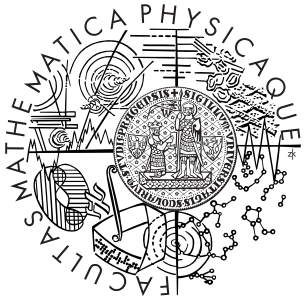
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bohumír Zámečník

Interactive Preview Renderer for Complex Camera Models

Department of Software and Computer Science Education

Supervisor:	Dr. Alexander Wilkie
Study program:	Computer Science
Specialization:	Software Systems, Computer Graphics

Prague 2011

I would like to thank hereby to my supervisor, Dr. Alexander Wilkie, for his valuable advices during the creation of the thesis and for text correction. Also, I would like to thank to my parents for their support of my studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, ... Bohumír Zámečník

Název práce: Interactive Preview Renderer for Complex Camera Models
Autor: Bohumír Zámečník
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Dr. Alexander Wilkie
E-mail vedoucího: alexander@wilkie.at

Abstrakt: Podařilo se implementovat interaktivní renderer umožňující uživatelům zobrazit náhled efektů zobrazování pomocí objektivů, jako jsou hloubka ostrosti, bokeh (rozostřená světla) nebo tilt-shift konfigurace. Je založen na moderní metodě kombinující síly rasterizace na GPU a ray tracingu. V průběhu práce bylo vytvořeno nnoho modelů a interaktivních vizualizací souvisejících jevů. Vytvořena byla též neinteraktivní simulace zobrazování pomocí složeného geometického objektivu, jež vykazuje optické aberace. Rovněž je k dispozici implementace prototypu současných rychlých spreading filtrů. Podařilo se důkladně prostudovat a shrnout principy optického zobrazování, modelů objektivů a metod pro vykreslování hloubky ostrosti v počítačové grafice, jakož i srovnat hlavní myšlenky metod. Díky tomu jsou naznačeny nové možnosti v reprezentaci chování složitých objektivů, jež by šly uplatnit např. pro akceleraci vykreslování.

Klíčová slova: syntéza obrazu, modely kamery, hloubka ostrosti, GPU, image-based ray tracing

Title: Interactive Preview Renderer for Complex Camera Models
Author: Bohumír Zámečník
Department: Department of Software and Computer Science Education
Supervisor: Dr. Alexander Wilkie
Supervisor's e-mail address: alexander@wilkie.at

Abstract: An interactive renderer was implemented that allows users to preview the effects of imaging with lenses, such as depth of field, bokeh (defocus highlights) and tilt-shift lens configurations. It is based on a state-of-the-art method which combines the power of GPU rasterization and ray tracing. Many models and interactive visualizations were created. A non-interactive simulation of a complex geometrical lens model has been made which is able to produce optical aberrations. Also a prototype implementation of recent fast spreading filters is available. A thorough summary of the principles of optical image formation, lens models and depth of field rendering methods used in computer graphics is given along with a comparison of the approaches and new insights. New possibilities of representing the behavior of complex lenses are suggested, which could be employed to accelerate the rendering.

Keywords: image synthesis, camera models, depth of field, GPU, image-based ray tracing

Contents

Chapter 1

Introduction

1.1 Motivation

A long-lasting trend in computer graphics is in striving for more and more realistic synthesized imagery. Much progress has been made in creating complex scene models, material models, light transport methods, etc. In one branch of realism, believable rendering, the goal is to produce images that are hard to decide by humans whether being real of synthesized. On the other hand predictive rendering tries to actually give a physically correct output, not only a good looking one. One of the aspects of achieving realistic look is also in the way how the image of a synthesized scene is captured.

From real life people are accustomed to how they perceive the surrounding world with their own eyes and to the images captured by photographic or movie cameras. In order to deliver such a familiar look in image synthesis we need to focus on modeling of the process of image capture by various cameras.

The most important property of real-world cameras is the ability to focus certain parts of the image and blur the others. The contrast between sharp and unsharp acts for humans as a clue for better perception of the spatial distribution of objects within a scene [36]. Sharp regions also attract one's attention. In addition the depiction of out-of-focus regions can lead to a very pleasing look. All those features of real-world cameras are being heavily exploited by artists to deliver images of great appeal to the viewers.

Since the early years the author has been always amazed by the various optical effects, such as depth of field and bokeh, and their artistic usage. Later this led to becoming an enthusiastic photographer. The original motivation to starting this project was to explore the effects of tilt-shift camera configurations without the usage of real-world view cameras or tilt-shift lenses, both of which are expensive and quite clumsy to operate. The possibility of simulating various lenses was an interesting follow-up.

What is depth of field?

To make a rough idea of what is depth of field and the related optical effects take a look at real-world photographs. Both photographic lenses and human eyes are not capable of imaging the whole scene sharply, except for a single focused plane (see figure 1.1a). A single out-of-focus point light might produce a variously-sized spot,

(a) Depth of field

(b) Bokeh

(c) Tilt-shift

(d) Complex effects

Figure 1.1: Illustration real-world photographs

and overlapping spots cause blur. The blur increases non-linearly with the distance to the focal plane. The region around the focal plane with the blur so small to be perceived as being sharp is called the depth of field.

Very bright single point light sources (including some specular reflections) produce distinguishable bright spots called bokeh in photographic jargon (fig. 1.1b). Their shape and intensity distribution depends on a particular lens and its settings.

Some camera constructions allow to rotate the sensor of the camera and move it laterally so that the focal plane also gets rotated leading to an unnatural, yet pleasing effect, the tilt-shift (fig. 1.1c). It allows for interesting artistic effects such miniature look.

Photographic lenses can be due to technological reasons quite complex optical systems, each one with a slightly different imaging behavior. Also the resulting image depends on the transport of light in the scene itself. The last photo (fig. 1.1d) shows effects such as clipping the bokeh spots by shadowing within the scene and semi-transparent out-of-focus foreground, referred to as partial occlusion.

1.2 Project goals

However interesting are the vast possibilities it was necessary to restrict the scope of the thesis project. The goals were set to the following:

- To study the optical principles leading to image formation in photographic cameras and provide a summary thereof.

- To study the existing approaches and methods of depth of field rendering.
- To choose an interactive method which offers the greatest room for extensibility, while not introducing an excessive amount of systematic artifacts.
- To implement an interactive depth of field renderer based on the chosen method.
- To explore the possibilities of simulating complex camera models and tilt-shift configurations, and also doing so interactively.

The resulting program should be able to serve as a basis for further research.

1.3 Organization

The thesis text is organized in the following way. The second chapter introduces the reader to the basic theory of optics and image formation necessary for understanding the rest. It also gives a thorough overview of camera models used in computer graphics. The next chapter presents the various approaches for depth of field rendering in computer graphics and provides details on the reference ones and those chosen as a foundation for our implementation. Based on studying and comparing many depth-of-field rendering methods some new ideas are suggested here. The fourth chapter describes our implementation of the main interactive renderer, several prototypes of other methods and some additional mathematical models. In the following chapter the results are presented, in particular measurements and the rendered images. After the conclusion the appendices provide some more details on the important algorithms and mathematical methods and a user's guide.

The thesis is accompanied with a CD containing the source codes, binaries and data of the developed software, the thesis in PDF and LaTeX format and the resulting images and measurements.

