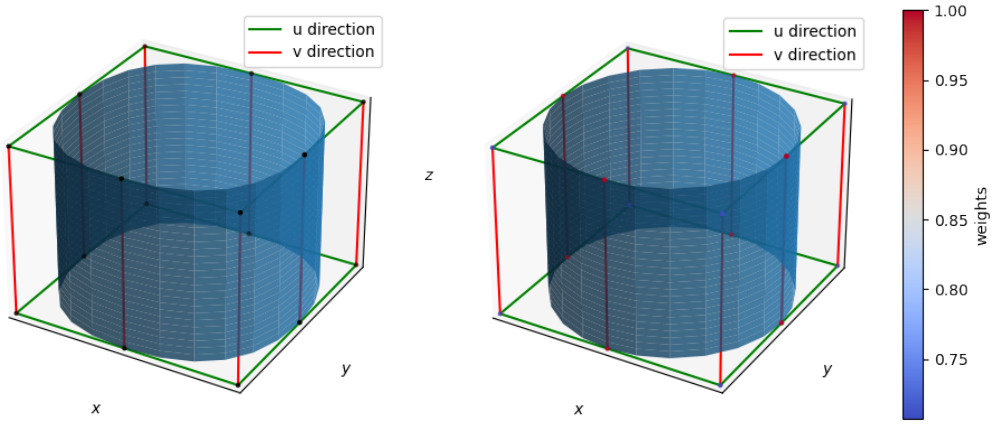


Figure 2.4.3: A saggable B-spline surface of degrees $(3, 3)$ with indicated directions of the u and v parameters.



(a) Approximate representation of a cylinder with a B-spline surface of degrees $(2, 1)$ in the u and v direction respectively.

(b) Exact representation of a cylinder with a NURBS surface of degrees $(2, 1)$ in the u and v direction respectively, with weights chosen as in [Piegl and Tiller, 1997, ex. 7.2, sec 8.3].

Figure 2.4.4: Two NURBS surfaces: left a B-spline surface (or a NURBS surface with trivial weights), right a NURBS surface with specific weights to exactly represent a cylinder.

The B-spline surface

$$\mathbf{S} : [0, 1]^2 \to \mathbb{R}^3 \tag{2.4.6}$$

is then defined as

$$\mathbf{S}(u, v) := \sum_{i=0}^{n_1} \sum_{j=0}^{n_2} N_{i,p}(u) N_{j,q}(v) \mathbf{P}_{i,j}. \tag{2.4.7}$$

NURBS surfaces

NURBS are a generalization of B-spline surfaces:

$$\mathbf{S}(u, v) := \sum_{i=0}^{n_1} \sum_{j=0}^{n_2} \frac{w_{i,j} N_{i,p}(u) N_{j,q}(v)}{\sum_{i'=0}^{n} \sum_{j'=0}^{m} w_{i',j'} N_{i',p}(u) N_{j',q}(v)} \mathbf{P}_{i,j}. \tag{2.4.8}$$

When $w_{i,j} = 1$ for all i, j, then $\mathbf{S}$ is a B-spline surface again, since the enumerator simplifies to one by the partition of unity property of the B-spline basis functions.

NURBS can describe certain curves and surfaces that B-splines cannot. Looking at fig. 2.4.4, we see that this 9×2 control net does not precisely describe a cylinder with a B-spline surface, but it does so with a NURBS surface with the proper weights. In fact, B-splines can not describe exact circles and cylinders with a finite amount of control points. For the construction of cylinders using NURBS surfaces see Piegl and Tiller, 1997, ex. 7.2, sec 8.3.

A nice property of B-spline and NURBS surfaces is the *strong convex hull property*. This property states that for all positive weights, if $(u, v) \in [u_{i_0}, u_{i_0+1}) \times [u_{i_0}, u_{i_0+1})$ then $\mathbf{S}(u, v)$ is in the convex hull of the control points $\mathbf{P}_{i,j}$ where $i_0 - p \leq i \leq i_0$ and $j_0 - q \leq j \leq j_0$ [Piegl and Tiller, 1997, P4.25 on p. 130].

NURBS surfaces are not used in the research performed for this thesis, but are part of the parametric surface implementation and thus NURBS can quite easily be used in follow-up research.

Derivatives

First derivatives of B-spline basis functions are easily computed using the following formula [Piegl and Tiller, 1995, eq. 2.7]:

$$N'_{i,p}(u) = \frac{p}{u_{i+p} - u_i} N_{i,p-1}(u) - \frac{p}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u). \tag{2.4.9}$$

By their definition the first partial derivatives of B-spline surfaces are easy to compute with the formula above:

$$\frac{\partial S}{\partial u}(u, v) = \sum_{i=0}^{n_1} \sum_{j=0}^{n_2} N'_{i,p}(u) N_{j,q}(v) \mathbf{P}_{i,j}, \tag{2.4.10}$$

and the first derivative with respect to v is computed analogously. These derivatives are used in section 4.3.2 to compute normal vectors to the B-spline surface.

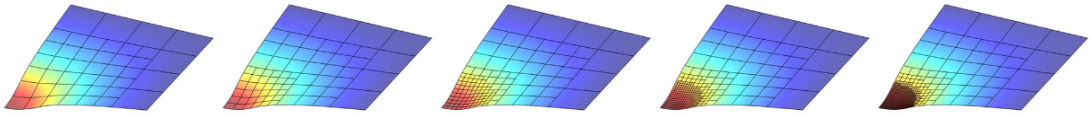


Figure 2.4.5: Local geometry dependent refinement of a NURBS control grid (mesh), here as a model of a rigid sphere deforming a membrane. Figure taken from [Zimmermann and Sauer, 2017, fig. 9].

2.4.3. More generalized surfaces

In this section the shortcomings of B-spline and NURBS surfaces as well as some existing extensions to these concepts are discussed. These are not used in the research for this thesis, but are mentioned here as a possible direction for follow-up research.

Shortcomings of B-spline and NURBS surfaces

Standard B-spline and NURBS surfaces as described above are of a 'tensor-product nature'. This is demonstrated by the fact that if we let

$$\mathbf{N}(u) := \begin{pmatrix} N_{0,p}(u) & \dots & N_{n_1,p}(u) \end{pmatrix}^\mathsf{T} \tag{2.4.11a}$$

$$\mathbf{M}(v) := \begin{pmatrix} N_{0,q}(v) & \dots & N_{n_2,q}(v) \end{pmatrix}^\mathsf{T}, \tag{2.4.11b}$$

then

$$\mathbf{N}(u)\mathbf{M}(v)^\mathsf{T} \in [0,1]^{(n_1+1)\times(n_2+1)} \tag{2.4.12}$$

yields the grid of basis function products used in the evaluation of the surface at $(u, v) \in [0, 1]$, where each product corresponds to a control point. This indicates that the control net is always rectangular, and thus local grid refinement with a few extra control points is not possible. By knot insertion [Piegl and Tiller, 1997, sec. 5.2] it is only possible to add an entire row or column of control points.

Local refinement

Because of the limited capacity for B-spline and NURBS surfaces to perform local refinement, augmented B-spline and NURBS concepts have been developed. These are for instance the locally refinable counterparts: LR B-splines [Patrizi et al., 2020] and LR NURBS [Zimmermann and Sauer, 2017]. An example of this is shown in fig. 2.4.5.

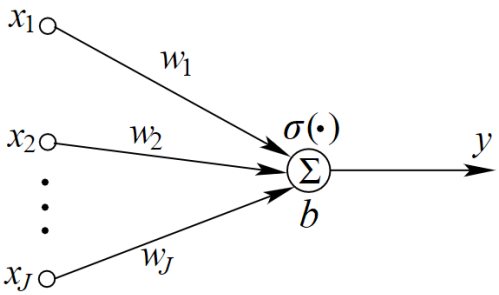


Figure 2.5.1: Schematic depiction of the McCulloch-Pitts neuron model. here σ denotes the activation function, Σ the inner product $\mathbf{w}^\mathsf{T}\mathbf{x}$ and b the bias. Figure adapted from [Du and Swamy, 2019, fig. 1.2].

2.5. Physics-informed machine learning

This section covers the topic of physics-informed machine learning. This theory is incorporated in the optimization pipeline implemented for our experiments. This pipeline is explained in detail in chapter 5.

2.5.1. A brief history

Neural Network (NN) implementations are inspired by the human brain. The average human brain consists of $\sim 10^{11}$ neurons, each neuron connecting to up to tens of thousands of other neurons. Because of this, NN implementations are also called connectionist models. In the 1940s this modelling started with setting up rules to describe the behaviour of biological neurons based on discoveries in their physiology.

The Hopfield model introduced in 1982 [Hopfield, 1982] formed the start of modern research in neural networks. This model works at the system (brain) level rather than the level of a single neuron. It can be used for information storage and solving optimization problems. The most important advance in neural network research is the development of *back-propagation* (BP) in 1986 (discussed in section 2.2.2), although it was later found out that it was already invented in 1974 by Paul Werbos in his dissertation.

The brain is a dynamic processing system that evolves its structure and thus its functionality through processing of information at different hierarchical levels (*i.e.* quantum/molecular/neuron/neuron ensembles/brain/human populations). Building computational models that incorporate these principles prove to be efficient for solving complex problems [Du and Swamy, 2019, sec. 1.1].

Neural networks are now used to approximate function solutions to implicit and possibly very high dimensional problems, which generally cannot be solved analytically.

2.5.2. Neural network architectures

Neurons

In most networks the neuron is modelled as follows:

$$y = \sigma\left(\mathbf{w}^\mathsf{T}\mathbf{x} + b\right). \tag{2.5.1}$$

This is called the McCulloch-Pitts neuron model. Here

- $\mathbf{x} \in \mathbb{R}^J$ is the input vector;
- $\mathbf{w} \in \mathbb{R}^J$ is the weight vector;
- $b \in \mathbb{R}$ is the bias;
- $\sigma : \mathbb{R} \to \mathbb{R}$ is the *activation function*;
- $y \in \mathbb{R}$ is the output.

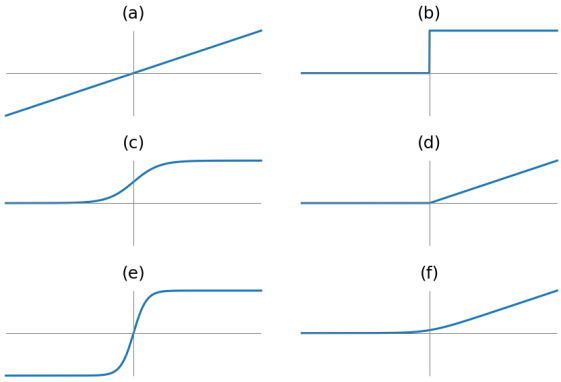


Figure 2.5.2: Some common activation functions as a function of x: (a) linear; x, (b) step; $\mathbb{1}_{(0,\infty)}(x)$, (c) sigmoid; $\frac{1}{1+e^{-x}}$, (d) rectified linear (ReLU); $\max\{0, x\}$, (e) hyperbolic tangent (tanh); $\frac{e^x-e^{-x}}{e^x-e^{-x}}$, (f) softplus; $\ln(1+e^x)$ [Berner et al., 2021, table 1].

This neuron model is depicted in fig. 2.5.1. The weights and the bias of the neuron are the degrees of freedom used to make this neuron (or rather a collection of neurons, see below) represent a desired function.

The activation function is where non-linearity is introduced [Du and Swamy, 2019, sec. 1.2]. Neural networks are typically used to solve non-linear problems, since linear problems can often be solved analytically with linear algebra methods. Examples of common activation functions are shown in fig. 2.5.2.

Layers

A neural network is generally composed of neurons as in eq. (2.5.1) by means of function composition; for most neurons the inputs $\mathbf{x}$ consist of outputs y of other neurons.

A straight-forward way of construction a network is in terms of *layers*; each layer consists of a set of neurons. One neural network architecture type that makes use of layers is the *multi-layer perceptron* (MLP). In this architecture each layer is only connected to the directly preceding and directly following layer. The first layer is called the *input layer*, the last layer is called the *output layer* and the layers in-between are called hidden layers, see fig. 2.5.3a.

One of the simplest ways to connect two consecutive layers in an MLP is *fully connected*. Here every neuron in the first layer is connected to every layer in the second layer. However, sometimes leaving out connections can yield a network with better training behaviour, see for instance Yu et al., 2012. A well known example of this is in *image processing* by neural networks [Albawi et al., 2018]. Say that in both of two consecutive layers of a network each neuron represents a pixel of a (gray-scale) image. In a picture most likely each pixel is most correlated to the pixels closest to it. It therefore makes sense to only connect a neuron in the first layer to neurons in the second layer that represent pixels in some rectangular grid centered around the pixel represented by the neuron in the first layer, see fig. 2.5.3b. This is called a *convolutional layer*, and the rectangle is called the *convolution kernel*.

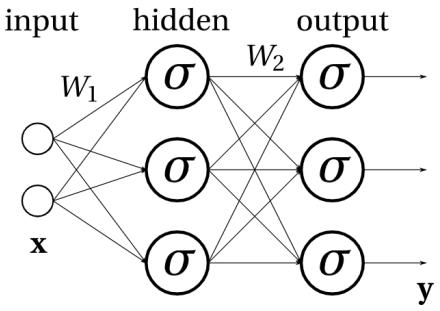
Hyper-parameter tuning

Hyper-parameter tuning is the process of choosing variable values that determine global properties of the neural architecture, for instance layer width and the number of layers (the network *depth*). In many problems the needed size of a neural network to capture the complexity of the desired task of the network is not a-priori known. Therefore there are several methods to hone in on an efficient network, like *grid-search*: this is the process of testing the learning efficiency of each network represented by a point in a grid in the hyper-parameter space. For efficiency here it is good practice to include an *early-stopping-criterion* [Du and Swamy, 2019, sec. 2.2.2]: a rule to determine whether an architecture behaves poorly as soon as possible so that it can be discarded to increase the grid-search efficiency.

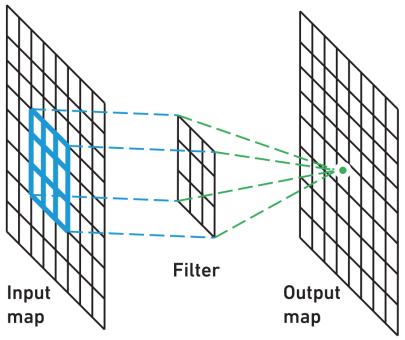
2.5.3. Training/Learning and the loss function

Assuming some neural network architecture (see section 2.5.2), we obtain a function space of functions that are representable by this architecture:

$$\mathcal{F} := \{\mathcal{N}(\,\cdot\,; W) : \Omega \to \mathbb{R}^d \mid W \in \mathcal{W}\}, \tag{2.5.2}$$



(a) Schematic depiction of the multi-layer perceptron neural network architecture. Figure taken from [Crijns, 2021, fig 2.5].



(b) Outline of a convolutional layer. Figure taken from [Yakura et al., 2018, fig. 2].

Figure 2.5.3: Schematic neural network depictions.

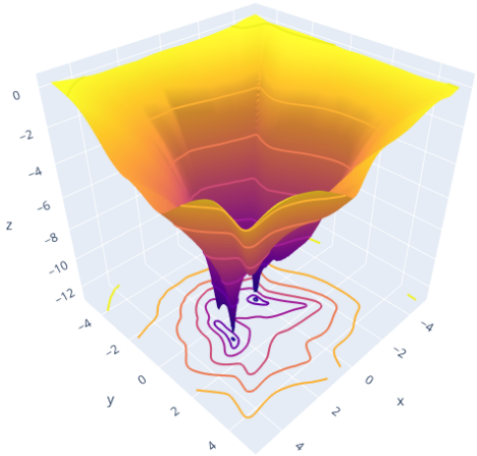


Figure 2.5.4: An example of a loss landscape, where x, y are trainable parameters and z is the loss value. Figure taken from [Berner et al., 2021, fig. 1.5]

where $\mathcal{W}$ denotes the set of all possible value combinations W of the trainable parameters of the neural network, *i.e.* the weights and biases of all neurons. The 'variety' of functions that is present in $\mathcal{F}$ is loosely referred to as the *expressivity* of the architecture.

We want to find the function $\mathcal{N} \in \mathcal{F}$ which most accurately solves our problem. In order to formalize this accuracy we need to define a *loss function*: a function that computes a single number reflecting the correctness of the output of a neural network, often as related to the input. For examples of loss functions see eqs. (2.5.5) and (5.2.1).

For a given input of the network the loss function defines a *landscape* (see fig. 2.5.4) on the space of values of the trainable parameters of the network, in which we want to find the minimum.[8] Since this landscape is defined on the trainable network parameters, the dimensionality of this landscape is determined how many trainable parameters are present in the neural network architecture.

If this landscape varies too much over the various elements in the input set of the network, we cannot consider the problem well posed.

Well posedness can have various meanings: in the strictest sense we can define a problem to be well posed if we can prove that a solutions exists. In an optimisation problem it is more usefull to say that a problem is well posed if a solution with a sufficiently small loss for a sufficient amount of inputs exists.

In order to navigate this landscape it is highly beneficial for this landscape to be smooth, at least C^1. The smoothness depends on various aspects, like the choice of the activation functions and the nature of the loss function. In the case of C^1 smoothness we can apply *gradient descent*, the simplest

[8] Sometimes the loss function involves multiple evaluations of the network but here for simplicity we assume only one.

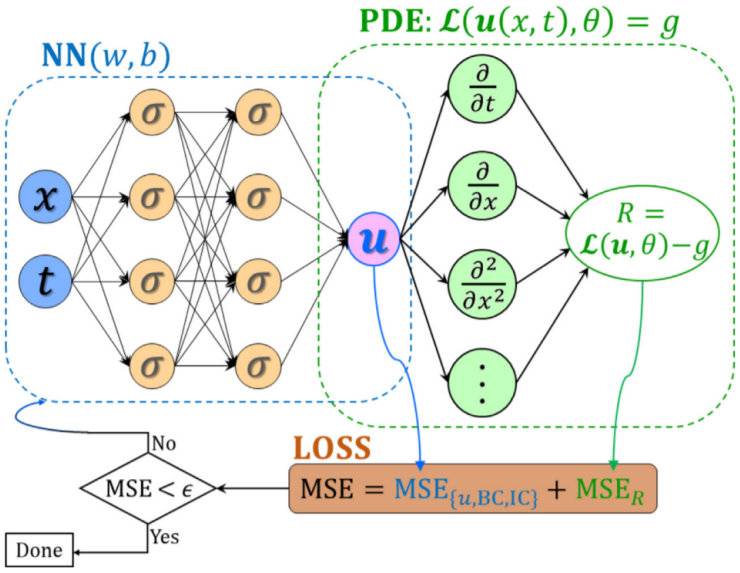


Figure 2.5.5: Schematic depiction of a physics informed neural network (PINN). Here x, t denotes a point in the space-time domain, $\mathcal{L}$ denotes a differential operator dependent on certain parameters θ, u denotes the solution to the differential equation $\mathcal{L}(u,t) = g$. R refers to loss for (random) interior points in the space-time domain. Figure taken from [Meng et al., 2019, fig. 1].

way to navigate this smooth landscape:

$$W^{i+1} = W^i - \eta \nabla_W \text{Loss}. \tag{2.5.3}$$

This defines a step in the landscape based on the current values of the trainable network parameters W^i, the gradient of the loss function with respect to the trainable network parameters (at some input of the network and W^i) ∇_WLoss, and the *learning rate* $\eta > 0$. This is based on the property of the gradient that it points towards the direction of the fastest increase in the landscape, and thus the opposite direction towards the fastest decrease. This is the most naive way of finding a minimum in the landscape, since a fixed learning rate does not take the local geometry of the landscape into account. This can lead to too slow progress for too small a learning rate, or jumping over (local) minima for too large a learning rate.

To improve upon gradient descent multiple *optimizers* have been developed, such as *Adam* [Kingma and Ba, 2014] and *L-BFGS* [Liu and Nocedal, 1989].

2.5.4. Machine learning in physics problems

There has been a lot of progress in understanding multi-scale physics (physics phenomena that have important features on multiple scales of time and/or space) over the past 50 years, in diverse applications from biophysics to geophysics. This has mainly been by solving partial differential equations (PDEs) numerically, using finite difference, finite element, finite volume, spectral or even meshless methods. Despite this progress, modelling and predicting the evolution of nonlinear multi-scale systems using these methods involves high costs and various sources of uncertainty. What's more, solving *inverse* problems (as in the project discussed in this thesis) is often exceedingly expensive and requires fine-tuned theoretical work and programming effort.

Machine learning has been shown to be able to explore massive design spaces and provide sensible answers to ill-posed problems, for instance in detecting climate extremes [Kurth et al., 2018]. In the most general form, physics-informed machine learning can be described as *'the process by which prior knowledge stemming from our observational, empirical physical or mathematical understanding of the world can be leveraged to improve the performance of a learning algorithm'* [Karniadakis et al., 2021].

Physics Informed Neural Networks (PINNs)

A recent example of the physics-informed machine learning philosophy is the physics-informed neural network (PINN), an example of which is depicted in fig. 2.5.5. The idea is to leverage the power of neural networks as *universal function approximators* [Berner et al., 2021, thm. 1.15] to some general

differential equation with boundary conditions. Somewhat formally, we consider the following problem for $u : \Omega \to \mathbb{R}^c$:

$$\mathcal{D}u = f \quad \text{on } \Omega \subset \mathbb{R}^d, \tag{2.5.4a}$$
$$\mathcal{B}u = g \quad \text{on } \partial\Omega, \tag{2.5.4b}$$

where $\mathcal{D}$ and $\mathcal{B}$ are (potentially non-linear) differential operators, and f and g are given functions on Ω and $\partial\Omega$, respectively. Note that the exact solution u must come from a space of sufficiently smooth functions to be able to evaluate these differential operators.[9]

In order to programmatically find the function in $\mathcal{F}$ (eq. (2.5.2)) that best approximates the solution to eq. (2.5.4), this problem must be turned into a loss function where the deviation from eq. (2.5.4) is computed at discrete points. This loss function is (for instance) defined as follows:

$$\text{Loss}_{\tilde{u}} = w_\Omega \sum_{i=1}^{n} \|\mathcal{D}\tilde{u}(\mathbf{x}_i) - f(\mathbf{x}_i)\|_2^2 + w_{\partial\Omega} \sum_{j=1}^{m} \|\mathcal{B}\tilde{u}(\mathbf{x}_j) - g(\mathbf{x}_j)\|_2^2, \quad \tilde{u} \in \mathcal{F}. \tag{2.5.5}$$

Here the image of the differential operators are evaluated in the interior points $\{\mathbf{x}_i\}_{i=1}^n \subset \Omega$ and the boundary points $\{\mathbf{x}_j\}_{j=1}^n \subset \partial\Omega$, respectively. The derivatives within these differential operators are evaluated by means of back-propagation (section 2.2.2). The terms $w_\Omega, w_{\partial\Omega} \geq 0$ are weights to prioritize certain terms in the loss.

Now we can reformulate our problem in terms of the neural network to minimize over to find the $W^* \in \mathcal{W}$ such that the loss is minimal:

$$W^* := \underset{W \in \mathcal{W}}{\text{argmin}}\, \text{Loss}_{\mathcal{N}(\,\cdot\,;W)}. \tag{2.5.6}$$

Neural networks in the context of caustic design

In what follows, we will explain the application of neural networks as a general framework to formulate caustic design processes. We want to find a lens or mirror surface that redistributes light from one or multiple sources into a desired target distribution on a detector screen. These optical surfaces are chosen from some finite-dimensional function space $\mathcal{F}$ of functions $\mathbf{S} : [0,1]^2 \to \mathbb{R}^3$, for instance in the form of B-spline or (LR-)NURBS surfaces. The dimensionality of $\mathcal{F}$ depends on the chosen hyper-parameters of these surfaces, like the number of control points and the inclusion of weights (for NURBS).