Chapter 2

Theory of camera models and depth of field

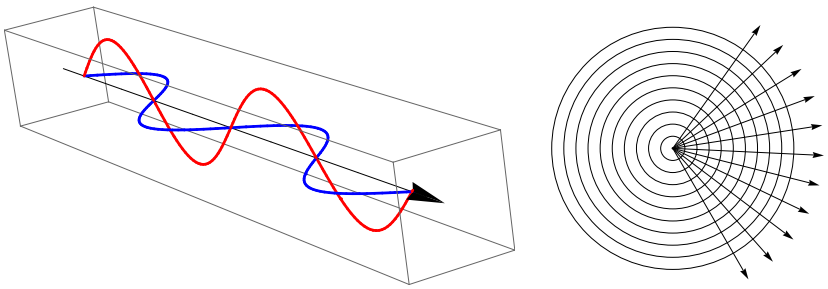
In this chapter we will summarize the most important principles of image formation in optical systems. This theory is a necessary for understanding the methods and algorithms which simulate the image formation using computers. We will quickly review the basic principles of light and optics [15, 66] and then delve into various camera models and some related phenomena.

2.1 Basic optical principles

2.1.1 Light – essence and properties

Light or visible light is electromagnetic radiation in a band visible to human eyes. In essence it is a wave-like oscillation of the electric and magnetic field present in the space (fig. 2.1a). It can travel through the space and carry energy. There exists a duality in light behavior – it exhibits both wave-like and particle-like properties. An elementary particle of light is called the *photon* and it represents the least quantum of light that can interact with other photons or with matter.

From the wave theory point of view a light wave is described by properties such as direction and speed of propagation, frequency or wavelength, intensity and polarization. In computer graphics we usually do not work with single photons or



(a) A single electromagnetic wave with two perpendicular components (electric and magnetic)

(b) Spherical wavefronts and rays of photons from a point light source

Figure 2.1: An illustration of light duality

light waves but rather with beams of light propagating in one direction called rays. A *ray* can be understood as a line perpendicular to the wavefront of a light wave parallel to its direction of propagation (fig. 2.1b). A beam of light containing many light waves or photons can be roughly described by its frequency spectrum.

Geometric optics describes the behavior of incoherent light (a bundle of light waves with generally different frequency, phase or polarization) and its interaction with structures on a scale much larger than its wavelength, eg. refraction or reflection. Wave or physical optics goes deeper and is able to explain even many small-scale effects such as interference or diffraction. Finally quantum optics provides to most comprehensive treatment of light. For image synthesis in computer graphics the framework of geometric optics is usually sufficient. More recently wave optics has been utilized in the field of physically-based and predictive rendering in order to provide greater realism (eg. in [73]).

An important property of geometric optics is that the rays of light interact linearly and independently, since the light is assumed to be incoherent. In practice we can eg. simply integrate or sum the amount of light energy incoming to a point on a sensor. On the other hand within the scope of wave optics multiple light waves can interfere and cancel each other out (which is quite a non-linear behavior).

2.1.2 Light transport

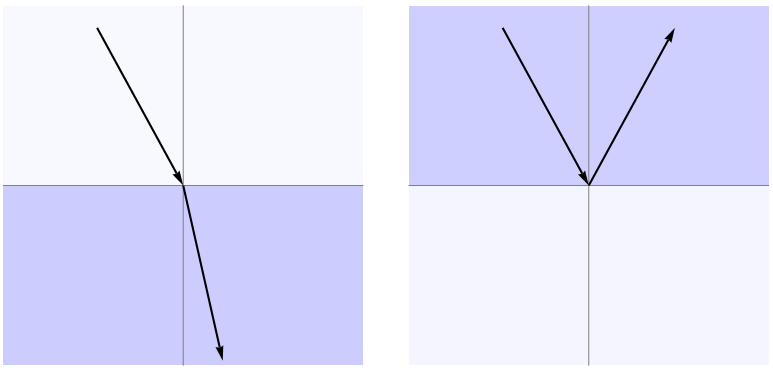
Light can arise due to various physical phenomena. However, in computer graphics the resulting properties of light are usually more relevant than the particular phenomenon of its source. Light can propagate though ideally transparent media without losing its energy. Since the electromagnetic field is present even in vacuum, a part of space containing no matter, the light can propagate in vacuum as well. The speed of light in vacuum c is the maximum speed light can travel. In other media light propagates slower and the velocity v_p depends on the material, light frequency and potentially other properties (such as polarization). The ratio of the light velocity in vacuum to the velocity in a medium, $\eta = \frac{c}{v_p}$, is called the *refractive index*. For real-world materials the refractive index varies depending on light frequency which is called *dispersion*.

When hitting a different material a ray of light can interact in various ways. It can be absorbed, usually being transformed into heat. It can be reflected back into another direction according to the material's reflectance distribution function. It can be refracted, continuing its travel inside the new medium. In reality those and other interactions are combined in various ways, such as partial absorption plus reflection or partial reflection plus refraction, etc.

A system of objects having effect on the light propagation is referred to as an *optical system*. When modeling optical systems such as cameras we will be primarily interested in refraction of light as it has the greatest influence on image formation.

Refraction

When a ray of light hits a planar interface between two media with different refractive indexes it might be in certain circumstances refracted such that its direction changes when entering the next media (fig. 2.2a). It is described by the Snell's law. Let the ray come from one medium with refractive index η_1 to the other medium with refractive index η_2, let θ_1 be the angle of the incident ray to the surface normal



(a) Proper refraction of light going to the denser medium

(b) Total internal reflection inside the denser medium

Figure 2.2: Refraction of light at a planar interface between two transparent media with different refractive index

(pointing to the first media) and θ_1 be the angle of the refracted ray to the other normal pointing the opposite way. The law states that the angles of incident and refracted rays are related to the refractive indexes by the formula:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2, \quad \text{or rewritten as} \quad \theta_2 = \arcsin(\frac{n_1}{n_2}\sin\theta_1)$$

In case of $\frac{n_1}{n_2}\sin\theta_1 > 1$, the arcsine is not defined and the ray is not refracted. Instead a *total internal reflection* (TIR) happens (fig. 2.2b). Since the value of sine is always at most one this can happen only if the rays is coming from a denser medium, ie. $\frac{n_1}{n_2} > 1$. The TIR can occur only within a range on incident angles limited by the *critical angle* $\theta_c = \arcsin\frac{n_2}{n_1}$, for which holds $\frac{n_1}{n_2}\sin\theta_1 = 1$.

The Snell's law can be expressed in several forms working directly with vectors of rays and surface normals rather than just angles [27, 21]. The following is one of them. Let I be the normalized vector of direction of the incoming ray pointing towards the point of intersection of the ray with the interface and N is the normalized surface normal pointing to the half-space of the original medium. Then we can express the direction R of the refracted ray as

$$\cos\theta_1 = N \cdot (-I) = \sqrt{1 - (\sin\theta_1)^2}$$
$$\cos\theta_2 = \sqrt{1 - (\sin\theta_2)^2} = \sqrt{1 - (\frac{n_1}{n_2})^2(1 - \cos\theta_1)^2}$$
$$R = \frac{n_1}{n_2}I + (\cos\theta_2 - \frac{n_1}{n_2}\cos\theta_1)N$$

Note that this law can be applied to arbitrary non-planar surfaces provided they can be locally approximated by a plane.

Wave optics phenomena

Just for completeness we mention two important phenomena from wave optics, interference and diffraction, without going into detail.

Interference can happen in case of coherent light (waves with the same frequency, phase etc.). At such conditions two overlapping waves can sum up in such a way they

cancel or amplify each other. Linearity is preserved only in the complex field but not in the real-valued field of intensity or amplitude. This can be important when simulating eg. lasers or oily streaks on water surface. For depth of field simulation with incoherent light interference cannot happen and can be neglected.

On the other hand, diffraction can play role in lens models. It can be described as follows. When light rays hit a sharp edge of an object some of the rays change their direction going into the geometrical shadow area. It is caused by the propagation of wavefronts which get cropped by the obstacle and from the Huygens principle stating that each point on a wavefront acts as a point source of a new spherical wavefront. The net result is that ray direction can be changed not only by reflection or refraction but also by diffraction.

2.1.3 Light sensing

In order to obtain information about light it is necessary to sense and measure it, that is to intercept it and convert to some measurable quantity. This is the goal of radiometry. A detailed introduction to radiometric concepts and quantities in the context of computer graphics can be found as a part of [70]. In this thesis we will follow their notation. First let us present some important quantities and the measurement equation.

Radiometry

Radiant energy Q is the total energy of photons present inside some region of space (eg. a planar sensor) in a time interval, it is measured in joules $[J]$. *Radiant power* Φ is radiant energy per unit of time: $\Phi = \mathrm{d}Q/\mathrm{d}t$, it is measured in watts $[W = J.s^{-1}]$. The important thing in photography and consequently in image synthesis is that the amount of radiant energy depends both on the radiant power and the exposure time: $Q(T) = \int_0^T \Phi(t)\mathrm{d}t$. When the power is constant it becomes a simple linear function: $Q(T) = \Phi T$.

Irradiance is the radiant power flowing per unit area: $E(\mathbf{x}) = \mathrm{d}\Phi/\mathrm{d}A(\mathbf{x})$, it is measured in watts per square meter $[W.m^{-2}]$. It can be thought as the energy flux flowing though a point. For incoming light the quantity is called irradiance, for light leaving the point it is called *radiant exitance*.

Finally, to measure the amount of light carried by a geometrical ray, a cone subtending an infinitesimally small solid angle ω, *radiance* is used:

$$L(\mathbf{x},\omega) = \frac{\mathrm{d}^2\Phi(\mathbf{x},\omega)}{\mathrm{d}A(\mathbf{x})\,\mathrm{d}\sigma(\omega)\,|\omega \cdot N(\mathbf{x})|} = \frac{\mathrm{d}E(\mathbf{x})}{\mathrm{d}\sigma(\omega)\,|\omega \cdot N(\mathbf{x})|}$$

Note the factor $|\omega \cdot N(\mathbf{x})|$ which appears from the fact the differential area is projected from an area perpendicular to the solid angle. A detailed explanation can be found in [70].

The steady state of the light transport in the scene can be represented by the *incident radiance function* $L_i(\mathbf{x},\omega) : \mathbb{R}^3 \times \mathcal{S}^2$, giving the radiance at point $\mathbf{x}$ incoming from the direction ω, and its counterpart *exitant radiance function* $L_i(\mathbf{x},\omega)$ describing the radiance outgoing from $\mathbf{x}$ to ω.

The fundamental task in image synthesis is to measure the radiant energy incoming to an area sensor (pixel) from the scene. It can be modeled by integrating

incident radiance from all the incoming directions over the sensor area Ω over a period of time $[0; t]$.

$$Q = \int_{\Omega \times \mathcal{S}^2 \times [0;t]} L_i(\mathbf{x}, \omega, t) \, |\omega \cdot N(\mathbf{x}, t)| \, \mathrm{d}A(\mathbf{x}) \, \mathrm{d}\sigma_{\mathbf{x}}(\omega) \, \mathrm{d}t$$

In case we assume the scene is static the energy is linearly proportional to the radiant power which can be computed by discarding time from the integral. The equation is called the *measurement equation*. Compared to [70] it is simplified a bit (we assume the sensitivity of the sensor is independent on position and incident angle).

$$\Phi = \int_{\Omega \times \mathcal{S}^2} L_i(\mathbf{x}, \omega) \, |\omega \cdot N(\mathbf{x})| \, \mathrm{d}A(\mathbf{x}) \, \mathrm{d}\sigma_{\mathbf{x}}(\omega)$$

Sensors

For humans the natural sensing devices are photosensitive cells in an eye's retina – rods and cones. Upon receiving a photon such a cell creates a corresponding nervous stimulus which is then processed further in the eye and in the brain. There is a single type of rods and three types of cones. Rods work mostly at low-light conditions and their output is perceived as being without color information. Cones on the other hand enable color vision. Each of the three types of cones is most sensitive to a different part of the visible spectrum. The simplified behavior of all the mentioned cell types can be understood as they integrate the frequency-filtered incoming irradiance. Together cones give three channels – red, green and blue – this way they project the spectra of incoming light into a three-dimensional color space.

For sensing light outside human eyes the widely used devices are photoelectric cells used in digital photography and photochemicals used in traditional photography. Similar to eye cells both devices are capable of producing a single channel. So in order to get RGB color results they have to equipped with three spectral filters.

2.2 Camera models and related phenomena

As the image synthesis in computer graphics mainly arises from photography in this section we will describe several camera models on different levels of abstraction and realism [6, 7]. Image formation in those models will be studied from the geometrical point of view. Some related optical phenomena has to be described too since they have a great importance in photography as well as in the rendering algorithms.

We will proceed from a general camera model, through pinhole, thin-lens and thick-lens models to a complex geometric lens model and tilt-shift configuration. Meanwhile we will learn about concepts such as depth of field, focus, blur, field of view, circle of confusion, bokeh, point spread function, vignetting, etc. Finally we will see further directions of generalizing the camera models in order to better simulate real optical systems.

2.2.1 General abstract camera model

The fundamental task of human vision, photography and then image synthesis in computer graphics is in creating 2D images as projections of light transport in 3D scenes. Photography and image synthesis mostly try to mimic what a human eye sees but their possibilities of imaging go beyond human vision.

The most primitive optical system for imaging consists of a scene where an arbitrarily complex light transport and interaction happens and of a sensor which detects rays of light coming from the scene.

In abstract terms the scene can be within 3D Euclidean space and the light that can be sensed can be represented by the incident radiance function $L_i(\mathbf{x}, \omega)$ expressing the radiance incoming along a ray from point $\mathbf{x}$ in direction ω. It is also sometimes called the *plenoptic function* [1]. For simplicity we are neglecting here the dependence on time (for dynamically changing scenes) and wavelength (for spectral radiance), a general function would look like $L_i(\mathbf{x}, \omega, \lambda, t)$.

In practice the sensor is usually planar, or more precisely rectangular, but also curved sensors occur, although rarely. In general a sensor's shape can be described by a 2D parametric surface. Further in this text we will assume the sensor is rectangular. Theoretically a sensor could measure incoming light at each infinitesimally small point but in practice it has to be divided into cells of finitely sized area, so that the total incoming energy is not infinitesimally small. In practice many artificial sensors are divided into a rectangular grid (raster) of pixels. In contrast, photosensitive cells in an eye (rods and cones) dot not have equal shape and are distributed stochastically.

Unfortunately, a plain sensor would not give much information about the scene. Each photosensitive cell (or pixel) of the sensor would integrate the contribution of light from all visible incoming directions, so that the sensor would give the total amount of incoming light, acting just similarly to an photometer.