We look at optics modelling in the form of-ray tracing, so the modelled setup will be referred to as a *scene*. This scene consists of one or more light sources, a mirror or lens and a detector screen. All the parameters that describe this scene are collectively denoted by $\boldsymbol{\theta}$, which are a specific element of the set $\boldsymbol{\Theta}$ of all possible choices of scene parameter values. The parameters $\boldsymbol{\theta}$ form the input of the ray-tracing operator $\mathcal{R}$. This then produces an output image $\mathcal{I}$.

To evaluate the quality of $\mathcal{I}$ we need a loss function Loss, which compares the output of $\mathcal{R}$ and the desired target distribution $\mathcal{I}_t$. So now we are interested in finding

$$\boldsymbol{\theta}^* := \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\text{argmin}}\, \text{Loss}(\mathcal{R}(\boldsymbol{\theta}); \mathcal{I}_t). \tag{2.5.7}$$

However, not all parameters will be variable in the optimization process, so we split the variables into *dependent*, *independent* and *fixed* ones: $\boldsymbol{\theta} = (\boldsymbol{\theta}^{\text{ind}}, \boldsymbol{\theta}^{\text{dep}}, \boldsymbol{\theta}^{\text{fix}})$. Fixed parameters are for instance the size of the screen. Independent parameters describe a certain scenario of the scene we are interested in, for instance light sources at a specific location. The dependent parameters are the ones we want to find, as a function of the dependent ones. These dependent parameters are always the optical surface parameters (*i.e.* control point coordinates and possibly weights) which define a function in $\mathcal{F}$, but might in some optimization experiments also involve the source locations. Thus a refined problem statement is finding the following mapping:

$$\mathcal{M} : \boldsymbol{\Theta}^{\text{ind}} \to \boldsymbol{\Theta}^{\text{dep}} \quad \text{such that} \quad \mathcal{M} : \theta^{\text{ind}} \mapsto \underset{\boldsymbol{\theta}^{\text{dep}} \in \boldsymbol{\Theta}^{\text{dep}}}{\text{argmin}}\, \mathcal{L}(\mathcal{R}(\boldsymbol{\theta}^{\text{ind}}, \boldsymbol{\theta}^{\text{dep}}, \boldsymbol{\theta}^{\text{fix}}); \mathcal{I}_t). \tag{2.5.8}$$

[9] Furthermore, we consider these operators in the classical sense instead of in the 'almost-everywhere' sense, since later-on we require that the image of these operators is point-wise evaluatable.

One example of a set $\boldsymbol{\Theta}^{\text{ind}}$ is a finite set of source locations of interest. The mapping $\mathcal{M}$ will be approximated by a neural network $\mathcal{N}(\boldsymbol{\theta}^{\text{ind}}; W)$ where W denotes the set of trainable parameters of the neural network (*i.e.* weights and biases). In order to make the training of this network feasible, $\boldsymbol{\Theta}^{\text{ind}}$ should not be too large.

Note that in eq. (2.5.8) the mapping $\mathcal{M}$ is not defined from the equation $\mathcal{R}(\boldsymbol{\theta}^{\text{ind}}, \mathcal{M}(\boldsymbol{\theta}^{\text{ind}}), \boldsymbol{\theta}^{\text{fix}}) = \mathcal{I}_t$. This is because it is not always a priori clear whether a $\boldsymbol{\theta}^{\text{dep}} \in \boldsymbol{\Theta}^{\text{dep}}$ exists that results *exactly* in the desired target distribution.

In the experiments performed for this thesis the mapping $\mathcal{M}$ is trivial. This is because each network is trained to only optimize the lens geometry and the rest of the scene is fixed, so there are only fixed and 'dependent' scene parameters.

3

Analytical ray-tracing problem statement

This chapter looks at general problem statements in caustic design from an analytical ray-tracing perspective and motivates the need for numerical methods to solve the arising non-linear optimization problems.

3.1. General problem statement

Let P be the set in 4D phase space (as defined in section 2.3.2) representing the light that leaves some light source. Then ray-tracing defines[1] a mapping $\mathbf{M}_{\boldsymbol{\theta}} : P \to S$ where S is the screen given by

$$S := [-R_x, R_x] \times [-R_y, R_y] \times \{z_{\text{screen}}\} \tag{3.1.1}$$

and $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ is a particular choice of parameters for the scene. There also exists a function $W_{\boldsymbol{\theta}} : P \to \mathbb{R}_{\geq 0}$ where $W_{\boldsymbol{\theta}} \geq 0$ which for each point in P gives the weight of that ray as it intersects the screen.

A measure $\mu_{\boldsymbol{\theta}}$ can be defined on S with its Borel σ-algebra $\mathcal{B}(S)$:[2][3]

$$\mu_{\boldsymbol{\theta}} : \mathcal{B}(S) \to [0, \infty), \qquad \mu_{\boldsymbol{\theta}}(A) = \int_{\mathbf{M}_{\boldsymbol{\theta}}^{-1}(A)} W_{\boldsymbol{\theta}} \mathrm{d}\lambda. \tag{3.1.2}$$

Here λ is a measure on P with an appropriate σ-algebra. Note that if the light source is positive etendue then λ should measure 4-volume, otherwise λ should measure the appropriate lower dimensional sets.

In contrary to some other free-form lens design methods like the one described in Schwartzburg et al., 2014, injectivity and thus invertibility of $\mathbf{M}_{\boldsymbol{\theta}}$ is not assumed; multiple rays can hit the same point on the screen. Thus $\mathbf{M}_{\boldsymbol{\theta}}^{-1}$ is the (always well-defined) set operation in-stead of the inverse function.

In particular $\mu_{\boldsymbol{\theta}}$ can be used to create an image matrix $\mathcal{I}(\boldsymbol{\theta}) \in \mathbb{R}_{\geq 0}^{n_x \times n_y}$ where $n_x \times n_y$ determines the resolution:

$$\mathcal{I}_{i,j}(\boldsymbol{\theta}) = \mu_{\boldsymbol{\theta}}\left(A_{i,j}\right), \qquad A_{i,j} = \left[\left(\frac{2i}{n_x} - 1\right) R_x, \left(\frac{2(i+1)}{n_x} - 1\right) R_x\right) \times \left[\left(\frac{2j}{n_y} - 1\right) R_y, \left(\frac{2(j+1)}{n_y} - 1\right) R_y\right) \times \{z_{\text{screen}}\}. \tag{3.1.3}$$

This corresponds to the bincount image reconstruction method explained in section 4.4. Computational ray-tracing with a finite amount of rays is effectively a Monte-Carlo integration approximation of $\mu_{\boldsymbol{\theta}}(A_{i,j})$.

[1] The term 'defines' here is debatable from a physics standpoint. A physicist would say that the optical system defines the mapping and ray-tracing is a way to evaluate this mapping.

[2] The Borel σ-algebra $\mathcal{B}(S)$ of a set S is the σ-algebra generated by all the open sets in S. More generally a σ-algebra of a set S is a collection of subsets of S which can all be associated a 'volume' in a consistent way by a measure.

[3] This is indeed a well-defined measure since :

- $\mu_{\boldsymbol{\theta}}(\emptyset) = \int_{\emptyset} W_{\boldsymbol{\theta}} \mathrm{d}\lambda = 0$,
- $\mu_{\boldsymbol{\theta}}(A) \geq 0$ by $W_{\boldsymbol{\theta}} \geq 0$,
- $\mu_{\boldsymbol{\theta}}\left(\bigcup_{n=1}^{\infty} A_n\right) = \int_{\bigcup_{n=1}^{\infty} \mathbf{M}_{\boldsymbol{\theta}}^{-1}(A_n)} W_{\boldsymbol{\theta}} \mathrm{d}\lambda = \sum_{n=1}^{\infty} \int_{\mathbf{M}_{\boldsymbol{\theta}}^{-1}(A_n)} W_{\boldsymbol{\theta}} \mathrm{d}\lambda = \sum_{n=1}^{\infty} \mu_{\boldsymbol{\theta}}(A_n)$ for a pairwise disjoint collection of sets $(A_n)_{n \geq 1} \subset \mathcal{B}(S)$.

This formalism can also be extended to multiple sources; in this case the measure $\mu_{\boldsymbol{\theta}}$ is defined as in eq. (3.1.2) but then as a sum over an integral per source.

In general, an inverse problem for finding the appropriate scene parameters $\boldsymbol{\theta}^*$ based on $\mathbf{M}_{\boldsymbol{\theta}}$ can be obtained by minimising over $\boldsymbol{\Theta}$ the difference of $\mu_{\boldsymbol{\theta}}$ to reference values on certain sets. Using a reference image $\mathcal{I}_{\text{ref}} \in \mathbb{R}_{\geq 0}^{n_x \times n_y}$ this can be expressed as

$$\boldsymbol{\theta}^* := \operatorname*{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \|\mathcal{I}(\boldsymbol{\theta}) - \mathcal{I}_{\text{ref}}\| \tag{3.1.4}$$

with some appropriate norm, for instance the Euclidean matrix norm introduced in section 5.2.

3.2. Derivation of the ray mapping for a plane wave sources

In this section, we define an explicit formula for the mapping $\mathbf{M}_{\boldsymbol{\theta}}$ for the case of a single plane wave light source and the other scene parameters chosen as follows:

- A specular reflecting/refracting lens, given by

$$\left\{(x,y,z) \in \mathbb{R}^3 \mid (x,y) \in P,\ 0 \leq z_{\text{in}} \leq f_{\boldsymbol{\theta}}(x,y)\right\}, \qquad P = [-r_x, r_x] \times [-r_y, r_y], \tag{3.2.1}$$

  for some function $f_{\boldsymbol{\theta}} \in C^1(P)$ such that $f_\theta(x,y) > z_{\text{in}}$ everywhere.
- A plane wave light source shining from $z = -\infty$ to the positive z direction within P.
- A screen as in the previous section.

Note that the surface of the lens at $z = z_{\text{in}}$ is orthogonal to the light beam, so only minimal reflection and no refraction of the transmitted light occurs. Therefore we look at refraction at the surface given by $f_{\boldsymbol{\theta}}$. We choose some $(x_0, y_0) \in P$ and determine the unit normal vector to the refractive surface:

$$\hat{\mathbf{n}} = \frac{1}{\sqrt{1 + \|\nabla f_{\boldsymbol{\theta}}\|_2^2}} \begin{pmatrix} \partial_x f_{\boldsymbol{\theta}} \\ \partial_y f_{\boldsymbol{\theta}} \\ -1 \end{pmatrix}. \tag{3.2.2}$$

The incoming direction of the rays is simply $\hat{\mathbf{s}} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^\mathsf{T}$, and so the inner product yields

$$\langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle = -\frac{1}{\sqrt{1 + \|\nabla f_{\boldsymbol{\theta}}\|_2^2}}. \tag{3.2.3}$$

Then Snell's law (eq. (2.1.10)) gives

$$\hat{\mathbf{t}} = n\hat{\mathbf{s}} - \left(n\langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle - \sqrt{1 - n^2\left(1 - \langle \hat{\mathbf{s}}, \hat{\mathbf{n}} \rangle^2\right)}\right) \hat{\mathbf{n}} \tag{3.2.4}$$

$$= \begin{pmatrix} 0 \\ 0 \\ n \end{pmatrix} + \frac{n - \sqrt{1 - (1 - n^2)\|\nabla f_{\boldsymbol{\theta}}\|_2^2}}{1 + \|\nabla f_{\boldsymbol{\theta}}\|_2^2} \begin{pmatrix} \partial_x f_{\boldsymbol{\theta}} \\ \partial_y f_{\boldsymbol{\theta}} \\ -1 \end{pmatrix}. \tag{3.2.5}$$

This yields the mapping

$$\mathbf{M}_{\boldsymbol{\theta}} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \frac{z_{\text{screen}} - f_{\boldsymbol{\theta}}}{\hat{t}_z} \begin{pmatrix} \hat{t}_x \\ \hat{t}_y \end{pmatrix} \tag{3.2.6}$$

$$= \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \frac{n - \sqrt{1 + (n^2 - 1)\|\nabla f_{\boldsymbol{\theta}}\|_2^2}}{n\|\nabla f_{\boldsymbol{\theta}}\|_2^2 + \sqrt{1 + (n^2 - 1)\|\nabla f_{\boldsymbol{\theta}}\|_2^2}} (z_{\text{screen}} - f_{\boldsymbol{\theta}}) \nabla f_{\boldsymbol{\theta}}. \tag{3.2.7}$$

The above equation demonstrates that even in this minimal setup the mapping $\mathbf{M}_{\boldsymbol{\theta}}$ and its dependency on the parameters $\boldsymbol{\theta}$ is in general non-trivial.

For some far-field applications the distance $z_{\text{screen}} - f_{\boldsymbol{\theta}}$ might be considered constant, since in those applications the refraction direction $\hat{\mathbf{t}}$ dominates over the place at which a ray leaves the lens in terms of determining where the ray hits the screen. This however yields a narrow purpose method, which is not the aim of this report. In a numerical ray-tracing approach such simplifications are not necessary. Furthermore, calculating ray intersections with free-form surfaces generally requires a numerical approach, further solidifying the need for numerical ray-tracing methods.

4

The differentiable ray-tracer

This chapter explains the theory behind the implemented ray-tracer.[1] The ray-tracer is sequential, which means that for each ray only a path from source to lens entrance surface to lens exit surface to screen is considered. The main direction of these paths is $+z$ in a Cartesian coordinate system.

4.1. The B-spline lens

This section explains how the considered free-form lenses are defined and and motivates this choice.

4.1.1. Lens definition

The used lenses have a rectangular extent in the x and y direction given by $r_x, r_y > 0$:

$$[-r_x, r_x] \times [-r_y, r_y].$$

In the z-direction the lens is bounded by a flat surface at z_{in} and a B-spline surface $\mathbf{S} = (X, Y, Z)$ at the opposite side. We want the B-spline surface to have the nice property that the mapping $(u, v) \to (x, y)$ is linear, since it simplifies many of the upcoming computations on ray sampling and intersections. We can achieve this by making use of Marsden's identity [Cohen et al., 2010, eq. 23]:

$$u = \sum_{i=0}^{n} u^*_{i,p} N_{i,p}(u), \quad u \in [0,1], \quad u^*_{i,p} = \frac{u_{i+1} + \ldots + u_{i+p}}{p}. \tag{4.1.1}$$

Where the $u^*_{i,p}$ are called the *Greville abscissae*. Using this identity we can choose the x, y coordinates of the control points such that

$$X : u \mapsto (2u - 1) r_x \in [-r_x, r_x], \quad Y : v \mapsto (2v - 1) r_y \in [-r_y, r_y]. \tag{4.1.2}$$

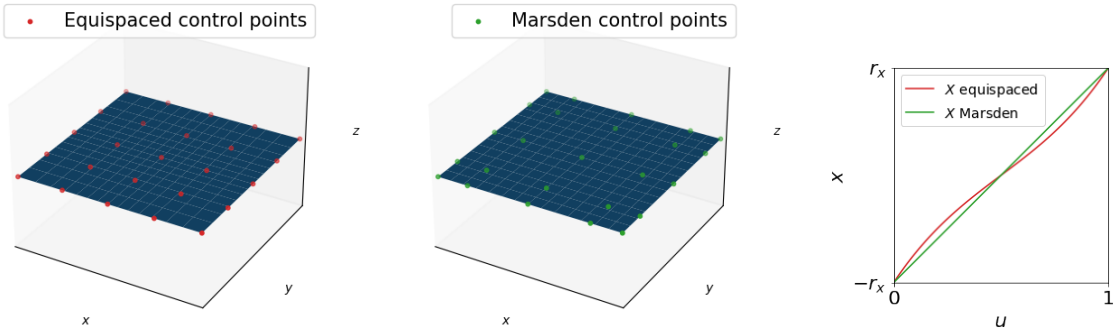
Using the definition of X we obtain

$$X(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P^x_{i,j} N_{i,p}(u) N_{j,q}(v) \tag{4.1.3a}$$

$$= \sum_{i=0}^{n} P^x_{i,0} N_{i,p}(u) \underbrace{\sum_{j=0}^{m} N_{j,q}(v)}_{=1}. \tag{4.1.3b}$$

Here the $N_{j,q}$ summation is 1 by the partition of unity property and we assume that the $P^x_{i,j}$ are identical for all j and a fixed i, so $j = 0$ is chosen as a representative. Now we see that if we define $P^x_{i,j} := u^*_{i,p}$

[1] This ray-tracer was developed as a substitute for the software Mitsuba 2 [Nimier-David et al., 2019], which turned out not to be suitable for the application discussed in this thesis at the time of writing. For more details on Mitsuba 2 see appendix E.



(a) Equi-spaced control points leading to non-linear X, Y.

(b) Control points distributed according to eq. (4.1.4) leading to linear X, Y.

(c) The mapping $X(u)$ for the two control point configurations. The mapping $Y(v)$ is completely analogous.

Figure 4.1.1: A flat B-spline surface $\mathbf{S}(u, v) = (X(u), Y(v), 0)$ with different control point grids leading to different functions X, Y. Both surfaces are evaluated on an equispaced grid in $\Omega = [0, 1]^2$ with open and equispaced knot-vectors.

then $X(u) = u$. Thus if we apply the mapping $u \mapsto (2u - 1)r_x$ to both sides of eq. (4.1.1), we obtain

$$(2u - 1)r_x = \left(2\left[\sum_{i=0}^{n} u^*_{i,p} N_{i,p}(u)\right] - 1\right) r_x = \sum_{i=0}^{n} (2u^*_{i,p} - 1) r_x N_{i,p}(u). \tag{4.1.4}$$

The second equality can be understood by expanding the 1 into the $N_{i,p}(u)$ by the partition of unity property. Thus if we define $P^x_{i,j} := (2u^*_{i,p} - 1)r_x$ and equivalently $P^y_{i,j} := (2v^*_{j,q} - 1)r_y$, then eq. (4.1.2) is satisfied.

Fig. 4.1.1 demonstrates the difference between an equispaced control net and one informed by eq. (4.1.1) in evaluating an equispaced grid in $\Omega = [0, 1]^2$ using open and equispaced knot-vectors. Note that eq. (4.1.1) still yields some equispaced interior control points in this case if their sum of knots does not contain more than one of the repeated boundary knots. Equispaced knot vectors will be used to define the lenses in this thesis, although that is not an assumption that is needed for the following computations.

4.1.2. Well-definedness

In order to create a well defined lens, the flat surface and the B-spline surface of the lens must not intersect. By the convex hull property of B-spline surfaces [Piegl and Tiller, 1995, P3.22, p. 105] it suffices to check that all control points are above z_{in}:[2]

$$P^z_{i,j} > z_{\text{in}} \quad \forall i, j. \tag{4.1.5}$$

Thus the lens is given by

$$z_{\text{in}} \leq z \leq Z(u, v) = Z\left(\frac{1}{2}\left(\frac{x}{r_x} + 1\right), \frac{1}{2}\left(\frac{y}{r_y} + 1\right)\right), \quad (x, y) \in [-r_x, r_x] \times [-r_y, r_y]. \tag{4.1.6}$$

[2] Note that this condition here only suffices to obtain a well defined simulation. For manufacturing purposes imposing stronger restrictions might be needed, like some minimal thickness $\delta < P^z_{i,j} - z_{\text{in}}$.

Figure 4.2.1: Render (bincount) from the plane wave source with 2.5×10^5 rays directly onto the screen for several ray sampling methods: (a) grid, (b) grid perturbed, (c) random uniform.

4.2. Sources and sampling

In this section various light sources, also called *emitters*, are introduced. These sources determine the origin $\mathbf{o} \in \mathbb{R}^3$ and direction $\hat{\mathbf{d}} \in \mathbb{R}^3$ such that $\|\hat{\mathbf{d}}\|_2 = 1$ of the start of a ray

$$l : t \mapsto \mathbf{o} + \hat{\mathbf{d}}t. \tag{4.2.1}$$

4.2.1. Plane wave

The plane wave source is the simplest source to model. It can be thought of as a plane equal in size and parallel to the lens entrance surface, but a bit further to the negative z direction (how much precisely does not matter).[3] All rays are emitted orthogonally to this plane in the positive z direction. Note that the flat entrance surface does not affect the direction of these rays.[4] Thus, to be computationally efficient, these rays can be sampled on the B-spline exit surface itself. Here the particular choice of control point x, y values as explained in section 4.1 is useful: because of this a uniform probability distribution on $\Omega = [0, 1]^2$ is equivalent to a uniform probability distribution on $[-r_x, r_x] \times [-r_y, r_y]$.

Ray weights

To properly model the transport of electromagnetic energy, each ray gets an associated ray weight w which represents the power it carries. For the plane wave this is dependent on the flux E at the point where the ray leaves the source. We only consider plane waves with some uniform flux E_0, which yields a total power of $\Phi_0 = 4r_x r_y E_0$.

If we have some (unbiased) sampling of n_{rays} rays, then each ray gets a weight of

$$w_0 := \frac{\Phi_0}{n_{\text{rays}}} = \frac{4r_x r_y E_0}{n_{\text{rays}}} \tag{4.2.2}$$

for conservation of power.

In the case that E is not constant one could use

$$w_i := \frac{E(u_i, v_i)}{\sum_{j=1}^{n_{\text{rays}}} E(u_j, v_j)} \int_\Omega E(u, v) \mathrm{d}(u, v). \tag{4.2.3}$$

Ray sampling

several different sampling methods on Ω for the plane wave have been implemented, which are demonstrated in fig. 4.2.1:

[3] This describes a specific case of a plane wave considered here. The bundle z cross-section does not have to be the same size as the lens x, y-extend, but generally it is not larger. Furthermore, the rays do not have to be orthogonal to the lens entrance surface, but can come in at a different angle. In this case the z distance between the source and the lens entrance surface does matter.

[4] This can be seen from Snell's law in section 2.1.3. Note that there is a small reflection coefficient R_0.

- **Grid** sampling samples an equispaced $m \times m$ grid on Ω (and is thus equispaced on the B-spline surface if $r_x = r_y$).
- **Grid perturbed** sampling adds a perturbation sampled from $U\left(\left[-\frac{1}{m-1}, \frac{1}{m-1}\right]\right)$ to the grid in Ω from the previous method.
- **Random uniform** sampling samples from $U([0,1]^2)$.

The choice of sampling method varies per application.

4.2.2. Point source

A point source is defined by a point $(x_s, y_s, z_s) \in \mathbb{R}^3$ and an intensity function $I(\phi, \theta)$. For a well-defined simulation we need that $z_s < z_{\text{in}}$, and we always take $(x_s, y_s) \in [-r_x, r_x] \times [-r_y, r_y]$.[5] We only consider point sources with a constant intensity in all directions and thus the emission is fully determined by the power Φ_0. Given some (unbiased) sampling of n_{rays} rays, the ray weights are then simply

$$w_0 := \frac{\Phi_0}{n_{\text{rays}}}. \tag{4.2.4}$$

To uniformly sample ray direction vectors

$$\hat{\mathbf{d}} \in \left\{\mathbf{x} \in \mathbb{R}^3 : \|\mathbf{x}\|_2 = 1\right\} \tag{4.2.5}$$

one can use

$$\hat{\mathbf{d}} = (\cos\theta \sin\phi, \sin\theta \sin\phi, \cos\phi), \tag{4.2.6}$$

where θ is sampled from $U([0, 2\pi])$ and

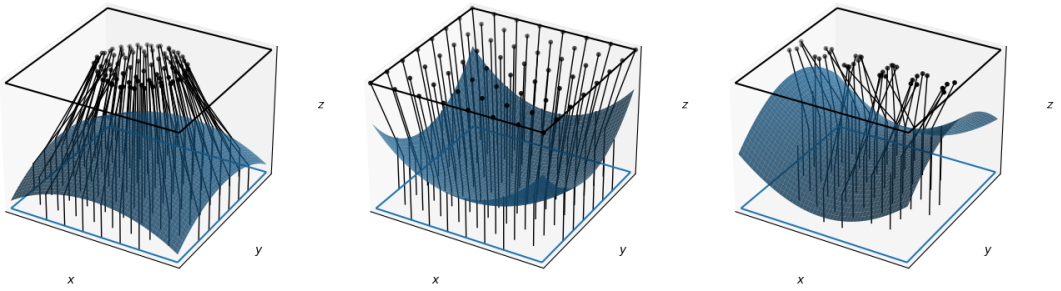
$$\phi = \arccos\left(1 - (1 - \cos\phi_{\max})\, a\right) \tag{4.2.7}$$

where a is sampled from $U([0,1])$ [Simon, 2015]. Here $\phi_{\max} \in [0, \pi]$ determines the maximum ϕ, as shown in fig. 4.3.2b. For point sources only random uniform sampling is used, which samples θ and a as above.