Nevertheless, if we insert an additional device between the scene and the sensor which would filter and/or transform the incoming rays we can eventually get a reasonably sharp and usable image of the scene. Some examples can include just a simple aperture or more or less complex lenses.

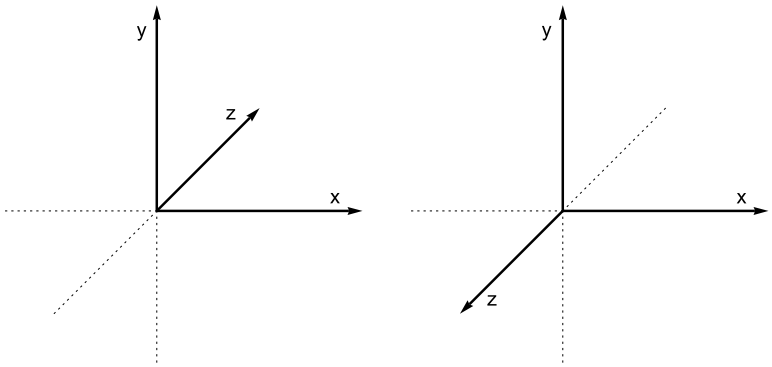
The majority or such devices is entirely rotationally symmetric (or at least most of their parts are) with the axis of symmetry being called the optical axis. This line coincides with the direction the device is pointed in.

Coordinate spaces and transformations

Probably in every 3D rendering context several coordinate spaces are defined and transformations among them are provided in order to simplify calculations. Obeying the exact definitions consistently is crucial to get all the formulas right.

The following coordinate spaces are linear vector spaces and the transformations are performed via matrix multiplication. When useful, homogeneous coordinates are utilized (ie. a space with a dimension higher by one) in order to naturally support transformations such as translation or perspective projection. See the section A.1 for more information.

The basic space is the world space in which the scene objects are defined. We define it as a 3D Euclidean space with right-handed coordinates x, y, z (fig. 2.3). This corresponds to the Object Coordinates in OpenGL. In case of an environment



(a) Left-handed system (b) Right handed system

Figure 2.3: Handedness of 3D coordinate system

with different conventions its specific world space can be transformed to this world space via a suitable transformation matrix. In practice the world space is represented by 4D homogeneous coordinates.

No matter what camera model we use the camera must be positioned in the world space and oriented somehow. Such a rigid-body transformation (consisting only of translations and rotations) takes us from the world space to the camera space. The transformations can be represented by a 4×4 matrix. The origin of the camera space is situated at the optical center of the lens or a pinhole aperture. The camera points in the direction of the $-z$ axis. The x axis points to the right and the y axis point upwards. The half-space with negative z coordinate is called the object half-space, while the rest is the image half-space.

Although the points on sensor can be treated in camera space, it is more natural to address them within a 2D space, since the sensor is assumed to be a 2D parametric surface. This leads to the sensor space, $[0.0; 1.0] \times [0.0; 1.0]$. For the output screen or image a pixel-based raster space is useful – the screen space, $[0; \text{width} - 1] \times [0; \text{height} - 1]$.

Visibility

In case we allow in light transport occlusion and propagation of light along arbitrary paths (such as with refraction or reflection) it is useful to define a *visibility function* as an indicator if there can exist a light path between two given points:

$$V(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if there can exist a light path between } \mathbf{x} \text{ and } \mathbf{y} \\ 0 & \text{otherwise} \end{cases}$$

A lens system then acts as a transformation between the exitant radiance function in the object space L_o and in the image space L'_o. Given a generalized visibility function which takes into account directions at the endpoints of a light path we formalize the relation between to two radiance functions.

$$V(\mathbf{x}, \omega_x, \mathbf{y}, \omega_y) = \begin{cases} 1 & \text{if there can exist a light path} \\ & \text{with endpoint rays } (\mathbf{x}, \omega_x) \text{ and } (\mathbf{y}, \omega_y) \\ 0 & \text{otherwise} \end{cases}$$

$$V(\mathbf{x}, \mathbf{y}) = \operatorname{sgn} \int_{\mathcal{S}^2} \int_{\mathcal{S}^2} V(\mathbf{x}, \omega_x, \mathbf{y}, \omega_y) \, \mathrm{d}\omega_x \, \mathrm{d}\omega_y$$

Since the visibility function allows visibility even on non-direct paths it can carry the information how the lens transforms and blocks all the rays from the scene towards the sensor. The general relation between L_o and L'_o is the following:

$$L'_o(\mathbf{y}, \omega_y) = \int_{\mathbb{R}^3} \int_{\mathcal{S}^2} L_o(\mathbf{x}, \omega_x) V(\mathbf{x}, \omega_x, \mathbf{y}, \omega_y) \, \mathrm{d}\omega_x \, \mathrm{d}\mathbf{x}$$

2.2.2 Pinhole model

A simple solution exists to obtain a more familiarly looking image than with a light meter – use a pinhole camera. Let us close the sensor into a light-tight box equipped with just a single (theoretically infinitesimally) small hole in front of the sensor called a pinhole (or more generally an aperture). For each pixel on the sensor the pinhole aperture would allow only a single ray from the scene, thus creating a sharp image. In practice this has several drawbacks:

- Only a finitely small hole can be manufactured, reducing the sharpness.
- The smaller the hole the less the amount of light that can come to the sensor, increasing the needed exposure time.
- At very small aperture sizes wave optics effects like diffraction would limit the sharpness of the image.

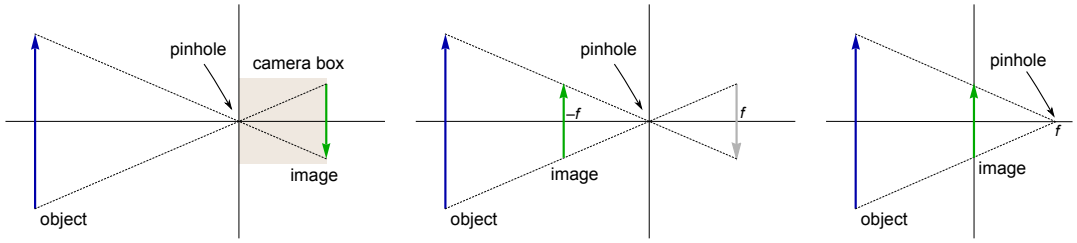
However primitive and imperfect, pinhole cameras were at the birth of photography and even today they are again increasingly popular with art photographers. For some physical applications they are necessary [1] and most importantly they constitute the basic and most widely used camera model in image synthesis.

Perspective projection

In the simplest configuration the pinhole is located at the origin of the camera space and the rectangular sensor lies in the image half-space and is oriented perpendicular to the z axis, so that its center intersects with the z axis (fig. 2.4a). The pinhole is called the *center of perspective* since all light rays pass through it.

The formation of the image of a point in the scene on the sensor can be understood as a ray-plane intersection where the ray originates in the object point and

[1] The advantage for pinholes compared to lenses is that the pinhole aperture is a free space without absorption. In contrast common glasses or lenses from other materials might absorb electro-magnetic radiation at frequencies needed for some applications (x-ray, gamma imaging [61]).



(a) Sensor at f, pinhole at 0 (b) Sensor at $-f$, pinhole at 0 (c) Sensor at 0, pinhole at f

Figure 2.4: Pinhole camera model and its variants

goes in the pinhole direction until it intersects the sensor plane. The position of the image $\mathbf{I} = (I_x, I_y, I_z)$ on the sensor depends on the object position $\mathbf{O} = (O_x, O_y, O_z)$ and image distance of the sensor f in the following way, as can be seen eg. via the similar triangles principle:

$$\mathbf{I} = \frac{f}{O_z}\mathbf{O} = (\frac{f}{O_z}O_x, \frac{f}{O_z}O_y, f)$$

Moreover, it can be treated as a perspective transformation, a special kind of a projective transformation[2]. This way it can be expressed as a linear transformation in a space of one additional dimension, ie. in the 4D homogeneous coordinates – with a simple 4×4 matrix $M_{pinhole}$, such that $\mathbf{I} = M_{pinhole}\mathbf{O}$;

$$M_{pinhole} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{pmatrix}$$

We can see the matrix is singular (with the rank three), ie. it is not invertible. It corresponds to the fact that the perspective transformation projects points with arbitrary depth to a single plane, which irreversibly discards some information.

This formulation of the perspective transformation in the pinhole camera model produces an image on the sensor which is flipped around both x and y axis (or equivalently rotated by 180° around the optical axis) from what is expected from the user's point of view. One possible solution would be to transform the resulting image but in practice the camera model is modified to produce the straight image directly. The sensor can be placed to the object half-space instead of the image half-space, just as in a ray-tracing device used by painters in history (fig. 2.4b). The perspective matrix remains the same but the depth f of the sensor changes its sign.

Such a modified pinhole camera model is widely used both in ray tracing and rasterization algorithms. In ray tracing it provides a way to generate outgoing rays with proper directions, while in rasterization the perspective matrices directly project object-space points to the sensor plane.

Another possible model of the pinhole transformation puts the center of perspective (the pinhole) at $(0, 0, f)$ and the front image plane at depth 0 (fig. 2.4c). The

[2]Note that projection is a transformation from an n-dimensional space to a space of a lower dimension.

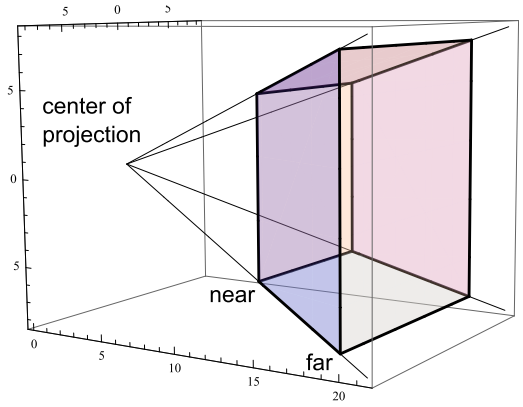


Figure 2.5: Perspective frustum – a pyramid with an apex at the center of projection clipped by the near and far planes.

matrix differs from the previous one by translation of the coordinate space before and after the pinhole transformation. Although this formulation is useful to know since it sometimes it is used in practice, more importantly it will lead us to the thin lens transformation.

$$M_{pinhole'}(f) = M_{translate}(-f)M_{pinhole}(-f)M_{translate}(f) =$$
$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -f \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{f} & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & f \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{f} & 1 \end{pmatrix}$$

Perspective frustum transformation

In rasterization hardware the perspective transformation is modified due to several factors. First, a rectangular sensor along with the center of projection define a viewing pyramid – everything outside is not visible and thus clipped. To prevent division by zero at the center of projection singularity in the conversion from homogeneous coordinates the viewing pyramid is truncated by the near plane. For testing visibility via z-buffer it is necessary for the transformed objects to retain the depth information (in contrast to plain projection onto a 2D surface). The depths are mapped into a finite interval and for minimizing numeric problems is it better to clip the pyramid also with a far plane, so that the ratio of the near/far distances is not too high.

The resulting transformation maps a frustum (a truncated pyramid, fig. 2.5) into a cube. It is invertible and not a projection, note that the depth is mapped to interval [-1; 1], not to a single value. An example projection matrix from the OpenGL context is given. Symbols r, l, t, b represent coordinates of right, left, top and bottom sides of the image rectangle on the near plane, n and f are the (unsigned) distances to the near and far planes (which in fact are located in the $-z$ half-space). The matrix assumes the center of projection is located in the origin (0,0,0), the viewing direction is $-z$. It maps the frustum into a $[-1; 1]^3$ cube.

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

By setting the r, l, t, b coordinates asymmetrically it is possible to obtain off-axis frusta which are useful in stereo-pair imaging and in some DoF rendering methods described further.

The $[-1; 1]^3$ cube is then remapped to the $[0; 1]^3$ cube which is available in the z-buffer.

2.2.3 Camera models with lenses

In order to overcome the practical limitations of real-world pinhole cameras the previously described optical system had to be improved to let more light pass through to the sensor without loosing sharpness. This means allowing more paths of light from a fixed point in the scene to a fixed point on the sensor. The more power can transmitted the less exposure time is needed to obtain a fixed amount of incoming light energy. Optical elements called lenses are widely utilized to solve this problem via refraction of light. They are made of materials transparent to visible light such as various sorts of glass or plastics. Lenses can be modeled at various levels of abstraction and realism. We will start by describing the simplest models, and then move on to more complex ones. The principle of a lens is not unique to optics and can be seen in other fields of physics and engineering.

So far the pinhole model only absorbed some incoming rays while letting the others continue in the original direction. The following models transform the rays, so they act as a mapping between rays incoming from the scene and rays outgoing to the sensor: $T(R_{in}) = R_{out}$. In addition some incoming rays can also be absorbed.

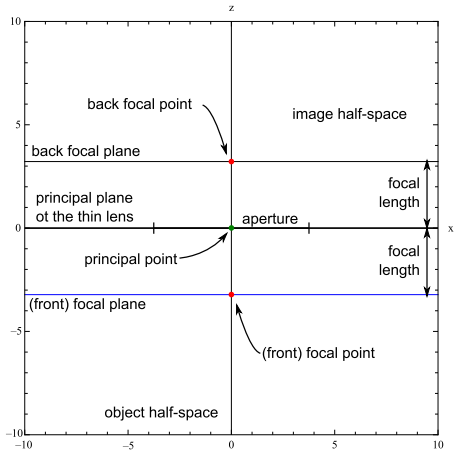
Thin lens model

We start with a theoretical model of a lens which is infinitesimally thin in the z axis direction. It has been widely used in optics and was introduced to computer graphics in the first article on depth of field rendering [59]. The lens is represented by a principal plane, $z = 0$, and it transforms rays of light incident to one side to rays outgoing from the other side. It can be limited by an aperture, an element opaque except for a planar region most often of a circular shape (see fig. 2.6a).

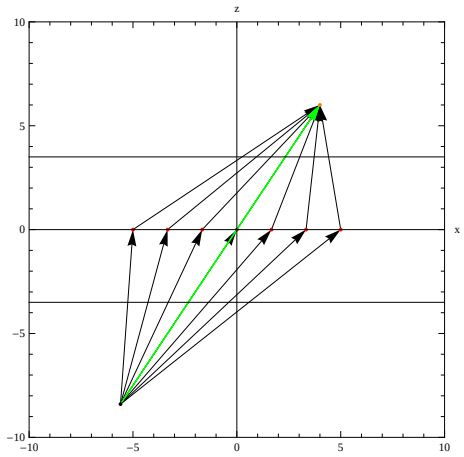
The thin lens model has a very important property – it transforms a bundle of collinear rays parallel to the optical axis to a double cone of rays converging at the focal point (fig. 2.6c). More generally, any bundle of collinear rays is focused into a skewed double cone with an apex at a point on the *focal plane*, a plane parallel to the principal plane (fig.2.6d), intersecting the focal point. It should not be confused with the *focus plane*, a plane where the camera is focused depending on the position of the sensor.