4.2.3. Ray sampling artefacts

The goal of defining a set of rays and associated ray weights is to create an unbiased discretization of the electromagnetic energy transport as emitted by a light source and transported trough an optical system. A proper definition and analysis of unbiasedness requires a study into stochastics, which goes beyond the scope of this thesis. The interested reader may consult Pharr et al., 2017, ch. 7. The introduction of Monte-Carlo noise is already discussed in section 2.2.1.

[5] This often happens in practice, and avoids rays entering the lens through the sides.

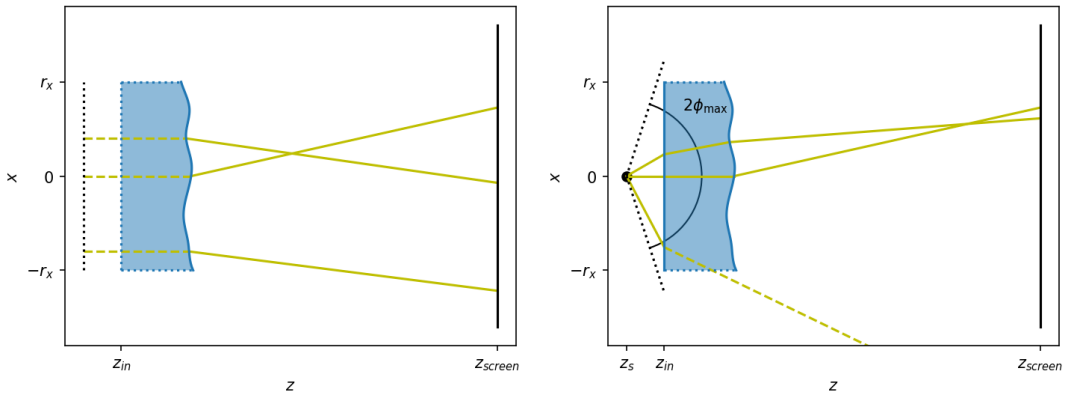


(a) Convex lens creating converging ray bundle.

(b) Concave lens creating diverging ray bundle.

(c) Saddle lens creating both a converging and diverging ray bundle, in different directions.

Figure 4.3.1: Ray refraction for a plane wave reaching a convex, concave and saddle lens In blue the entrance and free-form surfaces are shown, the black square is the screen.



(a) Schematic of the ray-tracing for a plane wave source.

(b) Schematic of the ray-tracing for a point source.

Figure 4.3.2: Ray tracing for a plane wave and a point source.

4.3. Ray-tracing

This section discusses the algebraic background of the implemented ray-tracing algorithm.

4.3.1. Sequential

It is assumed that light only enters the lens at the entrance surface. The x, y boundaries of the lens are considered to be absorbent, so no refraction or internal reflection takes place there.[6]

Only the ray path

$$\text{entrance surface} \rightarrow \text{screen} \tag{4.3.1a}$$

is considered for plane waves (see figs. 4.3.1 and 4.3.2a), and the ray path

$$\text{point source} \rightarrow \text{entrance surface} \rightarrow \text{free-form surface} \rightarrow \text{screen} \tag{4.3.1b}$$

for point sources (see fig. 4.3.2b).

4.3.2. B-spline intersection

To refract rays coming from a point source through the B-spline surface, intersections of the rays with the B-spline must be computed. Analytically this yields a $p+q$ degree piece-wise polynomial equation. To show this, first we transform the ray

$$\mathbf{r}(t) = \mathbf{o} + \hat{\mathbf{d}}t$$

[6] Intersections with the sides of the lens are not computed. These intersections do not occur in the plane wave case. In the point source case, by the constraints on x_s, y_s, rays cannot intersect the lens sides from the outside of the lens. Rays that do not intersect the B-spline surface after the entrance surface are considered lost to the system, which is equivalent to them being absorbed by the lens sides.

to u, v, z-space:

$$\tilde{\mathbf{r}}(t) = \tilde{\mathbf{o}} + \tilde{\mathbf{d}}t = \begin{pmatrix} o_u \\ o_v \\ o_z \end{pmatrix} + \begin{pmatrix} d_u \\ d_v \\ d_z \end{pmatrix} t = \begin{pmatrix} \frac{1}{2}\left(\frac{o_x}{r_x} + 1\right) \\ \frac{1}{2}\left(\frac{o_y}{r_y} + 1\right) \\ o_z \end{pmatrix} + \begin{pmatrix} \frac{d_x}{2r_x} \\ \frac{d_y}{2r_y} \\ d_z \end{pmatrix} t. \tag{4.3.2}$$

Then finding the intersection point of a ray with the B-spline surface comes down to finding the smallest real and positive root t of the function

$$\begin{aligned} f(t) &= Z\left(\begin{pmatrix} o_u \\ o_v \end{pmatrix} + \begin{pmatrix} d_u \\ d_v \end{pmatrix} t\right) - d_z t - p_z, \\ &= -d_z t - p_z + \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(o_u + d_u t) N_{j,q}(o_v + d_u t), \end{aligned} \tag{4.3.3}$$

if such a root exists. If such a root does not exist, then the ray misses the B-spline surface and is lost to the system, or equivalently can be thought of as being absorbed by one of the lens sides.

The function f is a piece-wise polynomial function of degree $p + q$, whose roots can thus not be computed analytically for $p + q > 4$. A numerical root finding scheme is the most flexible solution here. The following options are considered:

1. A Newton-Rhapson iterative method;

$$t_{n+1} = t_n - \frac{f(t_n)}{f'(t_n)} = t_n - \frac{Z\left(\begin{pmatrix} o_u \\ o_v \end{pmatrix} + \begin{pmatrix} d_u \\ d_v \end{pmatrix} t_n\right) - d_z t_n - p_z}{\nabla Z\left(\begin{pmatrix} o_u \\ o_v \end{pmatrix} + \begin{pmatrix} d_u \\ d_v \end{pmatrix} t_n\right)^\top \begin{pmatrix} d_u \\ d_v \end{pmatrix} - d_z}. \tag{4.3.4}$$

2. A triangle mesh intersection method.

The iterative method can be a good solution in some applications [Piegl and Tiller, 1995, p. 230], under the condition that a good first guess for the intersection can be obtained. Note however that the ray-tracer must be differentiable. Incorporating an iterative method in a pipeline that will be back-propagated is best to be avoided, since the iterative method yields a large and thus computationally expensive computational graph to back-propagate if the required number of iterations is high. This also depends on a good choice for a first guess, which is non-trivial.

A good compromise might be to use the triangle mesh intersection method to obtain a good first guess for the iterative method. This however is a very computationally expensive solution. Here we choose for only the triangle mesh method, to obtain a relatively computationally cheap proof-of concept intersection algorithm. The rest of this section is devoted to a triangle mesh intersection method making use of B-spline properties, divided in several phases.

Triangle mesh intersection phase 1: bounding boxes

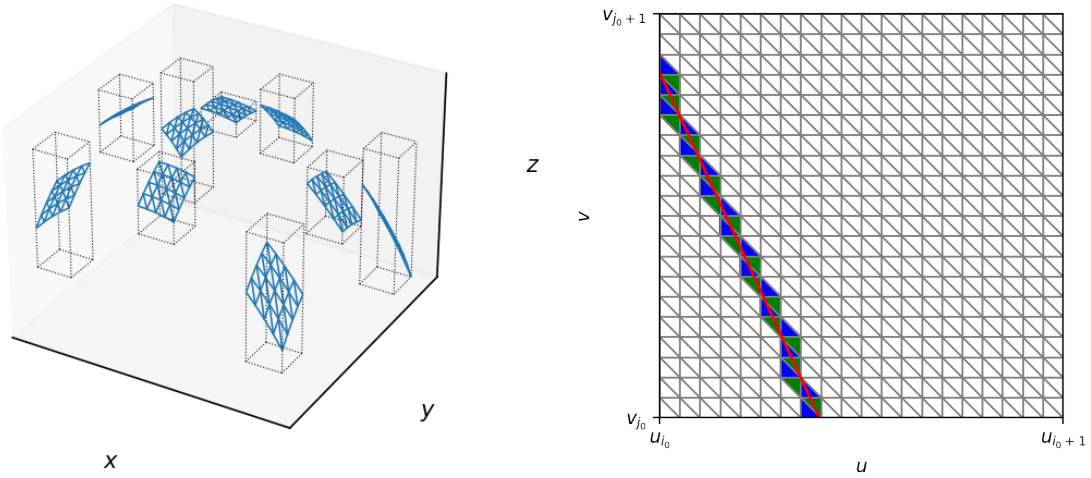
Since checking every ray against every triangle for intersection is computationally very expensive, it is helpful to have some bounding box tests which are a rough first test to see whether the ray is even in the vicinity of some section of the B-spline surface. Luckily the B-spline theory provides a tool for this in the form of the strong convex hull property (section 2.4.2). The convex hull property yields the bounding box

$$\left[u_{i_0}, u_{i_0+1}\right) \times \left[v_{j_0}, u_{j_0+1}\right) \times \left[\min_{\substack{i_0-p \le i \le i_0 \\ j_0-q \le j \le j_0}} P^z_{i,j}, \max_{\substack{i_0-p \le i \le i_0 \\ j_0-q \le j \le j_0}} P^z_{i,j}\right] \tag{4.3.5}$$

for the knot span product $\left[u_{i_0}, u_{i_0+1}\right) \times \left[v_{j_0}, u_{j_0+1}\right)$. Examples of such bounding boxes are shown in fig. 4.3.3a.

I have implemented two ways to apply these bounding boxes in the intersection algorithm. The first step is always a test for the entire surface (in (u, v, z)-space):

$$[0, 1]^2 \times \left[\min_{i,j} P^z_{i,j}, \max_{i,j} P^z_{i,j}\right]. \tag{4.3.6}$$



(a) Triangles and corresponding bounding box for a few knot span products of a spherical surface.

(b) Example of which triangles are candidates for a ray-surface intersection with the ray plotted in red, based on their u, v-domain.

Figure 4.3.3: Demonstrations of different phases of the implemented B-spline intersection algorithm.

For the second step there are 2 options:

- A boundary box test per knot span product.
- A recursive method where, starting with all knot span products, each rectangle of knot span products is divided into at most 4 sub-rectangles for a new bounding box test, until individual knot span products are reached.

The latter method is used for all results in this report.

Note that these bounding boxes make use of the strong convex hull property in a fairly weak way. A future iteration of this algorithm could make full use of the strong convex hull property to make the algorithm more efficient.

Triangle mesh intersection phase 2: (u, v)-space triangle intersection

Each non-trivial knot span product $\Omega_{i_0,j_0} := [u_{i_0}, u_{i_0+1}) \times [v_{j_0}, v_{j_0+1})$ is divided into a grid of n_u by n_v rectangles. Thus we can define the points

$$u_{i_0,k} := u_{i_0} + k\Delta u_{i_0}, \quad \Delta u_{i_0} := \frac{u_{i_0+1} - u_{i_0}}{n_u}, \quad k = 0, \dots, n_u, \tag{4.3.7a}$$

$$v_{i_0,\ell} := v_{j_0} + \ell\Delta v_{j_0}, \quad \Delta v_{j_0} := \frac{v_{j_0+1} - v_{j_0}}{n_v}, \quad \ell = 0, \dots, n_v. \tag{4.3.7b}$$

Each rectangle $[u_{i_0,k}, u_{i_0,k+1}) \times [v_{j_0,\ell}, v_{j_0,\ell+1})$ is divided into a lower left and an upper right triangle. In fig. 4.3.3b it is shown for a ray projected onto the (u, v)-plane in some Ω_{i_0,j_0} which triangles are candidates for an intersection in (u, v, z)-space. This is determined by the following rules:

- A lower left triangle is intersected in the (u, v)-plane if either its left or lower boundary is intersected by the ray;
- an upper right triangle is intersected in the (u, v)-plane if either its right or upper boundary is intersected by the ray.

The intersection of these boundaries can be determined by finding the indices of the horizontal lines at which the vertical lines are intersected:

$$\ell_k = \left\lfloor \frac{o_v + (u_{i_0,k} - o_u)\frac{d_v}{d_u} - v_{j_0}}{\Delta v_{j_0}} \right\rfloor, \tag{4.3.8}$$

and analogously k_ℓ.

Triangle mesh intersection phase 3: u, v, z-space triangle intersection
A lower left triangle can be expressed by a plane

$$T(u,v) = Au + Bv + C \tag{4.3.9}$$

defined by the following linear system:

$$\begin{pmatrix} u_{i_0,k} & v_{j_0,\ell} & 1 \\ u_{i_0,k+1} & v_{j_0,\ell} & 1 \\ u_{i_0,k} & v_{j_0,\ell+1} & 1 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} z_{i_0,k}^{j_0,\ell} \\ z_{i_0,k+1}^{j_0,\ell} \\ z_{i_0,k}^{j_0,\ell+1} \end{pmatrix}. \tag{4.3.10}$$

Here the following definitions are used:

$$z_{i_0,k}^{j_0,\ell} := Z^{i_0,j_0}(u_{i_0,k}, v_{j_0,\ell}), \tag{4.3.11}$$

$$Z^{i_0,j_0}(u,v) := Z|_{\Omega_{i_0,j_0}}(u,v) = \sum_{i=i_0-p}^{i_0} \sum_{j=j_0-q}^{j_0} P_{i,j}^z N_{i,p}(u) N_{j,q}(v). \tag{4.3.12}$$

This yields the plane

$$T(u,v) = z_{i_0,k}^{j_0,\ell} + n_u \left(z_{i_0,k+1}^{j_0,\ell} - z_{i_0,k}^{j_0,\ell}\right) \frac{u - u_{i_0,k}}{u_{i_0+1} - u_{i_0}} + n_v \left(z_{i_0,k}^{j_0,\ell+1} - z_{i_0,k}^{j_0,\ell}\right) \frac{v - v_{j_0,\ell}}{v_{j_0+1} - v_{j_0}}. \tag{4.3.13}$$

Note that to define this triangle, the B-spline basis functions are evaluated at fixed points in Ω independent of the rays or the $P_{i,j}^z$. This means that for a lens that will be optimized these basis function values can be evaluated and stored only once rather than in every iteration, for computational efficiency.

Computing the intersection with the ray $\tilde{\mathbf{r}}(t) = \tilde{\mathbf{o}} + \tilde{\hat{\mathbf{d}}}t$ is now straight-forward, and yields

$$t_{\text{int}} = -\frac{C + \langle \tilde{\mathbf{o}}, \mathbf{n} \rangle}{\langle \tilde{\hat{\mathbf{d}}}, \mathbf{n} \rangle}, \quad \mathbf{n} = \begin{pmatrix} 0 \\ 1 \\ \partial_u T \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ \partial_v T \end{pmatrix} = \begin{pmatrix} A \\ B \\ -1 \end{pmatrix}, \tag{4.3.14}$$

where $\mathbf{n}$ is a normal vector to the triangle, computed using the cross product. This also explains why $\langle \tilde{\hat{\mathbf{d}}}, \mathbf{n} \rangle = 0$ does not yield a well-defined result: in this situation the ray is parallel to the triangle.

The last thing to check is whether $\tilde{l}(t_{\text{int}})$ lies in the (u,v)-domain of the triangle, which can be checked by three inequalities for the three boundaries of the triangle:

$$o_u + d_u t_{\text{int}} \geq u_{i_0,k} \quad \wedge \quad 0 \leq o_v + d_v t_{\text{int}} - v_{j_0,\ell} < \frac{n_u}{n_v} \frac{v_{j_0+1} - v_{j_0}}{u_{i_0+1} - u_{i_0}} (u_{i_0,k+1} - (o_u + d_u t_{\text{int}})). \tag{4.3.15}$$

The computation for an upper right triangle is completely analogous. The upper triangle has a closed boundary where the lower triangle has an open one and vice versa, which means that the (u,v) domains of the triangles form an exact partition of Ω. Thus the triangle mesh is 'water-tight', meaning that no ray intersection should be lost by rays passing in between triangles.

Normal vector computation
Since $\tilde{\mathbf{r}}(t_{\text{int}})$ provides an approximation of the (u,v)-values of the intersection, these values can be used to provide a normal vector to the B-spline surface at the intersection for use in Snell's law:

$$\frac{\partial_u \mathbf{S}^{i_0,j_0} \times \partial_v \mathbf{S}^{i_0,j_0}}{\left\| \partial_u \mathbf{S}^{i_0,j_0} \times \partial_v \mathbf{S}^{i_0,j_0} \right\|} \Bigg|_{\begin{pmatrix} u_{\text{int}} \\ v_{\text{int}} \end{pmatrix}} = \frac{1}{\sqrt{\left[r_y \partial_u Z^{i_0,j_0}\right]^2 + \left[r_x \partial_v Z^{i_0,j_0}\right]^2 + 4 r_x^2 r_y^2}} \begin{pmatrix} -r_y \partial_u Z^{i_0,j_0} \\ -r_x \partial_v Z^{i_0,j_0} \\ 2 r_x r_y \end{pmatrix} \Bigg|_{\begin{pmatrix} u_{\text{int}} \\ v_{\text{int}} \end{pmatrix}}. \tag{4.3.16}$$

This is in contrast to what often happens in ray-tracing with a triangle mesh, where normal vectors are provided at the vertices and interpolated to get normal vectors on the triangles between the vertices (for instance in Phong shading [Newman and Phong, 1975]).[7]

[7] Phong shading was not implemented but is probably significantly faster than calculating normals from the B-spline surface. A study of the loss in accuracy when switching to something like Phong shading would be needed.

Screen intersection
From a ray l leaving the lens it is straight-forward to compute the intersection with the screen at $z = z_{\text{screen}}$ and its effective radii R_x^* and R_y^*:

$$t_s = \frac{z_{\text{screen}} - o_z}{d_z}, \quad x_{\text{screen}} = o_x + d_x t_s, \quad y_{\text{screen}} = o_y + d_y t_s, \tag{4.3.17a}$$

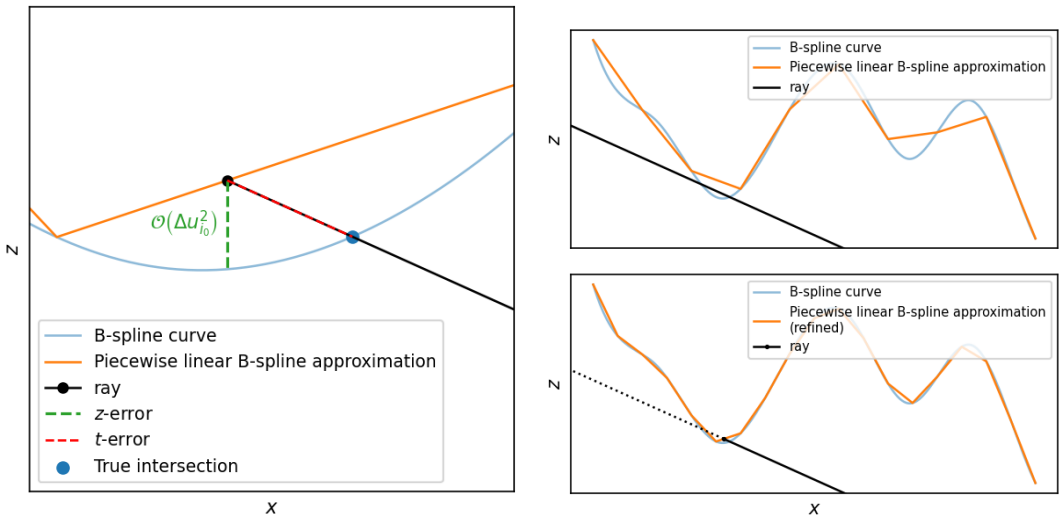
with the conditions

$$t_s > 0, \quad |x_{\text{screen}}| \leq R_x^*, \quad |y_{\text{screen}}| \leq R_y^*. \tag{4.3.17b}$$

The radii of the screen in the scene are R_x, R_y, but are slightly enlarged to R_x^*, R_y^* in this computation to better handle rays that hit the screen near the screen edges, which is explained in the next section.

Alternative and ray paths
The described ray-tracing algorithm only considers rays that go from a source through both lens surfaces to the screen. This could mean that some ray paths from the source to the screen are missed, like paths that involve internal reflections in the lens, or are wrong, like paths that intersect the free-form surface more than once. It is assumed that both of these factors are insignificant for the accuracy of the simulation as an optics model.

Ray weights
As discussed in section 4.2, each ray has an associated ray weight as it leaves the source. The final weight of a ray as it hits the screen is given by the product of this initial ray weight and (one minus) Schlick's approximation (section 2.1.3) for the refraction at both lens surfaces.



(a) Demonstration of the z-error and the t-error.

(b) Demonstration of how the path of a ray can change drastically due to a change in the triangle mesh refinement.

Figure 4.3.4: Ray intersection algorithm demonstrations.

4.3.3. Intersection algorithm convergence

To show that the triangle intersection method is well behaved we prove that it converges to a root of eq. (4.3.3) for $n_u, n_v \to \infty$.

Consider a ray intersecting with the triangle mesh within Ω_{i_0,j_0} in the lower left triangle given by the lower left vertex point $(u_{i_0,k}, v_{j_0,\ell})$, at $(u_{i_0,k} + \delta u, v_{j_0,\ell} + \delta v)$. Then for $0 \leq \delta u \leq \Delta u_{i_0}, 0 \leq \delta v \leq \Delta v_{j_0}$ we can make the following Taylor expansion:

$$Z(u_{i_0,k} + \delta u, v_{j_0,\ell} + \delta v) = Z(u_{i_0,k}, v_{j_0,\ell}) + \frac{\partial Z}{\partial u}(u_{i_0,k}, v_{j_0,\ell})\delta u + \frac{\partial Z}{\partial v}(u_{i_0,k}, v_{j_0,\ell})\delta v \tag{4.3.18a}$$

$$+ \frac{1}{2}\langle(\delta u, \delta v), \nabla\rangle^2 Z(u_{i_0,k} + \theta\delta u, v_{j_0,\ell} + \theta\delta v) \tag{4.3.18b}$$

for some $0 \leq \theta \leq 1$.

Note that Z is a piece-wise polynomial function defined on a finite domain and on a finite partition of Ω, so all the derivatives of Z are bounded. Thus by setting $\delta u = \Delta u_{i_0}, \delta v = 0$ and rearranging we find

$$\frac{\partial Z}{\partial u}(u_{i_0,k}, v_{j_0,\ell}) = \frac{Z(u_{i_0,k+1}, v_{j_0,\ell}) - Z(u_{i_0,k}, v_{j_0,\ell})}{\Delta u_{i_0}} + \mathcal{O}\left(\Delta u_{i_0}\right), \tag{4.3.19a}$$

and analogously with $\delta u = 0, \delta v = \Delta v_{j_0}$

$$\frac{\partial Z}{\partial v}(u_{i_0,k}, v_{j_0,\ell}) = \frac{Z(u_{i_0,k}, v_{j_0,\ell+1}) - Z(u_{i_0,k}, v_{j_0,\ell})}{\Delta v_{j_0}} + \mathcal{O}\left(\Delta v_{j_0}\right). \tag{4.3.19b}$$

Comparing the above to the triangle function T in eq. (4.3.13) as used in the intersection algorithm we find that in fact

$$T(u_{i_0,k} + \delta u, v_{j_0,\ell} + \delta v) = Z(u_{i_0,k} + \delta u, v_{j_0,\ell} + \delta v) + \mathcal{O}\left(\Delta u_{i_0}^2\right) + \mathcal{O}\left(\Delta v_{j_0}^2\right) + \mathcal{O}\left(\Delta u_{i_0}\Delta v_{j_0}\right). \tag{4.3.20}$$

As T is a piece-wise linear approximation of Z, as expected this yields a quadratic error. If for simplicity we assume that $n_v \propto n_u$, we obtain the simpler expression

$$T(u_{i_0,k} + \delta u, v_{j_0,\ell} + \delta v) = Z(u_{i_0,k} + \delta u, v_{j_0,\ell} + \delta v) + \mathcal{O}\left(\Delta u_{i_0}^2\right). \tag{4.3.21}$$

Note that by definition $\Delta u_{i_0} \propto n_u^{-1}$, so the error term is equivalently given by $\mathcal{O}\left(n_u^{-2}\right)$, which is independent of the specific knot span Ω_{i_0,j_0}.

We call the error term derived above the z-error, as indicated in fig. 4.3.4a. This is not the same as the t-error, which is the distance between the true intersection of the ray with the B-spline surface and the intersection with the triangle mesh. To investigate the order of the t-error, we define the following vectors:

$$\mathbf{v}_{\text{num}} := \tilde{\mathbf{r}}(t_{\text{int}}) = (u_{\text{num}}, v_{\text{num}}, T(u_{\text{num}}, v_{\text{num}})), \tag{4.3.22a}$$

$$\mathbf{v}_{\text{proj}} := (u_{\text{num}}, v_{\text{num}}, Z(u_{\text{num}}, v_{\text{num}})), \tag{4.3.22b}$$

$$\mathbf{v}_{\text{ext}} := (u_{\text{ext}}, v_{\text{ext}}, Z(u_{\text{ext}}, v_{\text{ext}})). \tag{4.3.22c}$$

These are the numerical intersection, the numerical intersection projected onto the B-spline surface in the z-direction and the exact intersection respectively. For the t-error we then find

$$\|\mathbf{v}_{\text{num}} - \mathbf{v}_{\text{ext}}\|_2 \leq \left\|\mathbf{v}_{\text{num}} - \mathbf{v}_{\text{proj}}\right\|_2 + \left\|\mathbf{v}_{\text{proj}} - \mathbf{v}_{\text{ext}}\right\|_2 \tag{4.3.23a}$$

$$\leq \underbrace{\left|T(u_{\text{num}}, v_{\text{num}}) - Z(u_{\text{num}}, v_{\text{num}})\right|}_{\mathcal{O}(n_u^{-2})} + \left|Z(u_{\text{num}}, v_{\text{num}}) - Z(u_{\text{ext}}, v_{\text{ext}})\right| \tag{4.3.23b}$$

$$+ \sqrt{(u_{\text{num}} - u_{\text{ext}})^2 + (v_{\text{num}} - v_{\text{ext}})^2}.$$

It is hard to conclude from this what the order of the t-error will be. The last two terms are only well behaved if $\mathbf{v}_{\text{num}}$ and $\mathbf{v}_{\text{ext}}$ are in the domain of the same triangle, which is not necessarily the case. This also depends on the specific B-spline surface. Furthermore, a change to n_u, n_v can have a large effect on $\mathbf{v}_{\text{num}}$; the change can make the difference between a hit or a miss, as demonstrated in fig. 4.3.4b.

This analysis would be better in a coordinate system aligned with the ray, but analysing the B-spline surface in such a coordinate system is non-trivial.

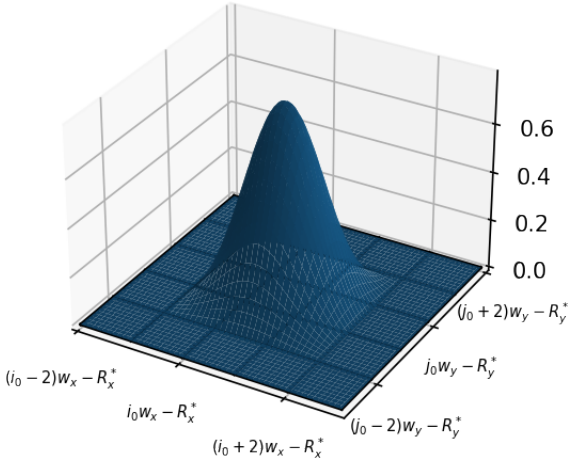


Figure 4.4.1: Gaussian reconstruction filter for $\alpha = 1$ and $(v_x, v_y) = (3, 3)$.

4.4. Image reconstruction

The set of ray intersections on the continuous detector screen must be turned into an image in the form of a matrix of brightness (power) values. The simplest way to achieve this is by using *bincount*, *i.e.* for each pixel its brightness is given by the sum of the ray weights of the rays that intersect that pixel. Note however that this is a discrete operation for rays with shifting paths during optimization; a ray that crosses the border between pixels causes a discontinuous jump in the brightness values of both those pixels.

Thus a smoother image reconstruction method must be used to yield an image that is differentiable with respect to parameters that affect the ray paths, *i.e.* the B-spline control point coordinates.

4.4.1. Gaussian reconstruction

The Gaussian image reconstruction here is inspired by Pharr et al., 2017, section 7.8.1.

Let the screen size $[-R_x, R_x] \times [-R_y, R_y]$ and the pixel resolution (n_x, n_y) be given, and define the pixel size

$$(w_x, w_y) = \left(\frac{2R_x}{n_x}, \frac{2R_y}{n_y}\right). \tag{4.4.1}$$

For reasons explained later in this section, a few extra 'ghost pixels' are added around the perimeter of the screen, which will not be part of the final render but are part of the image reconstruction process. This gives the effective screen radii

$$R_x^* := R_x + \frac{v_x - 1}{2} w_x, \qquad R_y^* := R_y + \frac{v_y - 1}{2} w_y, \tag{4.4.2}$$

and resolution $(n_x + v_x - 1, v_y + v_y - 1)$, where (v_x, v_y) are odd positive integers.

Using these radii the screen can be divided into a partition $\left(A_{i,j}\right)_{i=0,j=0}^{n_x+v_x-2, n_y+v_y-2}$ of pixels, where

$$A_{i,j} := [iw_x - R_x^*, (i+1)w_x - R_x^*) \times [jw_y - R_y^*, (j+1)w_y - R_y^*). \tag{4.4.3}$$

Say some ray intersects the screen in A_{i_0,j_0}. To define the contribution of the ray to this and the surrounding pixels, we define a *filter function* with a drop-off away from the center

$$\left(c_{i_0}^x, c_{j_0}^y\right) := \left(\left(i_0 + \frac{1}{2}\right) w_x - R_x, \left(j_0 + \frac{1}{2}\right) w_y - R_y\right) \tag{4.4.4}$$

of A_{i_0,j_0}:

$$f_{i_0}^x(x; v_x) = \begin{cases} e^{-\alpha\left(x - c_{i_0}^x\right)^2} - e^{-\alpha\left(\frac{v_x w_x}{2}\right)^2} & \text{if } \left|x - c_{i_0}^x\right| < \frac{v_x w_x}{2}. \\ 0 & \text{otherwise.} \end{cases} \tag{4.4.5}$$

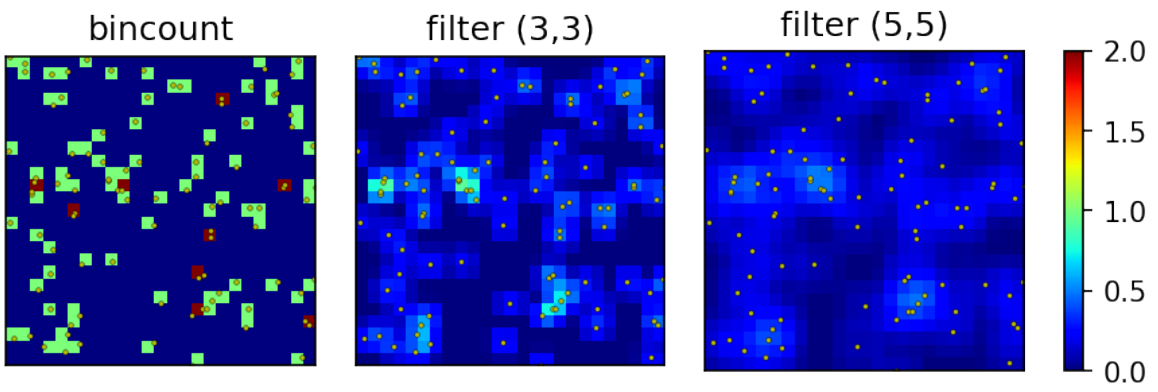


Figure 4.4.2: The image reconstruction based on a set of ray-screen intersections, for bincount and various reconstruction filter sizes and $\alpha = 1$.

$f^y_{j_0}(y; \nu_y)$ is defined analogously.

The full filter function is then given by the product

$$F_{i_0,j_0}(x,y) := f^x_{i_0}(x;\nu_x) f^y_{j_0}(y;\nu_y). \tag{4.4.6}$$

An example of this with $\alpha = 1$ and $(\nu_x, \nu_x) = (3,3)$ is shown in fig. 4.4.1. Here:

- α is a parameter that is roughly the inverse of the variance of the filter function.
- The values (ν_x, ν_y) together are called the *reconstruction filter size*, and determine the support of the filter function, These are odd integers which determine how many pixels the support of the filter function is wide and high. More precisely, this support is

$$\bigcup_{i=i_0-\frac{\nu_x-1}{2}}^{i_0+\frac{\nu_x-1}{2}} \bigcup_{j=j_0-\frac{\nu_y-1}{2}}^{j_0+\frac{\nu_y-1}{2}} A_{i,j}. \tag{4.4.7}$$

The sets in the union in eq. (4.4.7) also represent all the pixels that get a power contribution from a ray intersecting at $(x_{\text{int}}, y_{\text{int}}) \in A_{i_0,j_0}$. The extended image matrix $\mathcal{I} \in \mathbb{R}_{\geq 0}^{(n_x+\nu_x-1)\times(n_y+\nu_y-1)}$ from this ray alone is then given by

$$\mathcal{I}_{i,j} = w \frac{F_{i,j}(x_{\text{int}}, y_{\text{int}})}{\sum_{i'=i_0-\frac{\nu_x-1}{2}}^{i_0+\frac{\nu_x-1}{2}} \sum_{j'=j_0-\frac{\nu_y-1}{2}}^{j_0+\frac{\nu_y-1}{2}} F_{i',j'}(x_{\text{int}}, y_{\text{int}})}. \tag{4.4.8}$$

Here w is the weight of the ray as it intersects with the screen. The normalisation of the filter contributions makes sure that the ray weight is conserved as it is distributed over the different pixels. The output render of the algorithm is then the sub-image without the ghost pixels, for

$$\frac{\nu_x - 1}{2} \leq i < n_x + \frac{\nu_x - 1}{2}, \quad \frac{\nu_y - 1}{2} \leq j < n_y + \frac{\nu_y - 1}{2}. \tag{4.4.9}$$

Fig. 4.4.2 shows the result of Gaussian reconstruction summing over multiple rays, for various reconstruction filter sizes (ν_x, ν_y). Note that eq. (4.4.8) reduced to bincount reconstruction for $\nu_x = \nu_y = 1$.

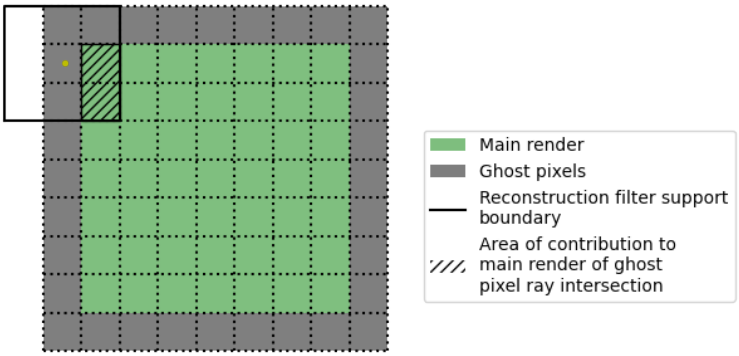


Figure 4.4.3: Demonstration of where ray-intersections of ghost pixels affect the main render at its boundary.

4.4.2. Differentiability

The Gaussian reconstruction indeed yields a differentiable render, but strictly speaking not always a perfect one. As fig. 4.4.1 shows, the filter functions are not differentiable at the boundary of their support. This is a consequence of the correction term $e^{-\alpha\left(\frac{v_x w_x}{2}\right)^2}$ in eq. (4.4.5) which only accounts for *continuity* at the support boundary, but not for differentiability. For α not too small this does not seem to matter in practice, but a fully C^1 filter function, possibly based on the cosine function, might yield more stable optimization based on these gradients. This, however, has not been validated in our experiments.

The choice to add ghost pixels is motivated by the objective to obtain meaningful gradients near the screen boundaries. By the dependency of the amount of ghost pixels on the reconstruction filter size, rays that intersect the ghost pixels will always give some contribution to the main render, see fig. 4.4.3. This contribution makes the movement of a ray between of-the-screen and on-the-screen smooth[8] as captured on the main render. Without the ghost pixels this would yield a discontinuity at the pixels near the boundary if a ray jumps off or on the screen there.

[8] Or at least continuous, given the previous remark on the reconstruction filter continuity.

5

The Optimization pipeline

This chapter discusses the complete optimization pipeline (fig. 5.3.1) for caustic design with a B-spline lens. The differentiable ray-tracer is explained in the previous chapter, this chapter discusses the implemented neural network architectures and the loss function.

5.1. The neural network

This section discusses the general concept of a neural network and the type of network architecture used for our experiments.

5.1.1. Architecture

Several neural network architectures will be considered, which will all have a trivial input of 1. This means that the neural networks will not strictly speaking be used to approximate a function, since the considered domain is trivial. Using non-trivial neural network domains is discussed in the outlook in section 7.2.

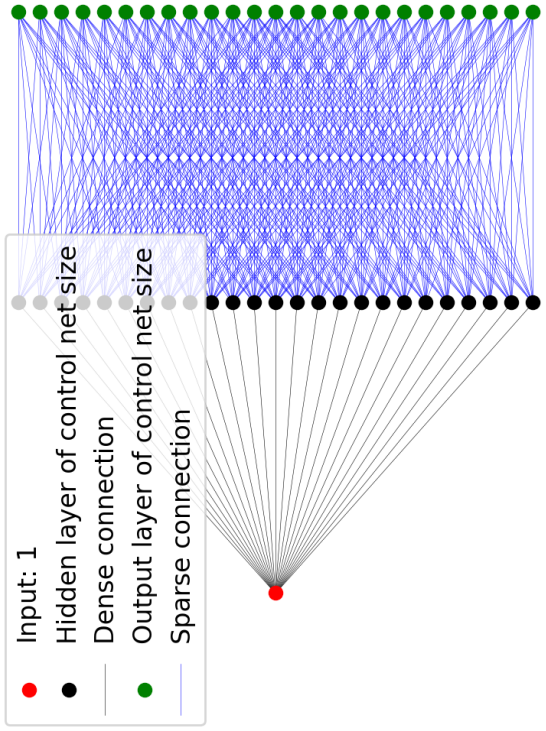
In this configuration the neural network can be thought of as a transformation of the space that is optimized over: from the space of trainable neural network parameters to the space of control point z-coordinate values. The goal of hyper-parameter tuning is then that the design landscape on the trainable neural network parameters yields better training behaviour than the design landscape on the space of control point z-coordinate values.

The considered architectures are:

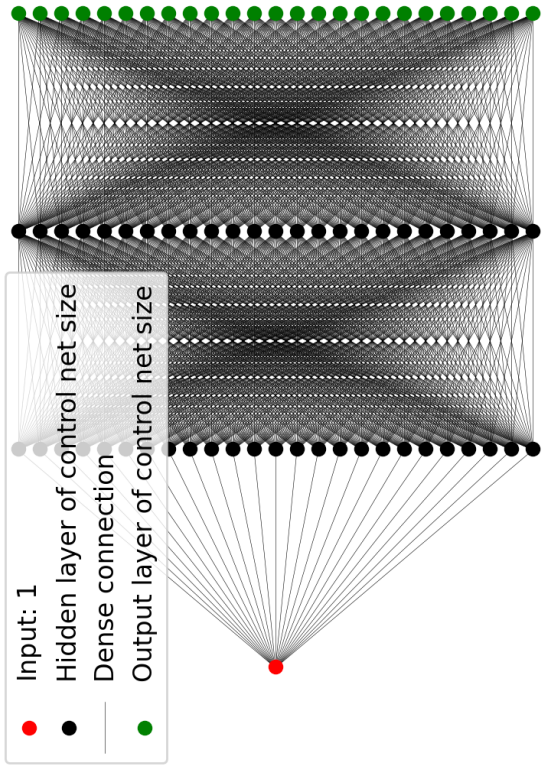
1. No network at all.
2. A sparse MLP where the sparsity structure is informed by the overlap of the B-spline basis function supports on the knot spans. In other words: this architecture aims to precisely let those control points 'communicate' within the network that share influence on some knot span product on the B-spline surface. This yields a layer with the same connectivity as a convolutional layer with kernel size $(2p+1, 2q+1)$,[1] but each connection has its own weight and each kernel its own bias, instead of only having a weight per element of the convolution kernel and one single bias for all kernels. The architecture is shown in fig. 5.1.1a.
3. Larger fully connected architectures are also considered, with a total of 3 layers of control net size. Note that that two consecutive such layers yield many weight parameters: n^4 for a square control net with 'side length' n. The architecture is shown in fig. 5.1.1b.

All activation functions are chosen to be the hyperbolic tangent, a choice which is motivated below.

[1] Using the degree notation convention from section 2.4.2.



(a) Sparse neural network architecture



(b) Dense neural network architecture

Figure 5.1.1: A sparse and a dense neural network architecture for a control net size of $(5, 5)$ and degrees $(2, 2)$.

5.1.2. Control point freedom

Control over the range of values that can be assumed by the control point z-coordinates is important to make sure that the scene stays physical (as mentioned in section 4.1), but also to be able to take into account restrictions imposed on the lens should it become part of a mechanical construction in some application. Note that the restriction eq. (4.1.5) for the control points being above the lens entrance surface is not critical for a plane wave simulation, since in that case the entrance surface can be moved arbitrarily to the $-z$ direction without affecting the ray-tracing.

Since the final activation function $\tanh$ has finite range $(-1, 1)$, this can easily be mapped to a desired interval $(z_{\min}, z_{\max})$:

$$y_{i,j} \mapsto z_{\min} + \tfrac{1}{2}(y_{i,j} + 1)(z_{\max} - z_{\min}), \tag{5.1.1}$$

which can even vary per control point if desired. Here $y_{i,j}$ denotes an element of the total output Y of the network.[2] The above can also be used as an offset from certain fixed values:

$$y_{i,j} \mapsto f\left(P^x_{i,j}, P^y_{i,j}\right) + z_{\min} + \tfrac{1}{2}(y_{i,j} + 1)(z_{\max} - z_{\min}). \tag{5.1.2}$$

The resulting B-spline surface approximates the surface given by $f(x, y) + \frac{1}{2}(z_{\max} + z_{\min})$ if $Y \approx 0$. This can for instance be used to optimize a lens that is globally approximately convex/concave. The choice of the hyperbolic tangent activation function accommodates this: since this activation function is smooth around its fixed point 0, when initializing the weights and biases of the network close to 0 there is no cumulative value-increasing effect in a forward pass trough the network, so that indeed $Y \approx 0$ in this case.

For the sake of comparability, in case 1 the optimization is not performed directly on the control point z-coordinates. In stead, for each control point a new variable for optimization is created, which is passed through the activation function and the correction as in eqs. (5.1.1) and (5.1.2) before being assigned to the control point.

5.2. The loss function

When optimizing a lens for an explicit reference image $\mathcal{I}_{\text{ref}} \in \mathbb{R}_{\geq 0}^{n_x \times n_y}$, the loss function is given by

$$\text{Loss} : \left(\mathbb{R}_{\geq 0}^{n_x \times n_y}\right)^2 \to [0, \infty), \qquad \text{Loss}(\mathcal{I}; \mathcal{I}_{\text{ref}}) = \frac{1}{\sqrt{n_x n_y}} \left\| \tfrac{1}{\Sigma \mathcal{I}} \mathcal{I} - \tfrac{1}{\Sigma \mathcal{I}_{\text{ref}}} \mathcal{I}_{\text{ref}} \right\|_2 = \sqrt{\frac{1}{n_x n_y} \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} \left(\frac{\mathcal{I}_{i,j}}{\Sigma \mathcal{I}} - \frac{\mathcal{I}_{\text{ref},i,j}}{\Sigma \mathcal{I}_{\text{ref}}} \right)^2} \tag{5.2.1}$$

The normalization of the images makes it so that the optimization is only dependent on the relative pixel values of the images (and the relative brightness of the sources if there are multiple). This effectively eliminates one lighting parameter, and the target image does not have to be scaled a priori to have a well-behaved optimization.

The above loss function, which is used to produce the results for this thesis, includes the target image $\mathcal{I}_{\text{ref}}$ explicitly, and compares it pixel-by-pixel to the produced render. It is however also possible to create a loss function that defines the optimized render implicitly. This can be done by constructing the loss function in such a way that it penalizes certain undesirable properties of the render, for instance a large standard deviation over the pixels.

[2] Y can be considered a vector as in fig. 5.1.1 or as a matrix in the formulas used here. This does not matter, as long as a consistent translation between the two is used.

5.3. The complete pipeline

Fig 5.3.1 shows the complete optimization pipeline. A forward pass consists of evaluating the neural network with an input of 1, creating a render with differentiable ray-tracing trough the lens defined by the network output, and computing the loss by comparing the render with a reference image.

Back-propagation is handled automatically by PyTorch. For an example of the computational graph of the forward pass used for back-propagation, see appendix C. The trainable neural network parameters are then updated based on the obtained gradients by the Adam optimizer, which concludes one iteration of the optimization.

5.4. Iterating

The convergence criterion Loss $< \varepsilon$ for some small $\varepsilon > 0$ is quite standard for loss-based optimization problems. It is however not used for the results in the following chapter. One reason for this is that it is not a-priori clear for this optimization pipeline which value of ε corresponds to a satisfactory result, which also depends on the application of the optimization.

Furthermore, for a sufficiently small ε this criterion might never be reached at all by the optimization. One reason for this could be that the optimization is stuck in a local minimum of the design landscape, the likelihood of which depends on the optimization starting point. Another reason could be that with the chosen hyper-parameters the optimization might not be able to reach such a small loss at all, *i.e.* the global minimum of the loss is larger than ε. Therefore at this stage it is better to define a maximum number of iterations as a stopping criterion.