The distance between the focal plane and the principal plane is called the focal length [3] of the lens. The focal point corresponds to the intersection of the focal plane

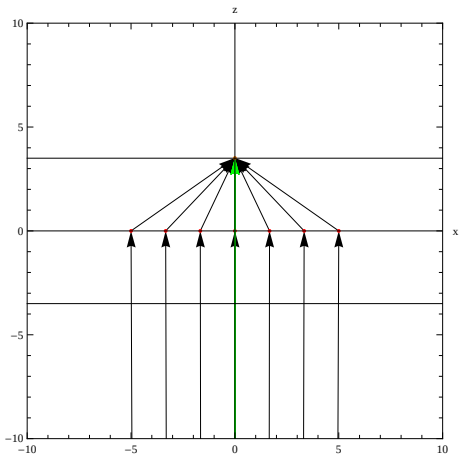
[3]When distinguishing converging and diverging lenses the signed distance is used – positive for



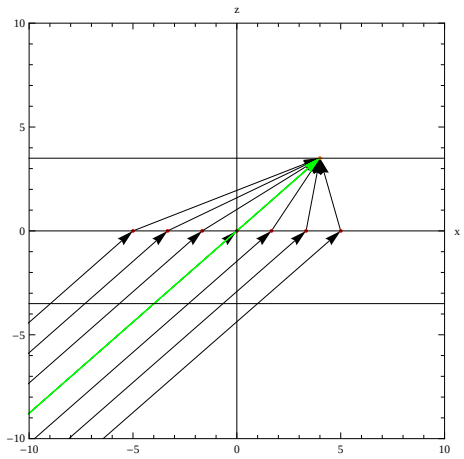
(a) Thin lens model.



(b) Finite object-image conjugate pair.

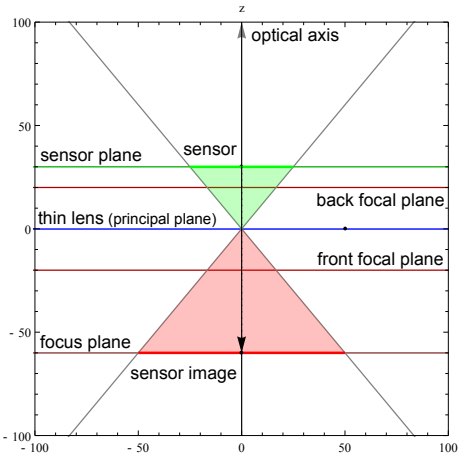


(c) Collinear rays (object at infinity, image at the back focus point).

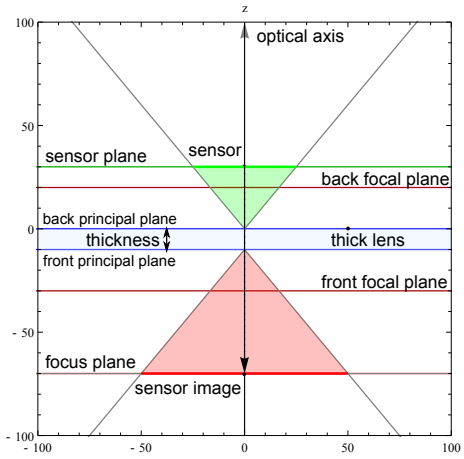


(d) Off-axis collinear rays (object at infinity, image at the front focal plane).

Figure 2.6: Illustration of the thin lens model and imaging of points.



(a) Thin lens



(b) Thick lens

Figure 2.7: Viewing pyramids through the principal point for the thin and thick lens.

with the optical axis. The focal length is the inherent parameter of the thin lens, thus the focal plane does not depend on the sensor (ie. its position or orientation). In fact there are two focal points and focal planes, depending on the half-space which the collinear rays are coming from. To distinguish them the focal point (or the focal plane respectively) in the image half-space is called the back one, and the other the front one. By default the front focal point and plane are denoted.

Another important property of thin lenses is that they transform every plane in the object half-space to a unique plane in the image-space. Or in other words for a thin-lens transform T_{thin} and a fixed plane P_o in the object half-space, there exists a unique plane P_i in the image space, such that for each point $O \in P_o$ holds $I = T_{thin}(O) \in P_i$. Here, the "plane in a half-space" means the intersection of a plane with the half-space.

The basic transformation only transforms one point to another. As we will see it is invertible, so that applying the transform again yields the original point. The pair of points which get transformed one to another (fig. 2.6b) is called a conjugate pair and those two points are said to be conjugate to each other [4]. A set of points (eg. a line, ray or plane) can be transformed point by point. The transformation can operate in homogeneous coordinates, which enables us to work with points at infinity.

The lens itself, being so thin, can be represented by a plane, the principal plane, corresponding to the xy plane of the camera space. If we place the planar sensor into the image half-space and transform it with the thin lens transformation its image lies the focus plane in the object half-space. A cone of rays diverging from any point light source on the focus plane get transformed by the thin lens to another cone of rays converging to a single point on the sensor, thus rendering the point light object as a point image – in focus.

This way the lens model trades off more than one path of light from a point in the scene to a corresponding point on the sensor for being able to display in focus only a part of the scene, a single plane.

Fortunately, it is possible to express the thin-lens transformation via a 4×4 matrix. To transform a point P (in homogeneous coordinates) from the object half-space to the image half-space it suffices to multiply it with matrix $M_{thin}(f)$, where f is the focal length of the thin lens:

$$P^* = M_{thin}(f)P, \qquad \text{where} \quad M_{thin}(f) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & -\frac{1}{f} & 1 \end{pmatrix}$$

Note that it differs from the modified perspective matrix $M_{pinhole'}(f)$ by an added 1 to the element in the 3rd column of the 3rd row. The matrix $M_{thin}(f)$ is regular, thus an inverse exists. Transforming a point P from the image half-space to the object half-space is done via the inverse matrix: $M^{-1}_{thin}(f)$:

converging ones and negative for diverging ones. We will only work with converging lenses and thus use the unsigned focal length only.

[4] We will denote conjugation by an asterisk, eg. A^* is a conjugate point to A

$$P = M_{thin}^{-1}(f)P^*, \qquad \text{where} \quad M_{thin}^{-1}(f) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{\mathbf{1}}{\mathbf{f}} & 1 \end{pmatrix}$$

The invertible nature of the thin-lens transformation is a big difference to the perspective (pinhole) transformation. Since $M_{thin}^{-1}(f)M_{thin}(f) = I$, transforming a point with a thin-lens forth and back results in the original point: $M_{thin}^{-1}(f)M_{thin}(f)P = P$.

It has to be noted that thanks to using homogeneous coordinates points at infinity can be represented. They are in form $(x, y, z, 0)$ and correspond to direction vectors. This is the reason why the thin-lens matrix is able transform a bundle of collinear rays (converging at infinity) into a cone converging at a single point at the focal plane – the conjugate point to the point at the focal plane is a direction from infinity.