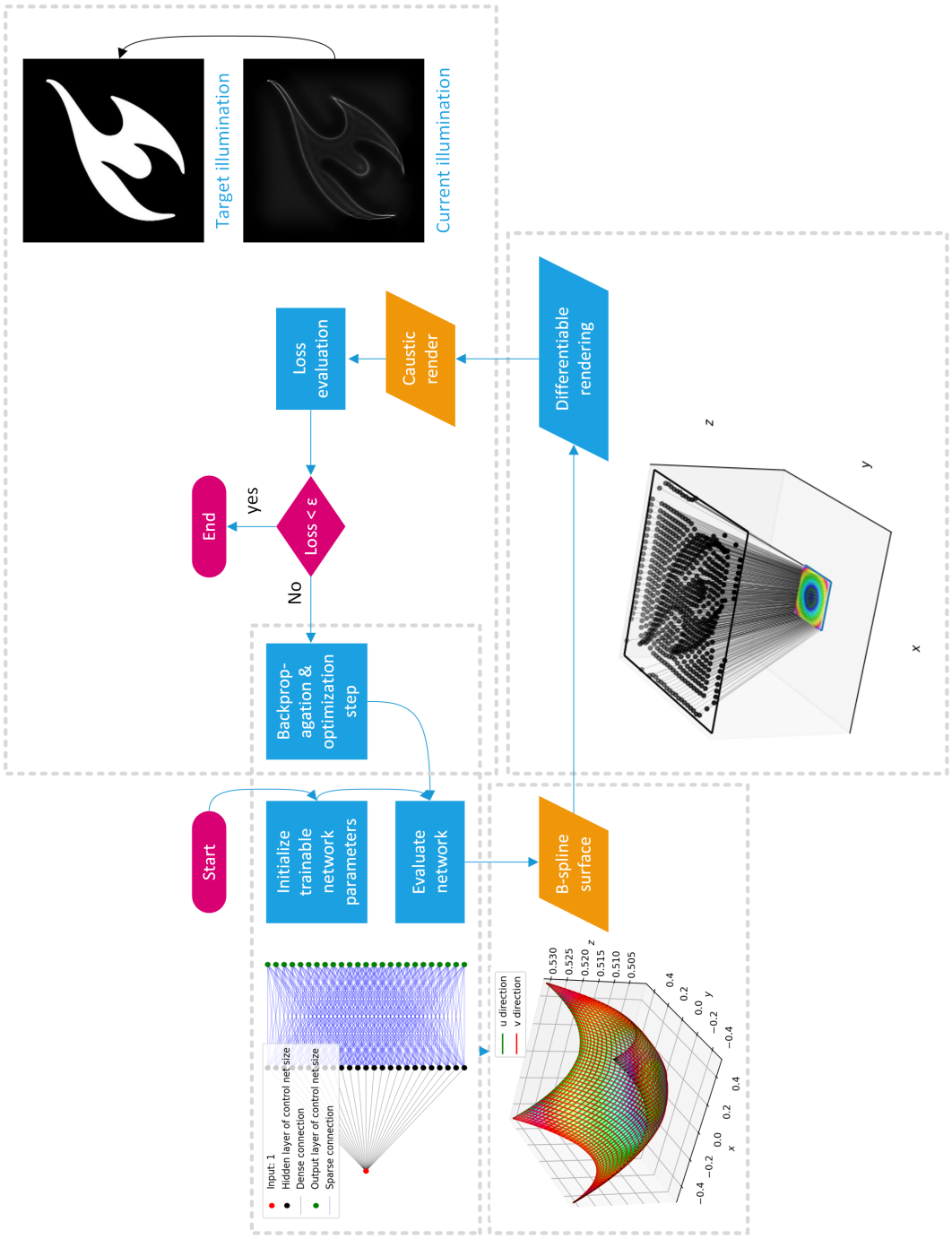


Figure 5.3.1: The complete optimization pipeline.

6

Results

In this section several results produced with the optimization pipeline discussed in the previous chapters are displayed and discussed. The details of the implementation can be found in appendix B. The implementation mainly uses PyTorch, a Python wrapper of Torch [Collobert et al., 2002],

None of the optimizations performed for this chapter took more than a few hours to complete, on a `HP ZBook Power G7 Mobile Workstation` with a `NVIDIA Quadro T1000 with Max-Q Design` GPU.

Most of the results in this chapter have been verified with *LightTools* [Synopsis, 2021], an established ray-tracing software package in the optics community. Lens designs were exported to LightTools in the form of a point cloud, which was then interpolated back into a continuous surface by LightTools. The LightTools simulations always use 10^6 rays, and since plane waves are not supported by LightTools these were approximated by a point source at $z = -10^6$.

Units of length are mostly unspecified, since the renders are invariant under uniform scaling of the optical system. This is a reasonable property of the model in the regime where the lens details are orders of magnitude larger than the wavelength of the incident light. Furthermore, the renders are directly proportional to a scaling of all ray weights and thus the source power, so the source and screen power also need no unit specification. Note that relative changes do have a non-trivial effect, like changes to the power proportion between sources or the distance proportions of the optical system.

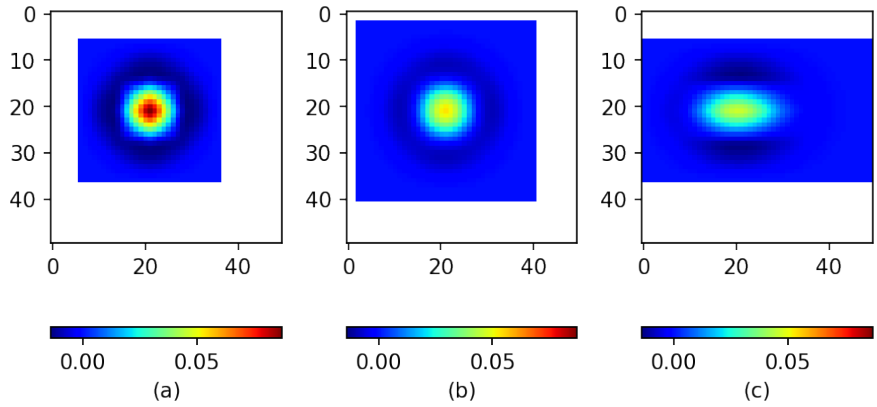


Figure 6.1.1: Gradients of a render of a plane wave through a flat lens (parallel sides), with respect to the z-coordinate of one control point. The zeros are masked with white to show the extend of the influence of the control point. These renders differ by: (a): degrees $(3, 3)$, reconstruction filter size $(3, 3)$, (b): degrees $(3, 3)$, reconstruction filter size $(11, 11)$, (c): degrees $(5, 3)$, reconstruction filter size $(3, 3)$.

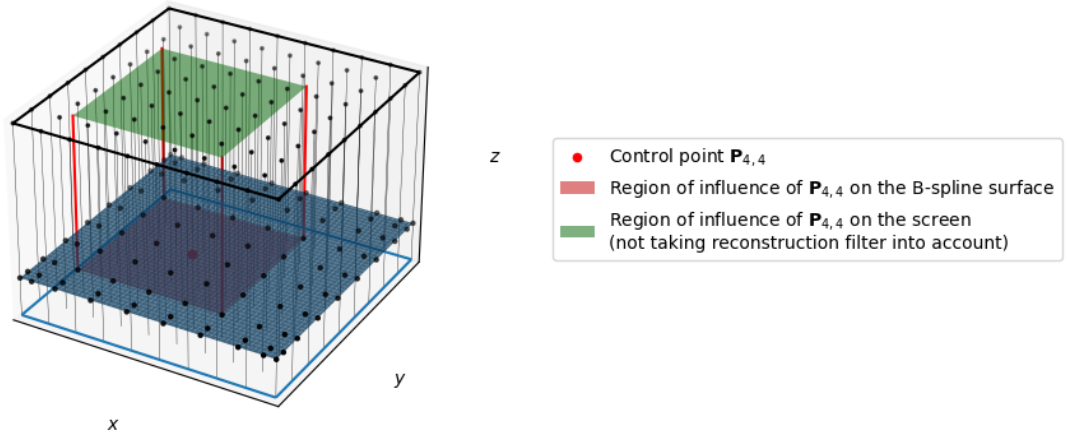


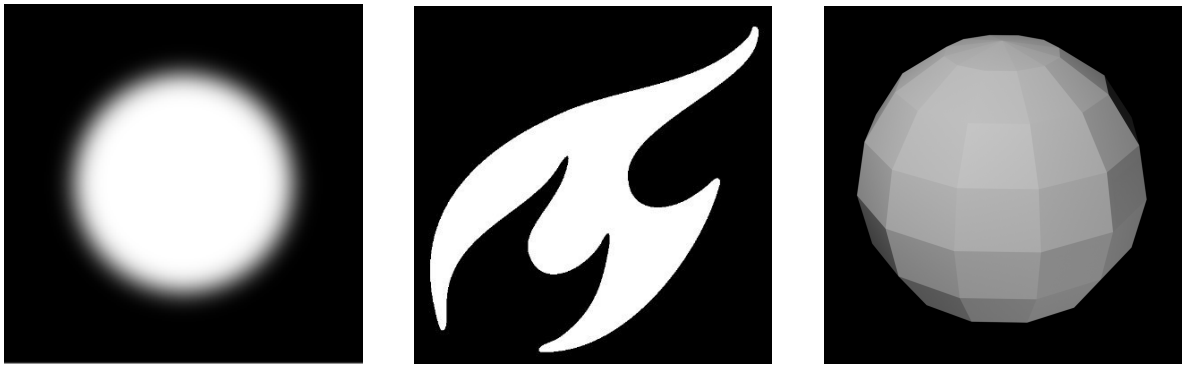
Figure 6.1.2: Demonstration of how one control point influences the render in the case of a flat lens with B-spline degrees $(3, 3)$ and a plane wave source.

6.1. Render derivatives with respect to a control point

This section gives a simple first look at the capabilities of the implemented differentiable ray-tracer: computing the derivative of a render with respect to a single control point. Obtaining this data is necessarily inefficient in the current PyTorch implementation, since this requires a forward mode automatic differentiation pass which is not currently (fully) supported by PyTorch. Therefore these derivatives are computed with pixel-wise back-propagation.

Fig. 6.1.1 shows the derivative of a render of a plane wave through a flat lens for various B-spline degrees and reconstruction filter sizes, and fig. 6.1.2 shows what one of these scenes looks like. The overall 'mountain with a surrounding valley' structure can be understood as follows: As one of the control points rises up, it creates a local convexity in the otherwise flat surface. This convexity has a focussing effect, redirecting light from the negative valley region towards the positive mountain region.

Noteworthy of these render derivatives is also their total sum: (a) -1.8161×10^{-8}, (b) 3.4459×10^{-8}, (c) 9.7095×10^{-5}. These small numbers with respect to the total illumination of the render of about 93 indicate conservation of light; as the control point moves out of the flat configuration, at first the total amount of power received by the screen will not change much. This is to be expected from cases (a) and (b) where the control point does not affect rays that reach the screen on the boundary pixels. But for all cases, all rays intersect the lens at right angles, and around $\theta = 0$ the slope of Schlick's approximation (fig. 2.1.2b) is very shallow, indicating very little decrease in refraction in favor of reflection.



(a) The circular tophat target illumination. (b) The TU flame target illumination. (c) The faceted ball target illumination.

Figure 6.2.1: The various target illuminations used in this section.

6.2. Sensitivity to initial state and neural network architecture

As with almost any iterative optimization procedure, choosing a proper initial guess of the solution is crucial for reaching a satisfactory local/global minimum in the design landscape (section 2.5.3). For training neural networks this comes down to how the weights and biases of the network are initiated. In this section we look at three distributions, see fig. 6.2.1: the circular top hat distribution, the TU Delft logo and an image of a faceted ball. For some experiments black padding or Gaussian blurring is applied to these images. We design lenses to produce these distributions from a plane wave, given various neural network architectures (introduced in section 5.1.1) and parameter initializations.

6.2.1. Circular top hat distribution from plane wave

Fig. 6.2.2 shows the progress of the loss over 1000 iterations for various neural network architecture and parameter initialization combinations. For the other parameters in these simulations see table D.1. For a few moments during the training the resulting free-form surfaces and rendered caustics are shown in figs. 6.2.3 to 6.2.7. Uniform here means that the initial trainable parameter values are sampled from a very small interval: $U\left(\left[-10^{-4}, 10^{-4}\right]\right)$, except for the no-network case; this is initialized with all zeros.

A first notable difference is between the random and uniformly initialized sparse neural networks. The uniformly initialized neural network performs much better and even no network performs better. This is probably because the uniformly initialized cases converge to a better (local) minimum than the randomly initialized case. Of course it could happen that the random initialization lands in a very favourable spot in the design landscape, but intuitively this seems very unlikely.

Another nice property of the uniformly initialized cases is their preservation of the symmetry present in these setups. As fig. 6.2.3 shows, here this leads to much simpler lenses, which are probably much less sensitive to manufacturing errors due to their relative lack of small detail. Interesting to note here is that if the sparse network is initialized with all parameters set to 0, then its optimization is identical to the no-network case. This is because in this case only the biases in the last layer achieve non-zero gradients.

Clearly the dense network with its $\sim 10^8$ parameters performs best. This large network has no drawbacks with respect to the sparse network in terms of memory usage or computation time, since these are still insignificant with respect to the use of these resources by the ray-tracer. The fully connected network will be used for all following optimizations in the results.

A full grid-search within the hyper-parameter space that defines some class of network architectures could reveal where in the increase of the architecture complexity diminishing returns for the optimization of these lenses arises, which could be part of follow-up research.

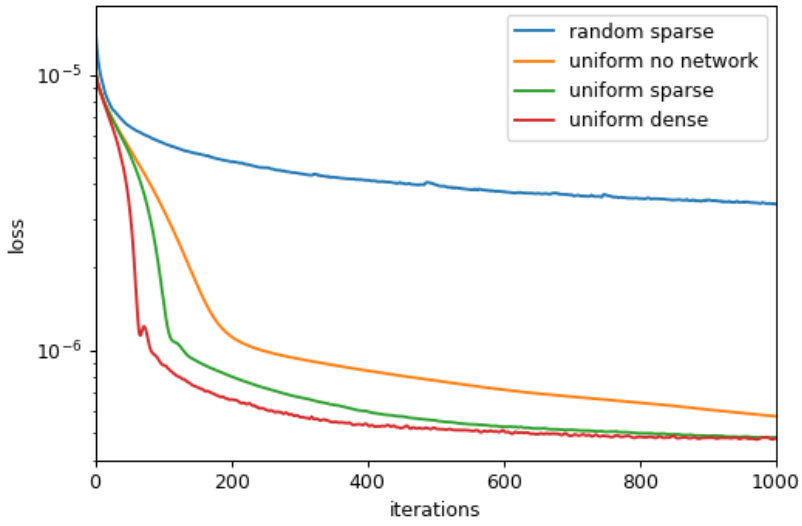


Figure 6.2.2: Loss progress over the iterations for various pipeline-setups for forming a tophat distribution from a plane wave.

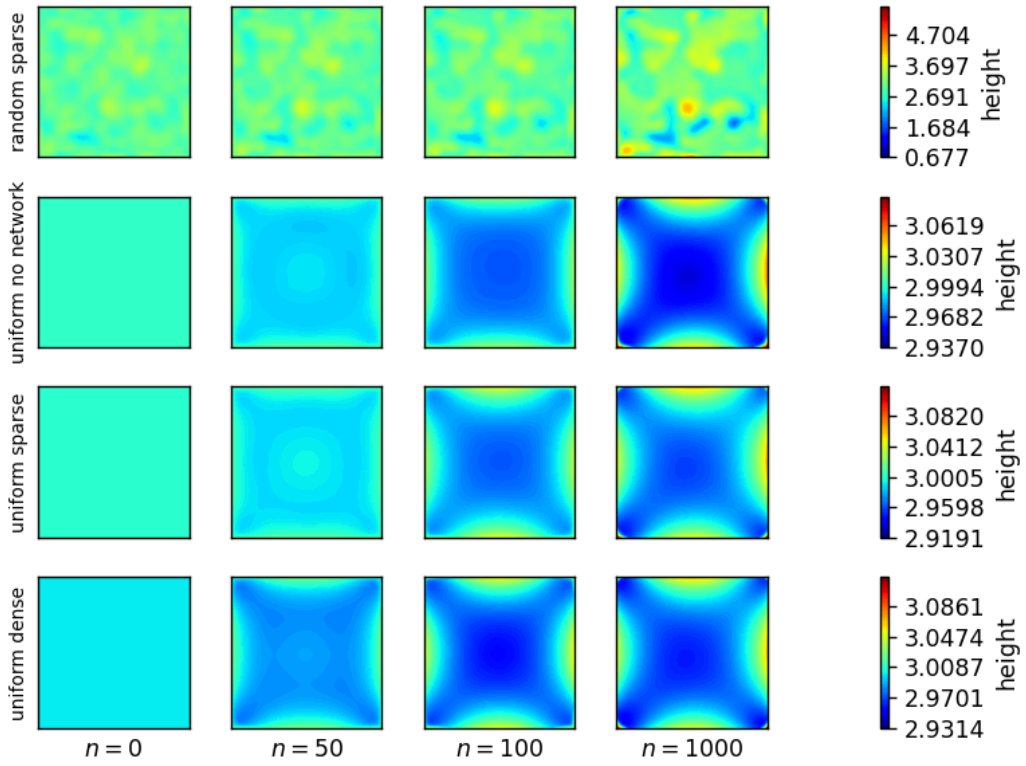


Figure 6.2.3: The lens height field after initialization ($n = 0$), and $n = 50, 100$ and 1000 iterations respectively, for different network architectures (section 5.1.1) and network parameter initializations (section 6.2.1).

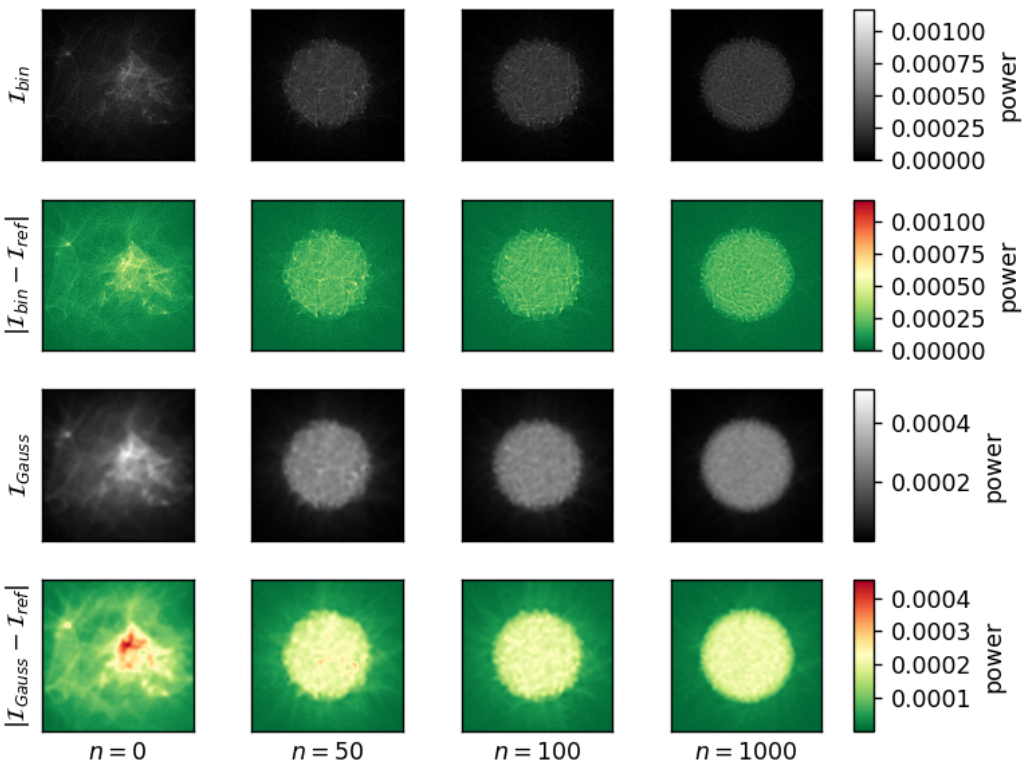


Figure 6.2.4: Renders and pixel-wise errors in the optimization progress of a random lens with a sparse network towards a circular tophat illumination.

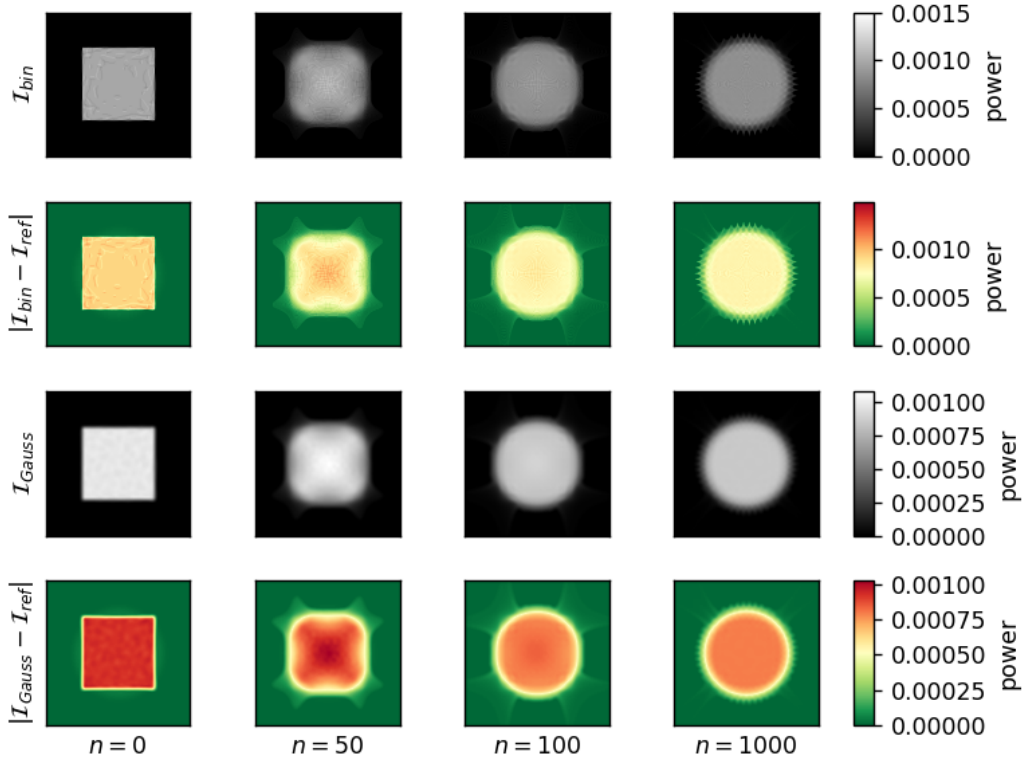


Figure 6.2.5: Renders and pixel-wise errors in the optimization progress of a flat lens with a sparse network towards a circular tophat illumination.

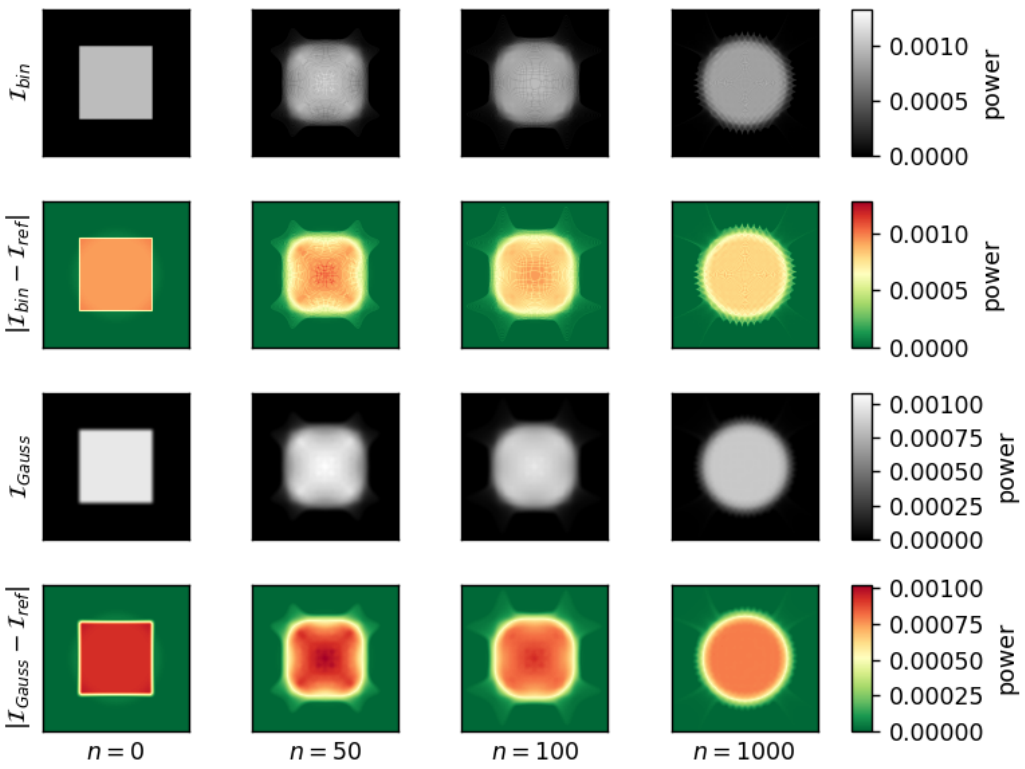


Figure 6.2.6: Renders and pixel-wise errors in the optimization progress of a flat lens without a network towards a circular tophat illumination.