In the thin lens model rays going through the center of the principal plane continue in the same direction (as in the pinhole model). In contrast, for other rays the direction is modified by the lens.

It turns out that apart from being a nice theoretical model of an ideal lens the thin-lens model acts as a quite good approximation of some real physical lenses under some given conditions – in particular, the Snell's law describing the refraction of rays of light:

$$\eta_1 \sin(\theta_1) = \eta_2 \sin(\theta_2)$$

The equation contains a trigonometric function, the sine, which can be expanded into Taylor series (an infinite sum of polynomials). The partial sums, Taylor polynomials, converge to the exact value and thus can act as an approximation. Taking only the first term of the series is called the first-order approximation: $\sin(\alpha) \sim \alpha$. This is called the *paraxial approximation*, since the error is small only for rays making a small angle with the optical axis.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2x+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \quad \forall x \in \mathbb{R}$$

In order to filter the incoming rays the thin-lens can be equipped with an stop of an arbitrary planar aperture shape situated at the principal plane. There are several reasons for this: real lenses have a finite aperture by design constraints, the aperture size and shape affect the quality of depth of field, the paraxial approximation is bad for great angles, algorithms for image synthesis require sampling a finite area. Usually a stop with a circular aperture centered at the optical axis is used, although different shapes such as regular polygons are quite common to mimic physical iris-blade diaphragms. In general, the aperture could be placed at a different depth.

Depth of field

Before we continue to other more complex camera models we should explain some important related concepts. When describing the thin lens model we have already mentioned the concept of focus. A point light object is imaged on the sensor *in focus* when the bundle of rays emerging from it converge to a single point on the sensor. Otherwise, when the rays from the point object arrive at different points on the sensor, the object is said to be *blurred*.

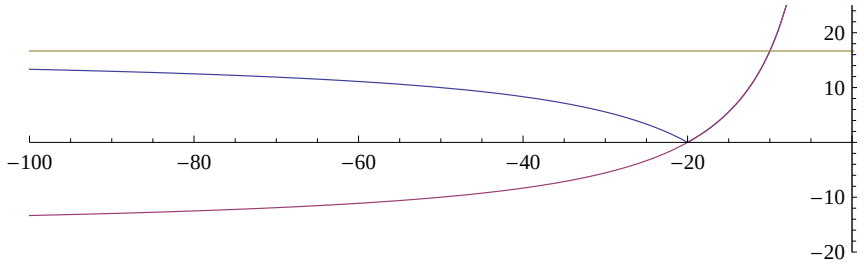


Figure 2.8: The plots of the signed (purple) and unsigned (blue) CoC radius depending on the light source depth for focus plane at $z_{fp} = -20$, focal length $f = 50$ and aperture radius $r_a = 10$ (the dimensions are exaggerated for illustration purposes). The (finite) limit value of unsigned CoC radius for light sources at infinity is in yellow. Note the hyperbolic shape on the signed CoC function and that the unsigned function grows monotonically with increasing distance of the light source from the focus plane. The CoC size limit is infinite at zero depth, however when the visible scene is clipped at a non-zero near plane the CoC size there is finite.

In the ideal thin lens model with a circular aperture the bundle of rays from a single point P incoming at the aperture is transformed into a double cone of outgoing rays with the apex at the conjugate point P^*. The sensor plane then intersects it yielding a conic section. In the simplest case when the sensor is oriented perpendicular to the optical axis the intersection is only circular, otherwise it is a conic section.

In summary, a single point is imaged as a point on the sensor when being in focus or as a circle when out of focus (under all the assumptions). It is called the *circle of confusion* (CoC). The radius of a CoC depends on the object distance from the lens plane, sensor distance from the lens plane, aperture radius and the focal length of the lens (fig. 2.8). It can be shown that the CoC radius can be computed by the following formula:

$$C(z_o, z_{fp}, r_a, f) = \frac{|z_o - z_{fp}| f r_a}{z_o(z_{fp} - f)},$$

where z_o is the object depth, z_{fp} is the focus plane depth, r_a is the radius of the aperture and f is the focal length. In case the aperture is not symmetric the CoC of a point nearer than the focus plane is oriented as the aperture, while the CoC of a point farther than the focus plane is flipped both horizontally and vertically. Sometimes thus is useful to compute the signed CoC radius:

$$B(z_o, z_{fp}, r_a, f) = \frac{(z_o - z_{fp}) f r_a}{z_o(z_{fp} - f)}.$$

The depth z_i where the CoC approaches zero can be computed via the transformation matrix or equivalently via the thin-lens equation:

$$\frac{1}{z_i} + \frac{1}{z_o} = \frac{1}{f}.$$

If the CoC is sufficiently small it is perceived as a point rather than an area by any sensor with limited spatial resolution (including a human eye). The size of the maximum CoC perceived as a point depends on many factors, such as the visual

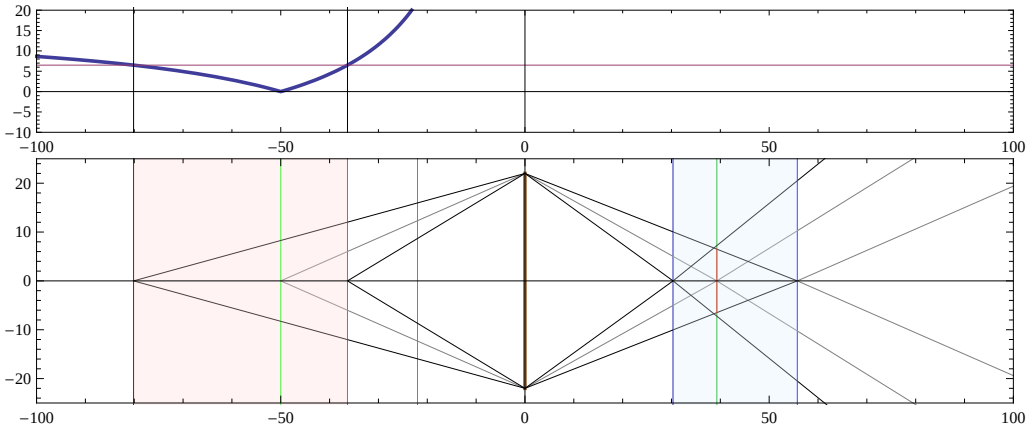


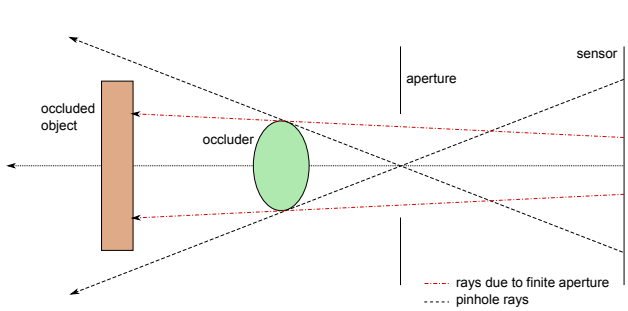
Figure 2.9: Depth of field formation and limits. Points in the object space (left) are imaged through a thin lens (center) to the image space. Without the loss of generality points on the optical axis are treated. The unsigned CoC radius function C plot is shown for completeness (thick blue). A point at the focus plane z_{fp} is imaged as a point on the sensor plane (green). For a fixed CoC size c_{limit} there are two object-space points with this value lying on the near and far DoF limit planes (red lines). In the region between (light red), the depth of field, the CoC radius is smaller than c_{limit}. The image of the object-space depth of field is the image-space depth of field (light blue). Parameters: $f = 22, r_a = 22, z_{fp} = -50, c_{limit} = 6.5$.

acuity of the viewer, viewing conditions, angular image size, etc. Objects imaged within this CoC limit then appear sharp. If we fix the parameters z_{fp}, r_a, f and map each point $O = (x, y, z)$ in the object half-space to the size of a CoC produced by it $(x, y, z) \rightarrow C(z; z_{fp}, r_a, f)$ we get a 3D scalar field. The CoC limit gives an isosurface in this field – the region inside (with lower CoC radii) is the *depth of field* (DoF) which is perceived sharp and the rest is perceived blurred (fig. 2.9). So that although only the focus plane is imaged completely sharp, a certain region around it is also perceived sharp at given conditions. In the simple camera configuration the region of depth of field is bounded by two planes perpendicular to the optical axis, which can be represented by a depth interval.