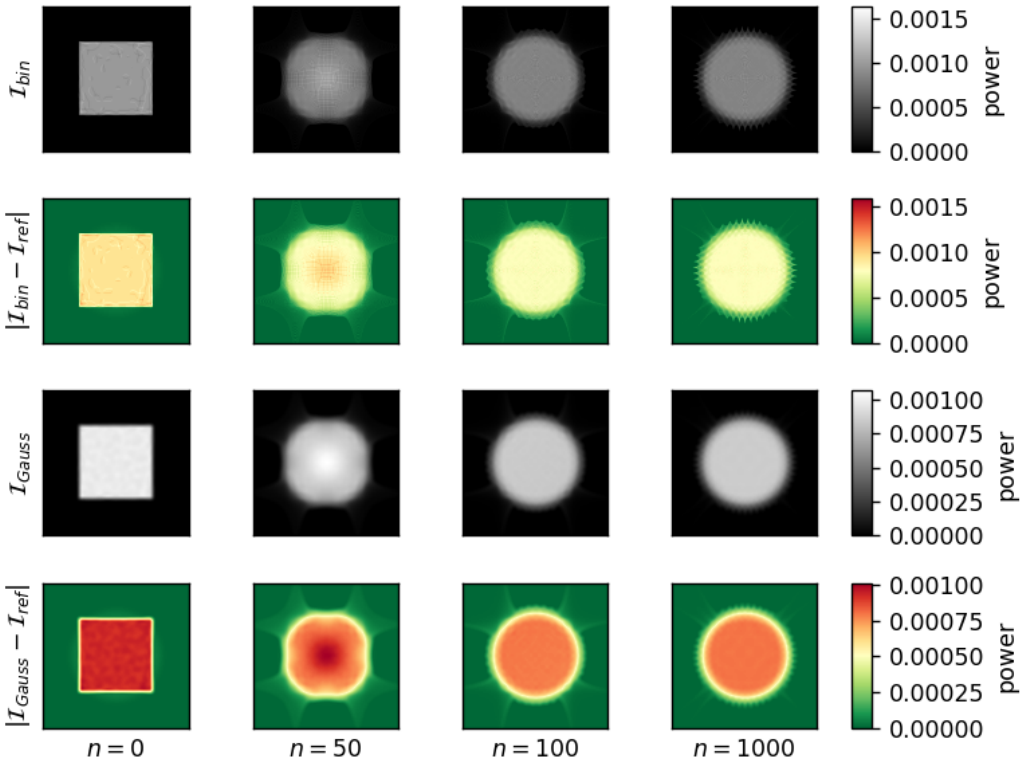


Figure 6.2.7: Renders and pixel-wise errors in the optimization progress of a flat lens with a dense network towards a circular tophat illumination.

6.2.2. TU flame and faceted ball from plane wave

In what follows we consider complex target distributions: the TU Delft flame (for a complex shape) and a faceted ball (for a target with various brightness levels). Here we still use the plane wave illumination, but lenses are now optimized for various magnifications, see table 6.1. These magnifications are defined as the scaling of the screen size with respect tot the smallest screen size $(0.64, 0.64)$. To give the lenses a good initial guess, control point offset functions f (section 5.1.2) of a concave spherical shape are chosen such that the initial caustic is roughly a bounding box of the target caustic. From now on only the dense neural network architecture introduced in section 5.1.1 will be used, as this architecture performed the best in the previous results. The other parameters of these optimizations are shown in table D.2.

The final renders and corresponding LightTools verifications are shown in figs. 6.2.9 and 6.2.10 respectively. These figures show that the optimization pipeline can handle these more complex target illuminations quite well. The LightTools renders do predict some artefacts within the caustic which the implemented ray-tracer does not, especially in the TU flame magnification 1 case. By eye, based on the LightTools renders, one would probably rate these results in the exact opposite order than as indicated by the losses shown in fig. 6.2.8.

A potential explanation of the increase in loss with the magnification factor in fig. 6.2.8 could be that the bigger the screen is, the further rays have to travel from the lens to the edges of the screen. This can be seen from the diverging nature of the ray bundle in the magnification 3 and 5 cases in fig. 6.2.12. This results in a larger sensitivity of the render to the angle with which a ray leaves the screen. This in turn gives larger gradients of the render with respect to the control points. Therefore the optimization takes larger steps in the neural network parameter space, possibly overshooting points that result in a lower loss.

The magnification 3 and 5 LightTools renders also show artefacts at the screen boundaries. A possible explanation for this is that the way the B-spline surfaces are transferred to LightTools is inaccurate at the surface boundaries.[1] This is because surface normals are inferred from fewer points on the B-spline surface at the boundary than in the middle of the surface by LightTools.

Furthermore, a lot of rays miss the screen (in the implemented ray-tracing simulation), because the target illuminations are black at the borders, so rays near the screen boundary will be forced off the extended screen (see section 4.4) by the optimization. Once rays are off the screen, they no longer contribute to the loss function. This especially occurs in this relatively simple zero-etendue case, where each point on the B-spline surface maps to a single point on the screen (or the extended plane around it). This guarantees that when rays miss the screen, the patch on the B-spline surface these rays originate from has no influence on the render and thus the loss function. This does not necessarily mean that this patch is idle for the rest of the optimization, since this patch can be in the support of a basis function which corresponds to a control point that still affects rays that hit the screen. Thus the probability of getting idle lens patches with this setup decreases with the B-spline degrees, since these determine the size of the support of the B-spline basis functions.

The above might even lead to oscillatory behaviour, with rays alternatingly just hitting and just missing the screen. This calls for a loss function that incorporates all rays, also the ones that miss the screen.

Fig. 6.2.11 shows the height field of the optimised B-spline lens surfaces. A densely varying colormap chosen since the deviations flat or smooth concave shape are quite subtle. This is due to the large lens exit angle sensitivity of the ray-screen intersections, due to the fact that the ratio lens size to screen size is large with respect to ratio lens size to screen distance.

Magnification	screen size	$f(x,y)$	starting shape type
1	$(0.64, 0.64)$	$\frac{1}{2}$	flat
3	$(1.92, 1.92)$	$\frac{1}{2} + 8 - \sqrt{8^2 - x^2 - y^2}$	concave
5	$(3.20, 3.20)$	$\frac{1}{2} + 4 - \sqrt{4^2 - x^2 - y^2}$	concave

Table 6.1: The screen size and control point offset function f (section 5.1.2) used per magnification in the TU flame and faceted ball optimizations (distances in centimeters).

[1] Assuming only rays from the B-spline surface boundaries reach the screen boundary area.

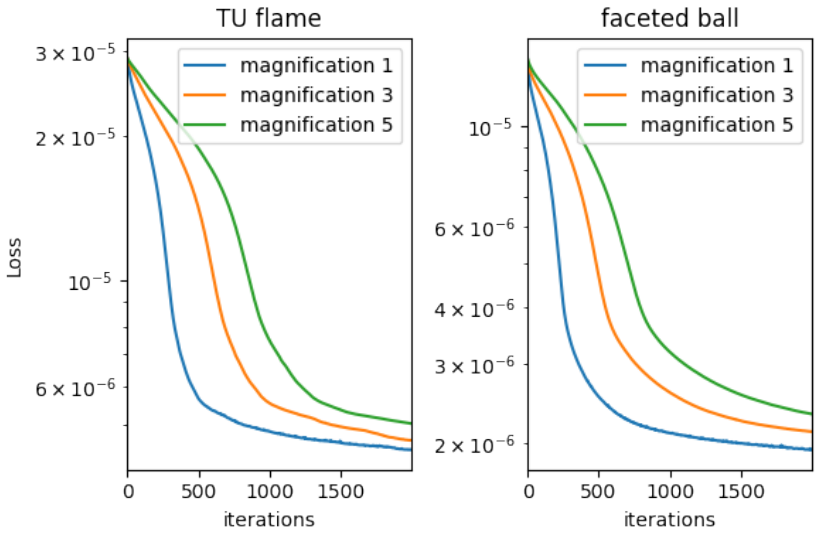


Figure 6.2.8: Loss progress for the various magnifications and target distributions.

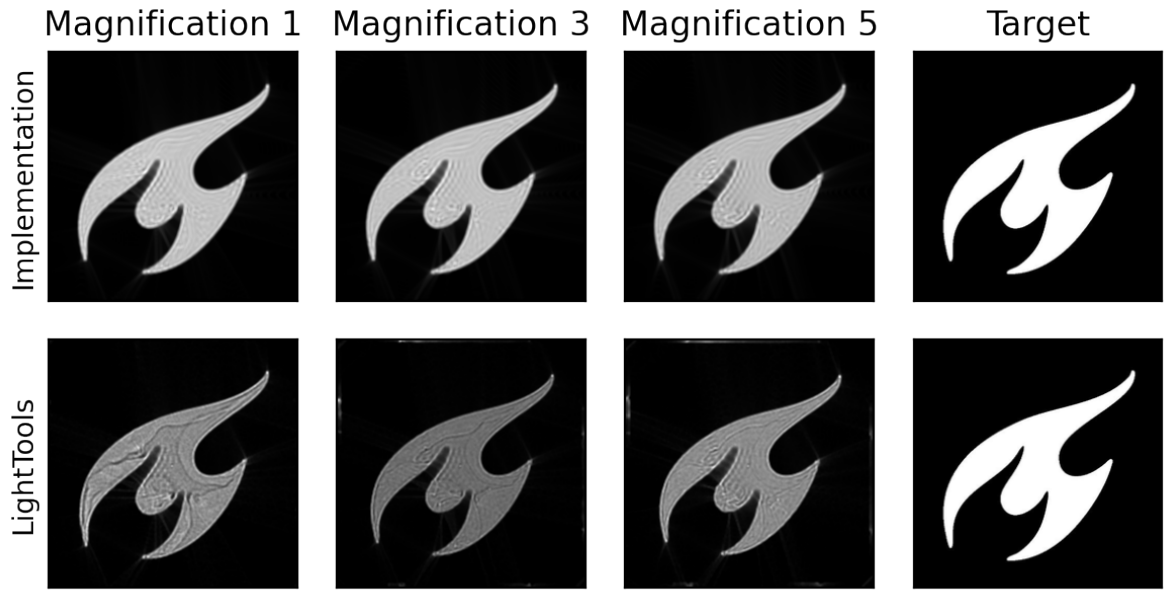


Figure 6.2.9: Implementation and LightTools renders of the TU flame target from the final lens design of the optimization.

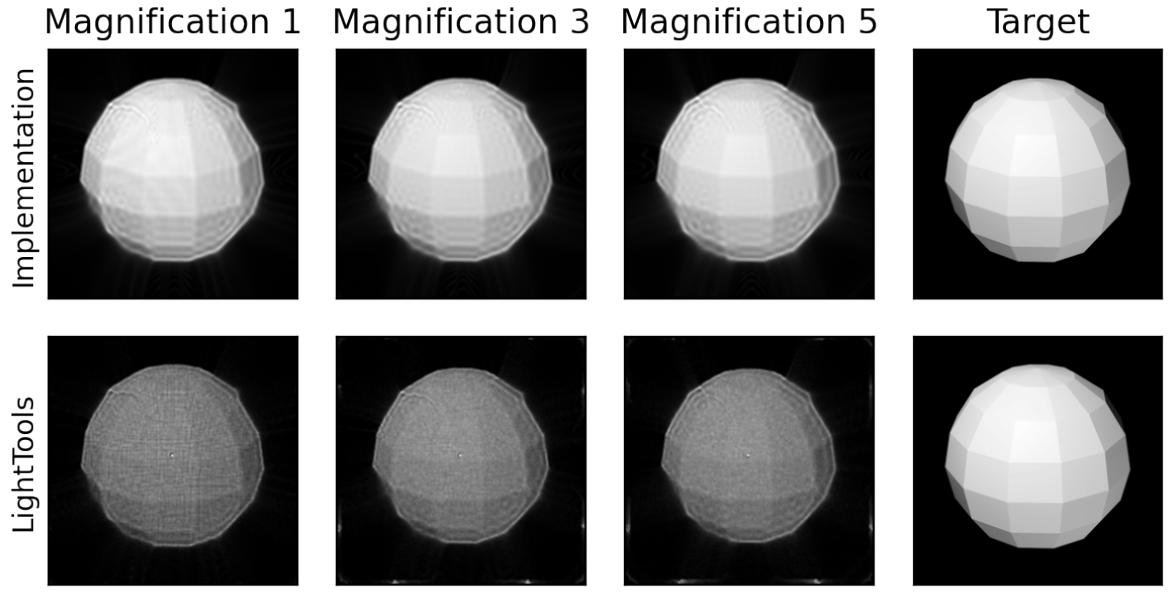


Figure 6.2.10: Implementation and LightTools renders of the faceted ball target from the final lens design of the optimization.

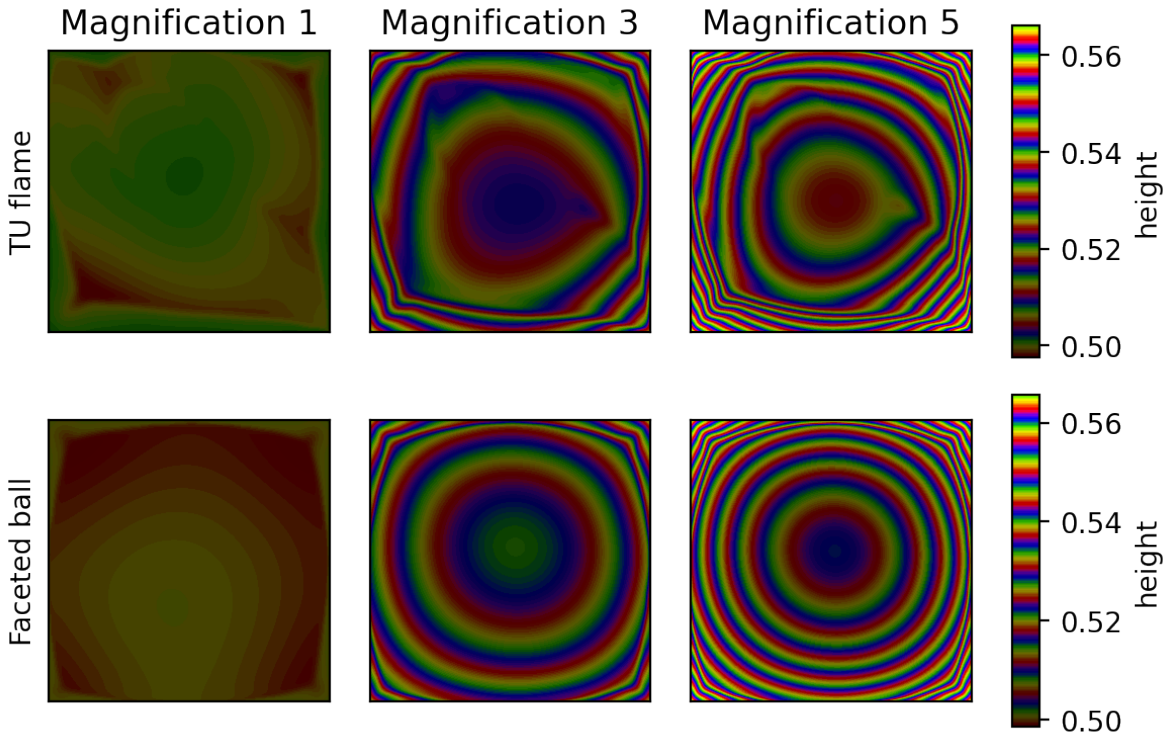


Figure 6.2.11: The lens designs for the different magnifications and two target distributions.

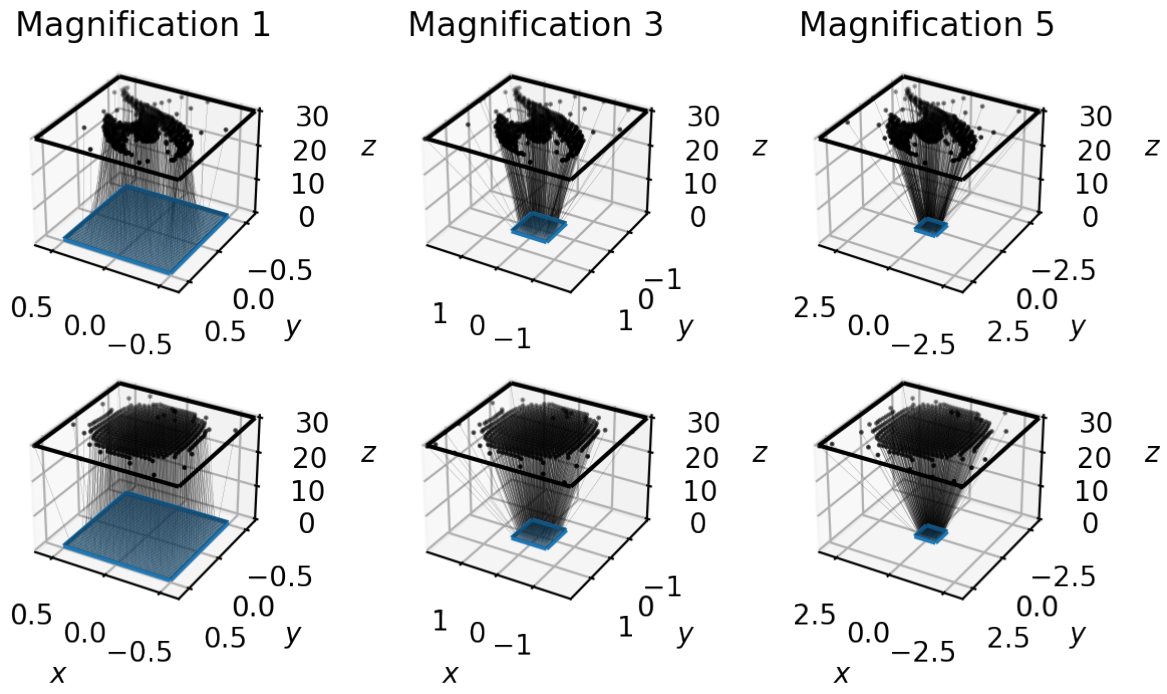


Figure 6.2.12: 25×25 traced rays through the final lens designs for the different magnifications and two target distributions.

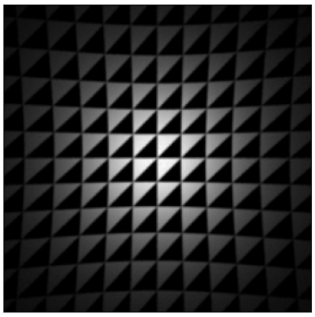


Figure 6.2.13: Render of a point source trough a flat lens, where only half of the triangles are checked for intersection. The curved lines are caused by the refraction making the mapping from the lens exit surface to the screen non-linear; if both surfaces are equipped with polar coordinates then angles are preserved but the radii-mapping is non-linear. The radii mapping and thus the entire mapping (in Euclidean coordinates) is linear in the limit of a lens with zero thickness.

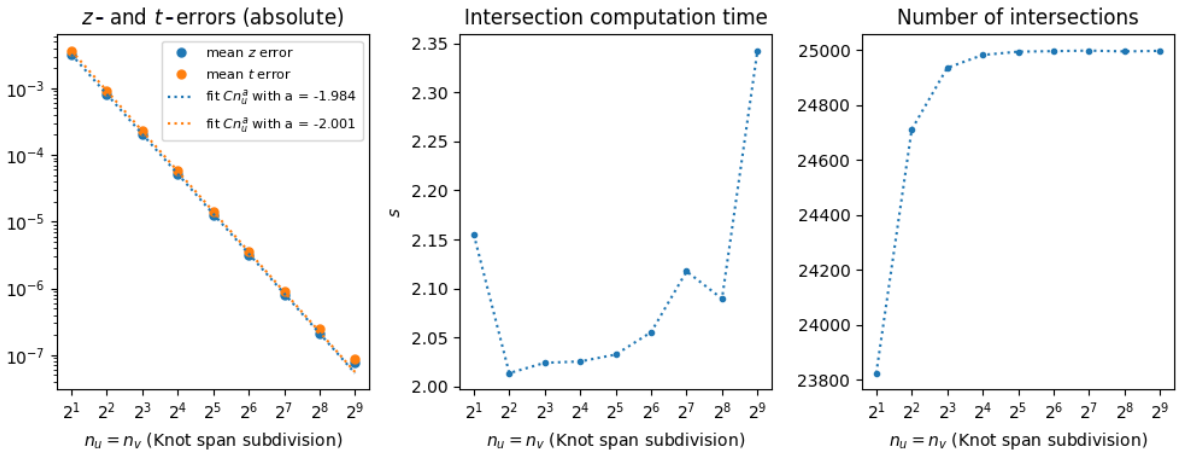


Figure 6.2.14: Performance results of the implemented B-spline-ray intersection algorithm for 25000 rays and multiple knot span subdivision values $n_u = n_v$.

6.3. B-spline intersection algorithm

This section shows some performance results of the intersection algorithm between a B-spline surface and a ray (section 4.3.2, with the recursive bounding box tests).

A first result is shown in fig. 6.2.13: a render produced by modelling a point source shining through a flat lens, where only half of the triangles in the mesh are checked for intersection. This demonstrates how the triangles map onto the screen in this simple case.

We are interested in the convergence behaviour of the algorithm as we let $n_u, n_v \to \infty$ for the knot span subdivisions. For simplicity we look at $n_u = n_v = 2^k$ for $k = 1, 2, 3, ...$ Fig. 6.2.14 shows the mean z- and t-error, the computation time and the number of intersections for a scene whose details are shown in table D.3.

By the analysis in section 4.3.3 it is expected that the z-intersection errors are roughly proportional to n_u^{-2} by $\Delta u_i \propto n_u^{-1}$. This is indeed what we find if we fit the mean z- and also the t-error to the curve $y(n_u) = Cn_u^a$ as seen in fig. 6.2.14. Here C is a proportionality constant and a gives the order of convergence.

As expected the computation time roughly goes up with n_u, since there are more and more triangles to check. The number of intersections going up with n_u could be explained by rays that are directed towards the edges of the B-spline surface: by the changing shape of the triangle mesh as the mesh gets finer, some rays miss all triangles for smaller n_u but later do hit one for larger n_u.

The details of this plot are dependent on the configuration of the modelled scene, but the errors always show roughly second-order behaviour.

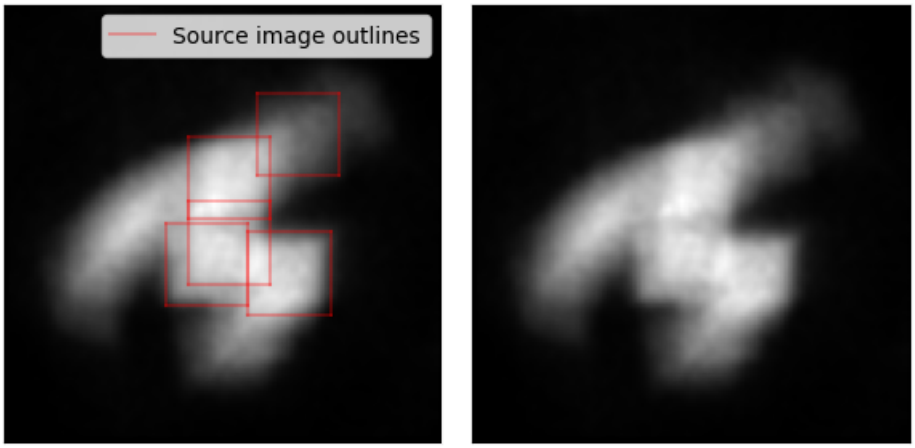


Figure 6.4.1: Indication of images of the source square in the LightTools render with the point source grid.

6.4. Optimization with a point source and a grid of point sources

In what follows we consider an optimization that makes use of the B-spline intersection algorithm. First we design a lens with one point source at $(0, 0, -5)$ with 5×10^5 rays to again form the TU flame. Then after ~ 200 iterations we change the source to an equispaced grid of 25×25 point sources with 10^3 rays each on $[-1, 1] \times [-1, 1] \times \{-5\}$, approximating an extended source with finite etendue. The other (hyper-)parameters of this optimization are shown in table D.4.

The resulting final renders and LightTools verifications can be seen in fig. 6.4.2. Notice here that our final renders are similar to the LightTools renders, indicating that ray-tracing with the implemented B-spline intersection algorithm works properly. Our renders are more blurry, due to the large reconstruction filter size.

The single-source point optimization performed quite well, although the illumination is less uniform than in the plane wave case (figs. 6.2.9 and 6.2.10). This lack of uniformity can be understood from the bumpiness of the lens for one point source in fig. 6.4.3. Based on this one could hypothesise that a smoother lens would work better, but the optimization will not achieve this, because it is stuck in a local minimum. Therefore it could help to modify the optimization to discourage these bumpy lenses, for instance with an additive penalty term in the loss function that penalizes the discrete Laplacian of the control point z-coordinates.

For these (hyper-)parameter values the renders with a grid of point sources approximate the extended source render quite well.

The progress of the loss as seen in fig. 6.4.4 shows that the optimization is still able to improve the loss, even after the transition to the grid of point sources. Interestingly, looking at fig. 6.4.2 again, the optimization seems to adopt the coarse strategy of filling up the target caustic with images of the source square, as shown in fig. 6.4.1.

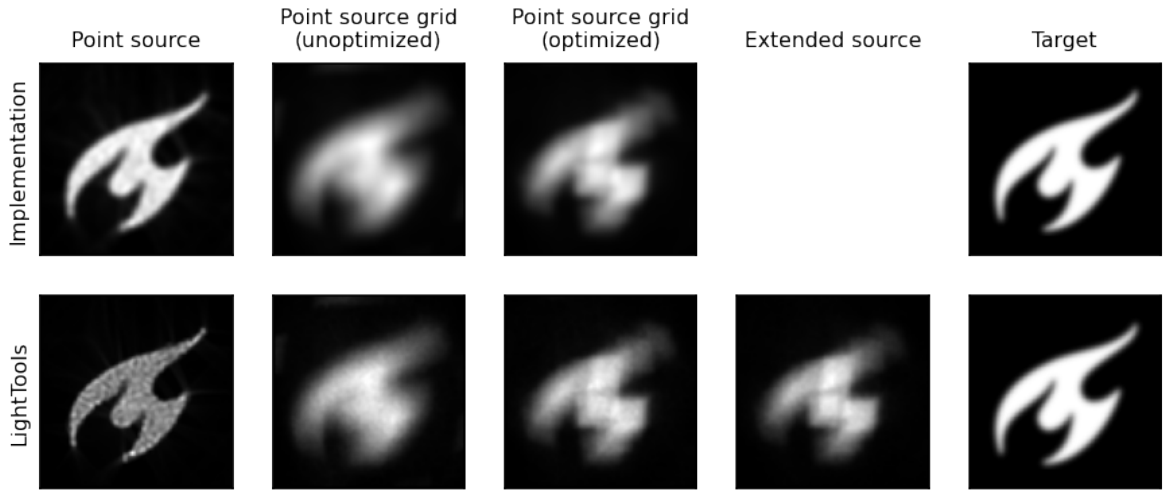


Figure 6.4.2: The final renders of the lens optimizations with point sources and the corresponding LightTools verifications. The extended source is not implemented in our ray-tracer, but is approximated by the point source grid.

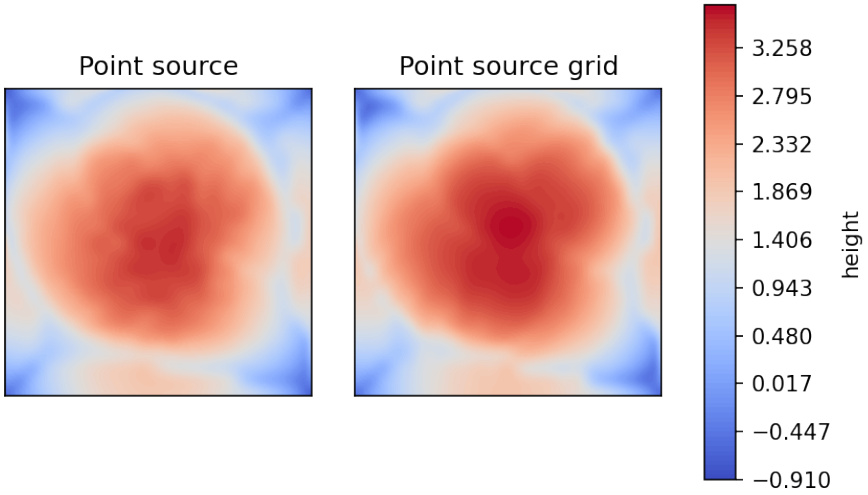


Figure 6.4.3: Height fields of the lenses optimized for the TU flame with point sources.

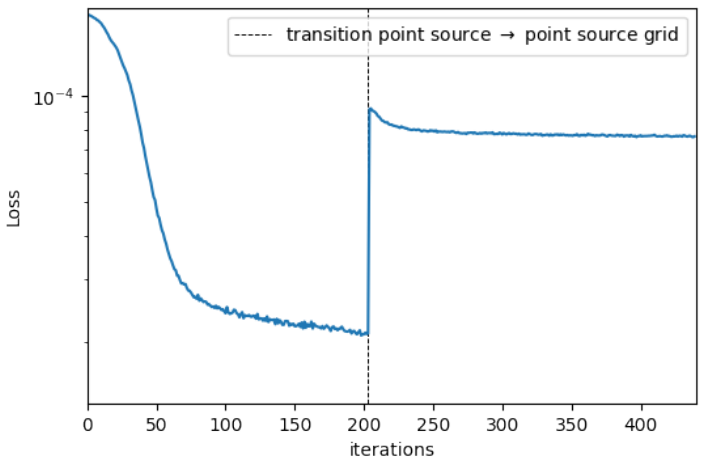


Figure 6.4.4: Loss over the iterations optimizing for the TU flame. The scene is initiated with a point source, and after ~ 200 iterations the point source is replaced by an equispaced grid of 25×25 point sources.

7 Conclusion

7.1. General

The goal of this research was rather broad, namely to investigate whether free-form lenses for caustic design can be designed from B-spline surfaces using differentiable ray-tracing and neural networks. The results show great promise for this technique, especially the plane wave results in combination with the LightTools verifications. As expected optimizations with the grid of point sources and the extended source proved to be more challenging, but the fact that also there the optimization pipeline managed to improve the lenses both in terms of loss and visually shows the potential of this technique in positive etendue optimizations.

7.2. Outlook

The research in this thesis has only scratched the surface of the potential applications of differentiable ray-tracing in optical design. Potential new avenues of research (both in imaging and non-imaging optics) are: different surface parameterizations, multiple free-form surfaces and or lenses, more sophisticated source and light models (*e.g.* wavelength spectra, polarisation) and gradient-index optics. This research could pave the way for the development for powerful design tools in the hands of optics designers.

The subsections below give more specific examples of potential further research.

Differentiable ray-tracing software

A reasonable attempt has been made to make the implementation for this research efficient, but computation did not get a large emphasis in the results since the code at this moment is at a proof-of-concept stage. The research can be continued with either Mitsuba 3 (an extension of Mitsuba 2 that claims to provide the required functionality), some other publicly available differentiable ray-tracer or the one implemented during this thesis work. Some advantages of Mitsuba and the self-implemented one are listed below.

- Mitsuba: Inclusion of polarisation- and wavelength-dependent scattering (color-banding), efficient ray-tracing for general NURBS geometries and beyond, taking internal reflections and lens re-entry into account.
- The current ray-tracer: The narrow-purpose implementation allows for easy adaptation and focused optimization.

Note that all Mitsuba advantages could also be implemented into the current ray-tracer, but that might take quite some time.

Neural network architecture

The extent of the research into various neural network architectures has been fairly limited due to the emphasis on the differentiable ray-tracing itself. Below is a list of possible directions that can be taken for a deeper look into the role of neural networks within this project.

- A grid search in neural network architectures to develop heuristics for the relationship between network architecture and optimization performance.
- Non-trivial inputs, for instance source locations and screen distance.
- More general inverse rendering; training a network that takes a caustic image as input and produces a lens design for this caustic as output. This requires far larger networks and far more training than used in this thesis.
- On-the-fly lens refinement with LR B-splines/NURBS: set up a criterion for locally adding control points to the surface based on local errors in the render. Doing this efficiently might require adapting the neural network architecture to the increased number of output without loss of previous training progress.

Loss function
The loss function defines the design landscape on the pipeline output space, and thus has a huge effect on the optimization behaviour. Therefore a fine-tuned loss function can yield optimization behaviour where the various design priorities are more aligned with a specific goal than they currently are with the current naive RMS loss function. Such priorities could include:

- Light throughput maximization (efficiency), enforced for instance by penalizing rays that miss the screen.
- 'Smoothness maximization', enforced for instance by penalizing the numerical Laplacian of the pixel-wise render error (yielding error diffusion).

Application: lenslet array
The design of multiple free-form lenses could be used to create an optical system that has certain degrees of adaptability in-situ, in the form of a *lenslet array*. This consists of an array of source-lens pairs, which all produce a caustic on the same screen. The resulting illumination is a linear combination of caustic produced by the individual lenses. Many illuminations in the span of these caustics could be produced simply by tuning the brightness of the individual sources, where these brightnessess are the solution to a linear system. The lenses could be designed to create a basis suitable for a given application.

Model generalization: Gradient index lens
The ray-tracing could be generalized to include sub-surface scattering for ray-tracing through a lens with smoothly varying index refraction. The following model could be used for this: the lens is given by a B-spline or NURBS *volume*

$$\mathbf{S} : [0,1]^3 \to \mathbb{R}^4. \tag{7.2.1}$$

Here the function $\mathbf{S}$ has 4 outputs: $\mathbf{S} = (X, Y, Z, n)$, made using three sets of B-spline basis functions and control points in $\mathbb{R}^4$. The first three outputs of $\mathbf{S}$ define a volume in $\mathbb{R}^3$, and the last output of $\mathbf{S}$ defines a smoothly varying refractive index field within that volume.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2005). *Tensorflow: A system for large-scale machine learning*.

Albawi, S., Mohammed, T. A., & Al-Zawi, S. (2018). Understanding of a convolutional neural network. *Proceedings of 2017 International Conference on Engineering and Technology, ICET 2017*, *2018-January*, 1–6. https://doi.org/10.1109/ICEngTechnol.2017.8308186

Berner, J., Grohs, P., Kutyniok, G., & Petersen, P. (2021). The modern mathematics of deep learning. http://arxiv.org/abs/2105.04026

Caflisch, R. E. (1998). Monte carlo and quasi-monte carlo methods. *Acta numerica*, *7*, 1–49.

Cohen, E., Martin, T., Kirby, R. M., Lyche, T., & Riesenfeld, R. F. (2010). Analysis-aware modeling: Understanding quality considerations in modeling for isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, *199*, 334–356. https://doi.org/10.1016/j.cma.2009.09.010

Collobert, R., Bengio, S., & Mari{é}thoz, J. (2002). *Torch: A modular machine learning software library*. Idiap.

Crijns, L. H. (2021). *Pinn inspired freeform design*. Delft University of Technology. https://repository.tudelft.nl/islandora/object/uuid%5C%3A17710753-e574-4439-b944-3e25230492a0

Du, K.-L., & Swamy, ·. M. N. S. (2019). *Neural networks and statistical learning second edition*.

Fairchild, M. D. (2013). *Color appearance models*. John Wiley & Sons, Incorporated.

Fernandez-Maloigne, C., & Robert-Inacio, F. (2013). *Digital color: Acquisition, perception, coding and rendering* (L. Macaire, Ed.). John Wiley & Sons, Incorporated.

Fischer, B., Tadic-Galeb, B., & Yoder, P. (2008). *Optical system design* (2nd). McGraw-Hill Education.

Fowles, G. R. (1975). *Introduction to modern optics (2nd edition)*. Dover Publications.

Gebremedhin, A. H., & Walther, A. (2020). *An introduction to algorithmic differentiation*. https://doi.org/10.1002/widm.1334

Genova, K., Cole, F., Maschinot, A., Sarna, A., Vlasic, D., & Freeman, W. T. (2018). *Unsupervised training for 3d morphable model regression*.

Hopfield, J. J. (1982). *Neural networks and physical systems with emergent collective computational abilities (associative memory/parallel processing/categorization/content-addressable memory/fail-soft devices)*. https://www.pnas.org

Jakob, W., Nimier-David, M., Loubet, G., Speierer, S., Ruiz, B., Vicini, D., & Zelter, T. (2019). *Mitsuba 2 documentation*. https://mitsuba2.readthedocs.io/en/latest/

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., & Yang, L. (2021). *Physics-informed machine learning*. https://doi.org/10.1038/s42254-021-00314-5

Kato, H., Beker, D., Morariu, M., Ando, T., Matsuoka, T., Kehl, W., & Gaidon, A. (2020). Differentiable rendering: A survey. http://arxiv.org/abs/2006.12057

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. http://arxiv.org/abs/1412.6980

Kiser, T., Eigensatz, M., Man, N. M., Bompas, P., & Pauly, M. (2021). *Architectural caustics–controlling light with geometry*.

Kurth, T., Treichler, S., Romero, J., Mudigonda, M., Luehr, N., Phillips, E., Mahesh, A., Matheson, M., Deslippe, J., Fatica, M., Prabhat, P., & Houston, M. (2018). Exascale deep learning for climate analytics. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 649–660. https://doi.org/10.1109/SC.2018.00054

Liu, D. C., & Nocedal, J. (1989). *On the limited memory bfgs method for large scale optimization*.

Loubet, G., Holzschuch, N., & Jakob, W. (2019). Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics*, *38*. https://doi.org/10.1145/3355089.3356510

Meng, X., Li, Z., Zhang, D., & Karniadakis, G. E. (2019). Ppinn: Parareal physics-informed neural network for time-dependent pdes. https://doi.org/10.1016/j.cma.2020.113250

Meyron, J., Mérigot, Q., & Thibert, B. (2018). Light in power: A general and parameter-free algorithm for caustic design. *ACM Transactions on Graphics*, *37*. https://doi.org/10.1145/3272127.3275056

Newman, W., & Phong, B. T. (1975). *Graphics and illumination for computer generated pictures*.

Nguyen, D. T., Meyers, C., Yee, T. D., Dudukovic, N. A., Destino, J. F., Zhu, C., Duoss, E. B., Baumann, T. F., Suratwala, T., Smay, J. E., & Dylla-Spears, R. (2017). 3d-printed transparent glass. *Advanced Materials*, *29*. https://doi.org/10.1002/adma.201701181

Nimier-David, M., Vicini, D., Zeltner, T., & Jakob, W. (2019). Mitsuba 2. *ACM Transactions on Graphics*, *38*, 1–17. https://doi.org/10.1145/3355089.3356498

Noll, R. J. (1976). Zernike polynomials and atmospheric turbulence. *J Opt Soc Am*, *66*, 207–211. https://doi.org/10.1364/JOSA.66.000207

OSA. (2019). *What is etendue, and why is it important?* https://www.optica.org/en-us/events/webinar/2019/what_is_etendue_and_why_is_it_important/

Papas, M., Jarosz, W., Jakob, W., Rusinkiewicz, S., Matusik, W., & Weyrich, T. (2011). Goal-based caustics. *30*.

Patrizi, F., Manni, C., Pelosi, F., & Speleers, H. (2020). Adaptive refinement with locally linearly independent lr b-splines: Theory and applications. *Computer Methods in Applied Mechanics and Engineering*, *369*. https://doi.org/10.1016/j.cma.2020.113230

Pavlakos, G., Zhu, L., Zhou, X., & Daniilidis, K. (2018). *Learning to estimate 3d human pose and shape from a single color image*.

Pharr, M., Jakob, W., & Humphreys, G. (2017). *Physically based rendering*. https://doi.org/10.1016/b978-0-12-800645-0.50001-4

Piegl, L., & Tiller, W. (1995). The nurbs book. https://doi.org/10.1007/978-3-642-97385-7

Piegl, L., & Tiller, W. (1997). *The nurbs book*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-59223-2

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Romijn, L. B. (2021). *Generated jacobian equations in freeform optical design*. www.ridderprint.nl.

Schlick, C. (1994). An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, *13*, 233–246. https://doi.org/10.1111/1467-8659.1330233

Schwartzburg, Y., Testuz, R., Tagliasacchi, A., & Pauly, M. (2014). High-contrast computational caustic design. *ACM Transactions on Graphics*, *33*. https://doi.org/10.1145/2601097.2601200

Sharma, A., Kumar, D. V., & Ghatak, A. K. (1982). *Tracing rays through graded-index media: A new method*.

Simon, C. (2015). *Generating uniformly distributed numbers on a sphere*. http://corysimon.github.io/articles/uniformdistn-on-sphere/

Synopsis. (2021). *Lighttools*. https://www.synopsys.com/optical-solutions/lighttools.html

Vuik, C., Vermolen, F., van Gijzen, M., & Vuik, M. (2017). *Numerical methods for ordinary differential equations* (second edition). VSSD.

Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y., & Sakuma, J. (2018). Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. *CODASPY 2018 - Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, *2018-January*, 127–134. https://doi.org/10.1145/3176258.3176335

Yan, X., Yang, J., Yumer, E., Guo, Y., & Lee, H. (2016). Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision. http://arxiv.org/abs/1612.00814

Yu, D., Seide, F., Li, G., & Deng, L. (2012). Exploiting sparseness in deep neural networks for large vocabulary speech recognition. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 4409–4412. https://doi.org/10.1109/ICASSP.2012.6288897

Yue, Y., Iwasaki, K., Chen, B. Y., Dobashi, Y., & Nishita, T. (2014). Poisson-based continuous surface generation for goal-based caustics. *ACM Transactions on Graphics*, *33*. https://doi.org/10.1145/2580946

Zimmermann, C., & Sauer, R. A. (2017). Adaptive local surface refinement based on lr nurbs and its application to contact. *Computational Mechanics*, *60*, 1011–1031. https://doi.org/10.1007/s00466-017-1455-7

A

Efficient evaluation of the B-spline basis functions

The formula below shows the written out definition of a general fourth degree basis function,[1][2]

$$
\begin{aligned}
N_{i,4}(u) := \quad & \frac{u-u_i}{u_{i+4}-u_i}\frac{u-u_i}{u_{i+3}-u_i}\frac{u-u_i}{u_{i+2}-u_i}\frac{u-u_i}{u_{i+1}-u_i}\mathbb{1}_{[u_i,u_{i+1})}(u) \\
& + \frac{u_{i+2}-u}{u_{i+2}-u_{i+1}}\mathbb{1}_{[u_{i+1},u_{i+2})}(u) \\
& + \frac{u_{i+3}-u}{u_{i+3}-u_{i+1}}\frac{u-u_{i+1}}{u_{i+2}-u_{i+1}}\mathbb{1}_{[u_{i+1},u_{i+2})}(u) \\
& + \frac{u_{i+3}-u}{u_{i+3}-u_{i+2}}\mathbb{1}_{[u_{i+2},u_{i+3})}(u) \\
& + \frac{u_{i+4}-u}{u_{i+4}-u_{i+1}}\frac{u-u_{i+1}}{u_{i+3}-u_{i+1}}\frac{u-u_{i+1}}{u_{i+2}-u_{i+1}}\mathbb{1}_{[u_{i+1},u_{i+2})}(u) \\
& + \frac{u_{i+3}-u}{u_{i+3}-u_{i+2}}\mathbb{1}_{[u_{i+2},u_{i+3})}(u) \\
& + \frac{u_{i+4}-u}{u_{i+4}-u_{i+2}}\frac{u-u_{i+2}}{u_{i+3}-u_{i+2}}\mathbb{1}_{[u_{i+2},u_{i+3})}(u) \\
& + \frac{u_{i+4}-u}{u_{i+4}-u_{i+3}}\mathbb{1}_{[u_{i+3},u_{i+4})}(u) \\
& + \frac{u_{i+5}-u}{u_{i+5}-u_{i+1}}\frac{u-u_{i+1}}{u_{i+4}-u_{i+1}}\frac{u-u_{i+1}}{u_{i+3}-u_{i+1}}\frac{u-u_{i+1}}{u_{i+2}-u_{i+1}}\mathbb{1}_{[u_{i+1},u_{i+2})}(u) \\
& + \frac{u_{i+3}-u}{u_{i+3}-u_{i+2}}\mathbb{1}_{[u_{i+2},u_{i+3})}(u) \\
& + \frac{u_{i+4}-u}{u_{i+4}-u_{i+2}}\frac{u-u_{i+2}}{u_{i+3}-u_{i+2}}\mathbb{1}_{[u_{i+2},u_{i+3})}(u) \\
& + \frac{u_{i+4}-u}{u_{i+4}-u_{i+3}}\mathbb{1}_{[u_{i+3},u_{i+4})}(u) \\
& + \frac{u_{i+5}-u}{u_{i+5}-u_{i+2}}\frac{u-u_{i+2}}{u_{i+4}-u_{i+2}}\frac{u-u_{i+2}}{u_{i+3}-u_{i+2}}\mathbb{1}_{[u_{i+2},u_{i+3})}(u) \\
& + \frac{u_{i+4}-u}{u_{i+4}-u_{i+3}}\mathbb{1}_{[u_{i+3},u_{i+4})}(u) \\
& + \frac{u_{i+5}-u}{u_{i+5}-u_{i+3}}\frac{u-u_{i+3}}{u_{i+4}-u_{i+3}}\mathbb{1}_{[u_{i+3},u_{i+4})}(u) \\
& + \frac{u_{i+5}-u}{u_{i+5}-u_{i+4}}\mathbb{1}_{[u_{i+4},u_{i+5})}(u)
\end{aligned}
$$

[1] Don't worry, I didn't write this out by hand.
[2] Here $\mathbb{1}_{[a,b)}$ is the indicator function of the interval $[a,b)$.

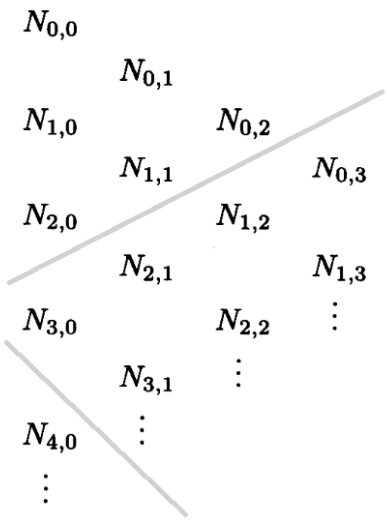


Figure A.0.1: Dependency tree for the construction of higher order B-spline basis functions. $N_{p,i}$ depends on $N_{p-1,i}$ and $N_{p-1,i+1}$. The grey lines enclose the region where non-zero's propagate if the input u is in the support of $N_{3,0} = \mathbb{1}_{[u_3,u_4)}$. Adapted from Piegl and Tiller, 1997, p. 51.

For a naive implementation the computational complexity of computing $N_{i,p}(u)$ is $\mathcal{O}(2^p)$. The exponential nature can be understood from the recursive formula eq. (2.4.3b). However, if you study the formula above, you will find a lot of reoccurring calculations. Furthermore, once you know in which knot span the input u is, most of the terms above can be neglected because the indicator functions will be 0.

Thus when we want to evaluate all basis functions $\{N_{i,p}\}_{i=0}^{n}$ at a particular input u, the best way to start is to find the corresponding knot span such that $u \in [u_\iota, u_{\iota+1})$. Then, given that the support of basis function $N_{i,p}(u)$ is $[u_i, u_{i+p+1})$, we know that u is in the support of $N_{\iota-p,p}, \dots N_{\iota,p}$, hence $p+1$ different basis functions. And that is only via $N_{\iota,0} = \mathbb{1}_{[u_\iota,u_{\iota+1})}$. This is probably best understood by looking at fig. A.0.1. This brings the computational complexity of evaluating all basis functions at one input down from $\mathcal{O}(2^p n)$ to $\mathcal{O}(p^2)$ (ignoring the search of the location of u in the knot vector).

The above ideas were developed by myself while working on an implementation of NURBS. The literature on NURBS is large, and better implementation practices exist than are discussed here, for instance for evaluating an equispaced grid in the surface domain and taking advantage of equispaced knot vectors.

B

Notes on the implementation

The implementations for this thesis are in GitLab repositories from Alex Heemels, details are below.

B.1. LightTools_verification

This repository contains instructions and a Python script for controlling LightTools programmatically, written by Alex Heemels. The script takes as input a description of the optical system as produced by Bart de Koning's code in NumPy's `.npz` format, and reconstructs from this the scene in LightTools before producing a render by ray-tracing.

B.2. PINN based freeform design (branch: MEP_Geometric_Optics_Bart)

This repository contains the implementations in Python made by Bart de Koning for this thesis. Below is an overview of the main folders.

Work_Lucas : this folder contains the implementations of Lucas Crijns, who worked on PINN inspired free-form design with wave optics as a Bachelor project, see here for more information. The `MEP_Geometric_Optics_Bart` branch was branched of of his `master` branch.

Manim : this folder contains Python scrips for animations for an early presentation on the subject of this thesis. The animations were created with the library Manim, for more information see here.

NURBS : this folder contains the NURBS (and B-spline) surface class implementation in PyTorch (and an early version in NumPy), as an improvement to Lucas' B-spline implementation. The folder also contains various Jupyter Notebooks demonstrating the properties and capabilities of the NURBS implementation, as well as a benchmark test comparing Bart's implementation to the `geomdl.NURBS` Python package.

Mitsuba : this folder contains implementations in Python for programmatically using the differentiable ray-tracer Mitsuba 2. The scripts can construct a scene in Mitsuba 2, including a lens mesh for which the implementation can be found in a different main folder: **Lens**. Render derivatives with respect to geometric parameters were not achieved.

MyRayTrace : this folder contains an early implementation of a non-differentiable ray-tracer with NURBS surfaces, all implemented in NumPy.

MyRayTrace_diff : this folder contains the implementation in PyTorch of the ray-tracer explained in chapter 4, as well as several Jupyter notebooks demonstrating the properties and capabilities of the differentiable ray-tracer implementation.

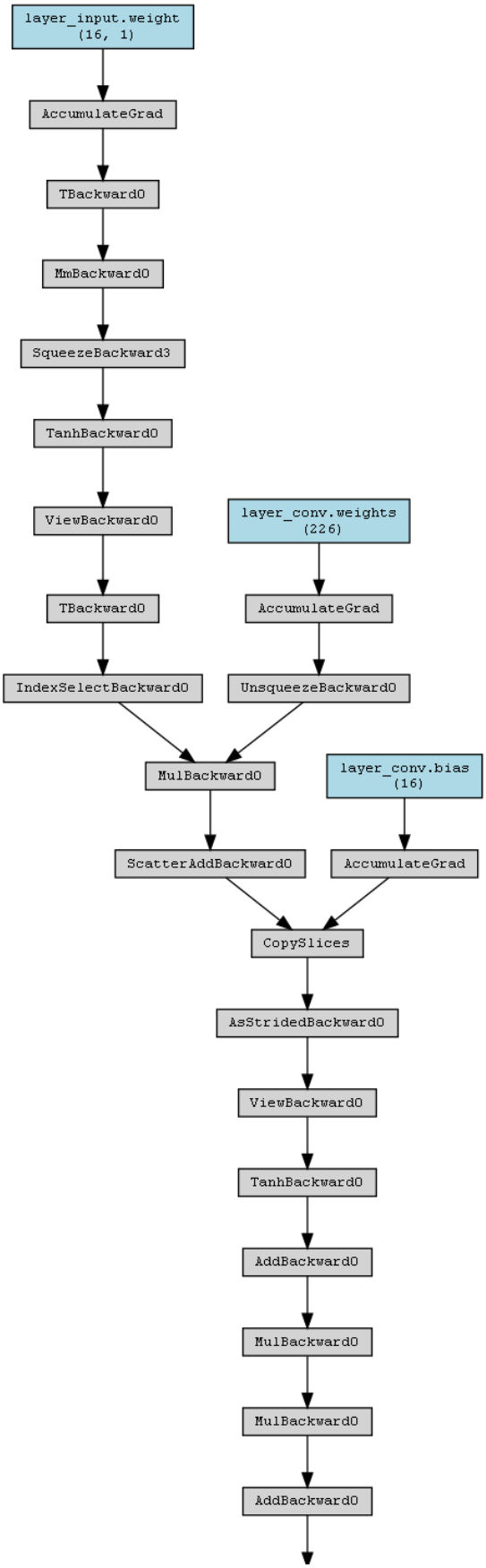
Learning : this folder contains a PyTorch implementation of the used neural network architectures and of the `trainer` class, which incorporates the entire optimization pipeline as discussion in chapter 5. Furthermore, here are all the performed optimizations in Jupyter notebooks.

The folders NURBS, Lens, MyRayTrace_diff and Learning also contain separate scripts for plotting the results of these implementations.

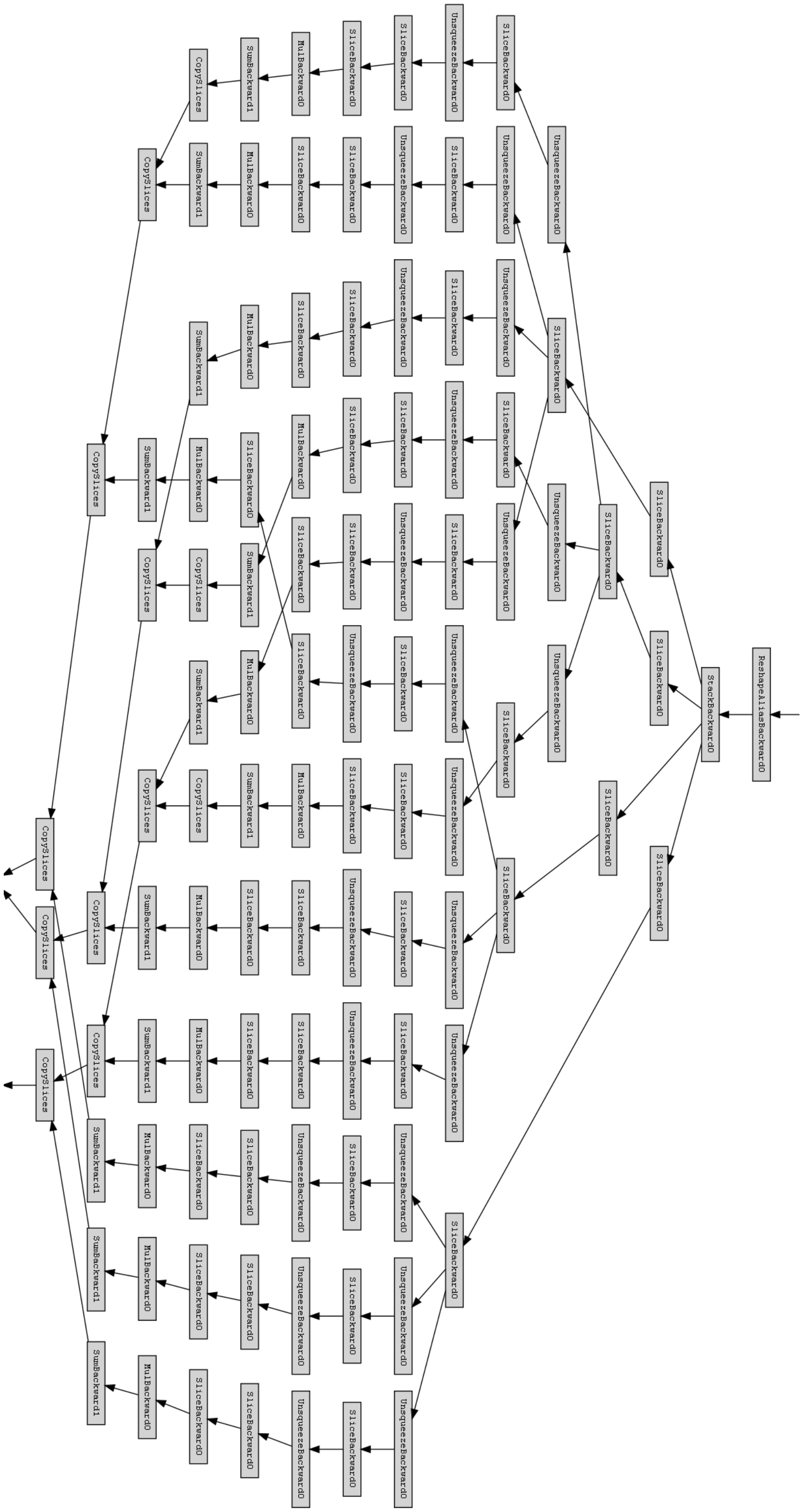
C

Computational graph

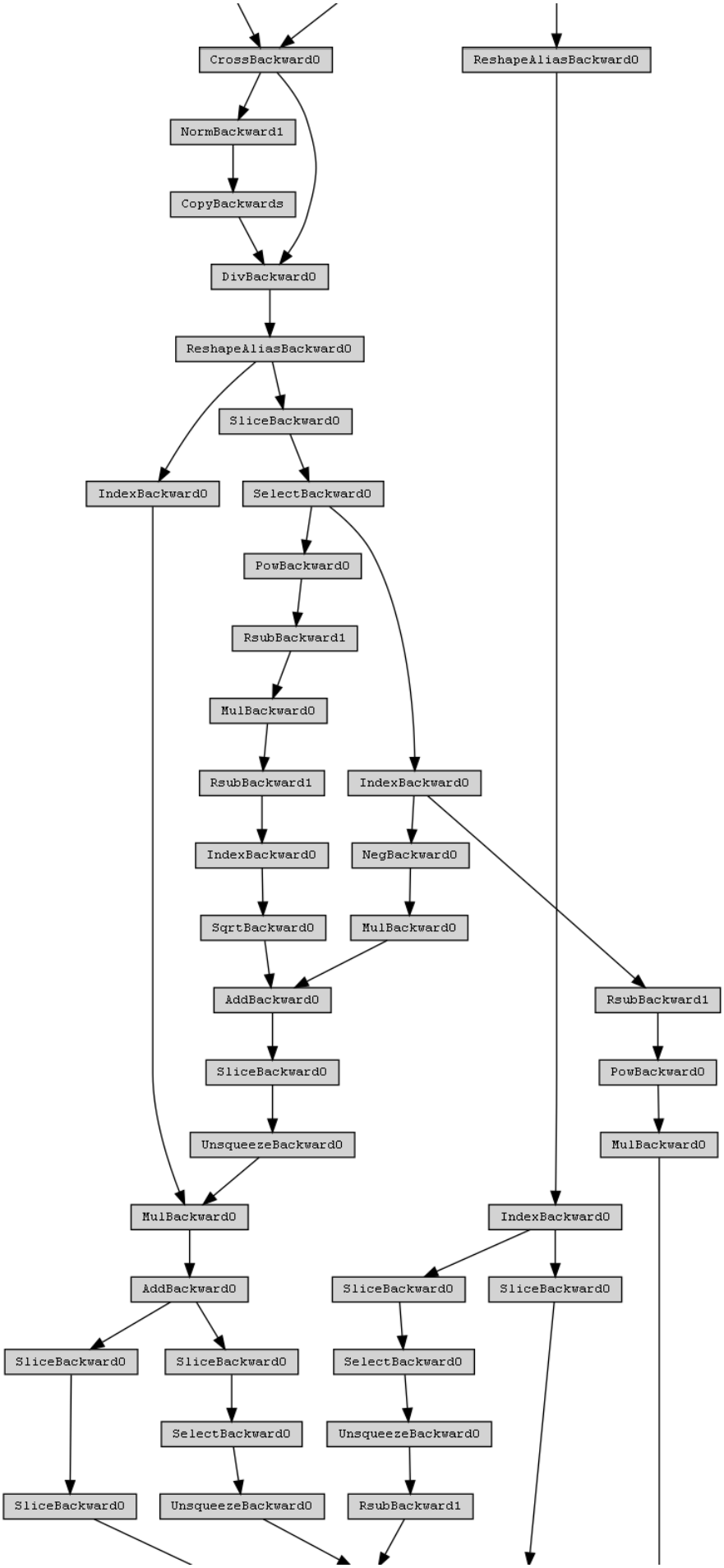
The computational graph shows the relationship between the dependent variable on which backward method is called (the loss) and the independent variables whose gradients are required (the trainable network parameters). Fig. C.0.1 shows this graph for a plane wave simulation with small hyper-parameters. For larger hyper-parameter values, like for the control net size, the graph quickly becomes impractically large for displaying. For simulations involving the intersection algorithm this is even worse.



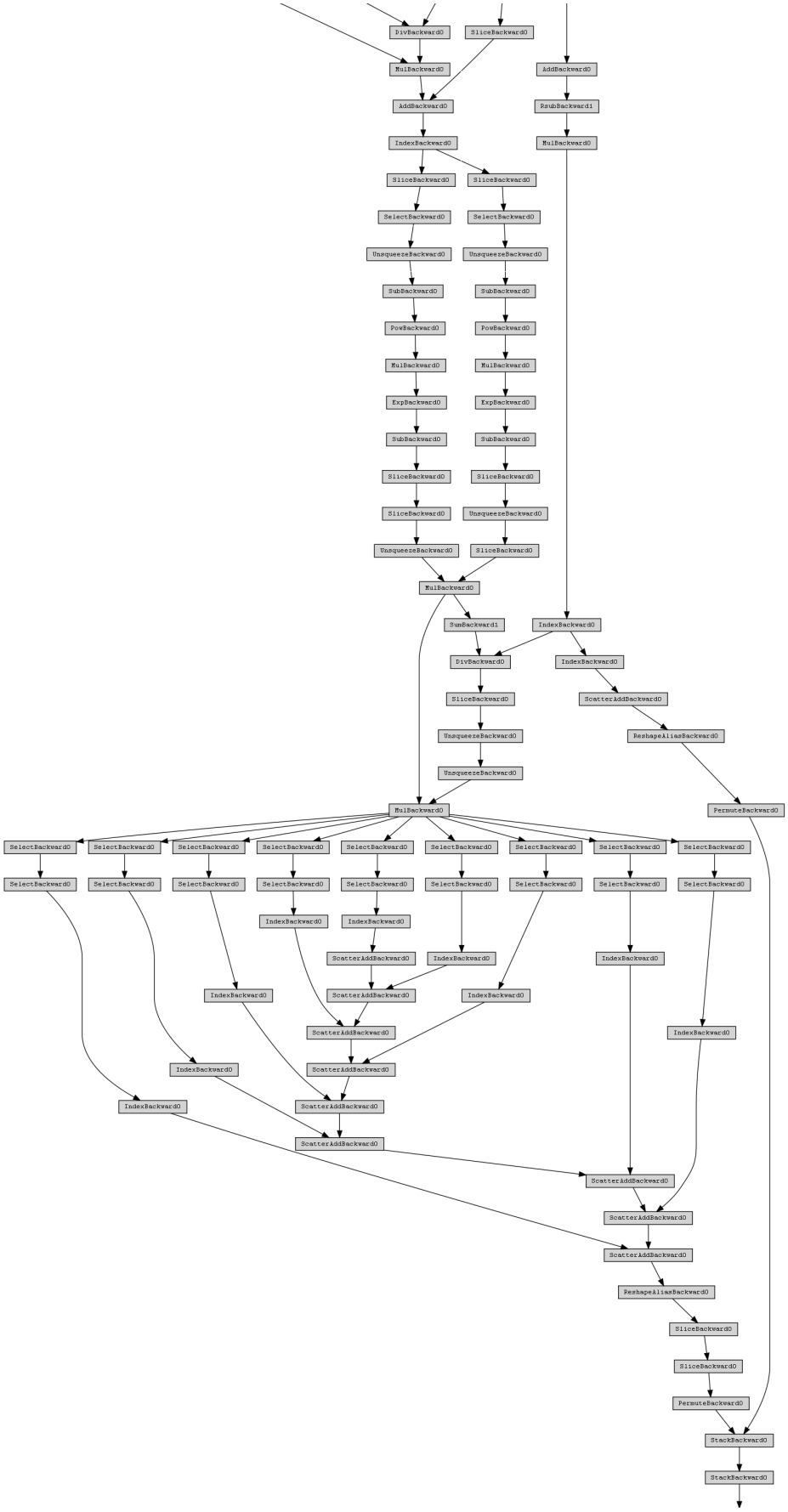
(a) Neural network computational graph section.



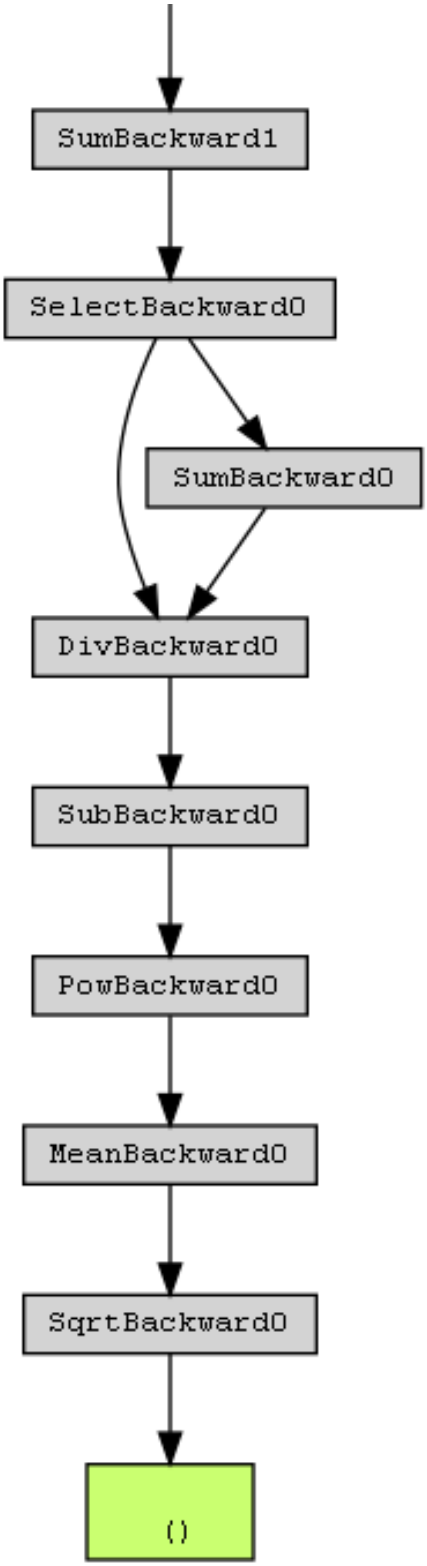
(b) B-spline surface computational graph section (rotated 90 degrees).



(c) Ray-tracing computational graph section.



(d) Image reconstruction computational graph section.



(e) Neural network computational graph section.

Figure C.0.1: Computational graph for a plane wave simulation, roughly subdivided into the sections neural network, B-spline surface, ray-tracing, image reconstruction and loss.

D

Simulation details

This part of the appendix gives all the (hyper-)parameters used for the simulations to obtain the results in shown in chapter 6.

parameter	value
refr. index n	1.5
control net size	$(25, 25)$
degrees	$(5, 5)$
knot vectors	open, equispaced
(r_x, r_y)	$(2, 2)$
NN output min	0
NN output max	6
z_{in}	0

(a) Lens (hyper-)parameters

parameter	value
irradiance	1
ray sample grid size	$(750, 750)$
ray sampler	grid
z_{screen}	9
(R_x, R_y)	$(4, 4)$
resolution	$(250, 250)$
filter size	$(11, 11)$
α	1
learning rate	10^{-4}
other Adam params	default

(b) Source, screen, image reconstruction and Adam optimizer (hyper-)parameters

Table D.1: Fixed (hyper-)parameters in the tophat optimizations.

parameter	value
refr. index n	1.544333973
control net size	$(50, 50)$
degrees	$(10, 10)$
knot vectors	open, equispaced
(r_x, r_y)	$(0.5, 0.5)$
NN output min	-0.1
NN output max	0.1
z_{in}	0

(a) Lens (hyper-)parameters

parameter	value
irradiance	1
ray sample grid size	$(750, 750)$
ray sampler	grid
z_{screen}	30
resolution	$(250, 250)$
filter size	$(3, 3)$
α	1
learning rate	10^{-5}
other Adam params	default

(b) Source, screen, image reconstruction and Adam optimizer (hyper-)parameters

Table D.2: Fixed (hyper-)parameters in the TU flame and faceted ball optimizations (distances in centimeters).

parameter	value
control net size	(15,15)
degrees	$(3, 3)$
knot vectors	open, equispaced
(r_x, r_y)	(5,5)
$f(x)$	$\sqrt{9^2 - x^2 - y^2} - 9$
z_{in}	none, no entrance surface
source loc	(0,0,-6)
$\phi_{\max}$	0.8
no. rays	25000
ray sampler	random uniform

Table D.3: Scene (hyper-)parameters for the B-spline-ray intersection algorithm convergence results.

parameter	value
refr. index n	1.5
control net size	$(25, 25)$
degrees	$(3, 3)$
knot vectors	open, equispaced
(r_x, r_y)	$(5, 5)$
NN output min	-0.7
NN output max	0.7
z_{in}	-2
$f(x)$	$\sqrt{8^2 - x^2 - y^2} - 5$
(n_u, n_v)	$(11, 11)$

(a) Lens (hyper-)parameters.

parameter	value
ray sampler	random uniform
$\phi_{\max}$	$\pi/2$
z_{screen}	25
(R_x, R_y)	$(20, 20)$
resolution	$(100, 100)$
filter size	$(9, 9)$
α	1
learning rate	10^{-3}
other Adam params	default

(b) Source, screen, image reconstruction and Adam optimizer (hyper-)parameters.

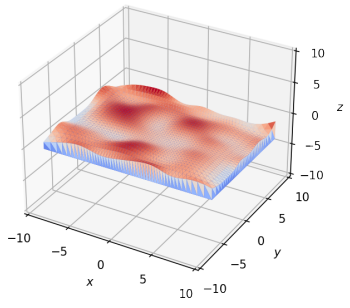
Table D.4: Fixed (hyper-)parameters for the point source (grid) optimizations.

E

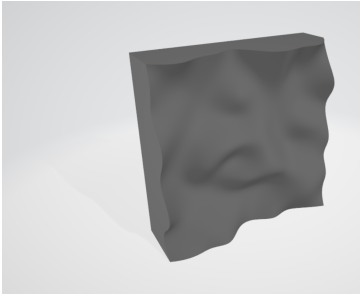
Mitsuba 2

The original plan of the optimization pipeline included Mitsuba 2 [Nimier-David et al., 2019] as a differentiable ray-tracer in stead of my own. Mitsuba 2 is a promising general purpose physically based ray-tracer, currently supporting derivatives with respect to a range of different parameter types like for texture and lighting. However, derivatives with respect to geometric parameters turned out not to be developed up to the extend that that these were usable for this project during this project's run. The figures in this appendix show some preliminary results from working with Mitsuba:

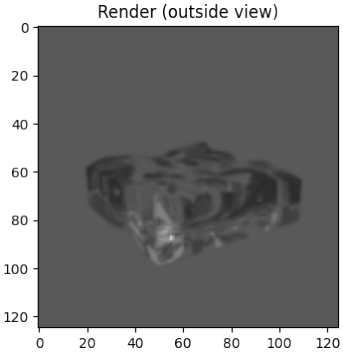
- fig. E.0.1a shows a closed triangle mesh constructed with the PyTorch B-spline implementation, to be exported to Mitsuba.
- fig. E.0.1b shows a lens mesh in Microsoft 3D viewer, from a `.ply` file generated by Mitsuba 2 from lens data loaded from PyTorch.
- fig. E.0.1c Shows a side view of a lens with (wrong) refraction, rendered with Mitsuba 2.



(a) A lens triangle mesh created with the PyTorch B-spline implementation.



(b) A lens ply mesh for Mitsuba 2, shown in Microsoft 3D viewer.



(c) Side view of a lens with (wrong) refraction, rendered with Mitsuba 2.

Figure E.0.1: Lens meshes created for use in Mitsuba 2.