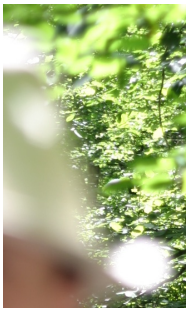
It is important to realize that the if a fixed total power of a beam of light gets spread over wider area (greater CoC size) the irradiance gets smaller. In practice it means that a larger CoC gets dimmer. This leads to the nature of bokeh. In photographic terminology *bokeh* [53] denotes an image of a small light source which is much brighter than its neighborhood, a distinguishable CoC.

For real physical lenses and when working beyond the simple geometrical optics the image of a point light source might be quite far from a perfect constant-intensity circle. Its appearance is, however, to a great extent affected by the shape of the aperture. The image of a point light source, the impulse response of the optical system, is called the *point spread function* (PSF) in optics. The CoC is thus a special kind of a PSF for simple idealized optical systems.

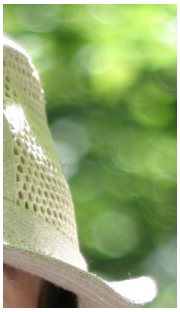
Apart from the physically-based depth of field there exits a non-physical *generalized depth of field* and methods for its rendering [44, 9, 47]. Its usage is for semantic and artistic purposes.



(a) A diagram of image formation leading to partial occlusion, redrawn from [7].



(b) A photograph with partial occlusion.



(c) The same, focused at the foreground.

Figure 2.10: Illustration of partial occlusion.

Partial occlusion

In the infinitesimal pinhole model a point on an object is either visible or occluded since it can be seen only through a single point, the pinhole. Models with a finite aperture, including finite-aperture pinhole, thin lens and others allow a point on an object to be visible from more than one viewpoint on the sensor. This leads to an effect called *partial occlusion* where a background object (totally occluded in the pinhole model) can be seen behind an edge of an out-of-focus foreground object, which in turn appears to be partially transparent (see fig. 2.10).

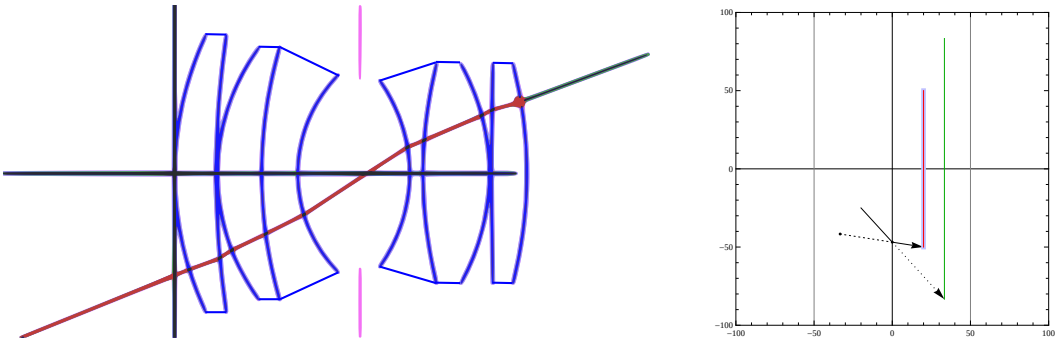
Thick lens model

Many physical lenses are composed of elements of non-trivial thickness and curvature. The thin lens model can be modified a bit to better approximate such lenses while still keeping a representation via a matrix. We might assume there are two principal planes instead of just one. An incoming ray then gets shifted from one plane to the other before being transmitted in a direction given by the thin lens matrix. The result is the same for transforming an object point to an image point, the image point is just translated compared to the thin lens transformation. See fig. 2.7b for illustration.

In the matrix terminology we can compose the thick lens transformation of a point of two operations – a thin lens transformation followed by a translation along the z axis direction:

$$M_{translate}(t) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & t \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad M_{thin}(f) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{f} & 1 \end{pmatrix},$$

$$M_{thick}(f,t) = M_{thin}(f)M_{translate}(t) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1-\frac{t}{f} & t \\ 0 & 0 & -\frac{1}{f} & 1 \end{pmatrix}$$



(a) Ray tracing of a Double Gauss lens (b) Pupils

Figure 2.11: Complex lens and entrance and exit pupils. Entrance and exit pupil in a thin lens model for an aperture stop which is not aligned with the principal plane. Rays pointing at either pupil get transformed so they fit within the aperture stop.

The inverse transformation is as follows:

$$M_{thick}^{-1}(f,t) = (M_{thin}(f)M_{translate}(t))^{-1} = M_{translate}^{-1}(t)M_{thin}^{-1}(f) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -t \\ 0 & 0 & \frac{1}{f} & 1-\frac{t}{f} \end{pmatrix}$$

Complex geometric models

The previous models have many assumptions to keep them simple. The drawback is they approximate real-world lenses very roughly and are very limited in providing realism. A more sophisticated model is needed to deliver features such as:

- correct geometrical image formation
  - non-ideal imaging – a single point might not be imaged as a single point (multiple rays which would intersect at a single point in the ideal model might not intersect at a single point in the complex models due to optical aberrations)
  - optical aberrations
  - geometrical distortions
  - vignetted bokeh (CoC shape clipped by multiple apertures)
- correct radiometrical image formation
  - light fall-off outwards from the image center caused by vignetting

A model presented in [41] is capable of this, while still working within the limits of geometrical optics. It simulates the passage of light through a lens system represented as a sequence of refractive lens element surfaces and stops aligned on a common optical axis. All components are assumed to be rotationally symmetric, in particular the lens elements are assumed to be spherical caps and the stops being circular, but a generalization is possible. In practice a refractive lens element can be made of a single piece of glass or plastics, in the model, however, we treat both surfaces of the element separately.

To make the terminology clear let us explain the difference between aperture, stop and diaphragm. An *aperture* is the widest planar transparent opening in a lens

r_c	t	i_r	d_a
58.950	7.520	1.670	50.4
169.660	0.240	1.00027715	50.4
38.550	8.050	1.670	46.0
81.540	6.550	1.699	46.0
25.500	11.410	1.00027715	36.0
N/A	9.0	1.00027715	34.2
-28.990	2.360	1.603	34.0
81.540	12.130	1.658	40.0
-40.770	0.380	1.00027715	40.0
874.130	6.440	1.717	40.0
-79.460	72.228	1.00027715	40.0

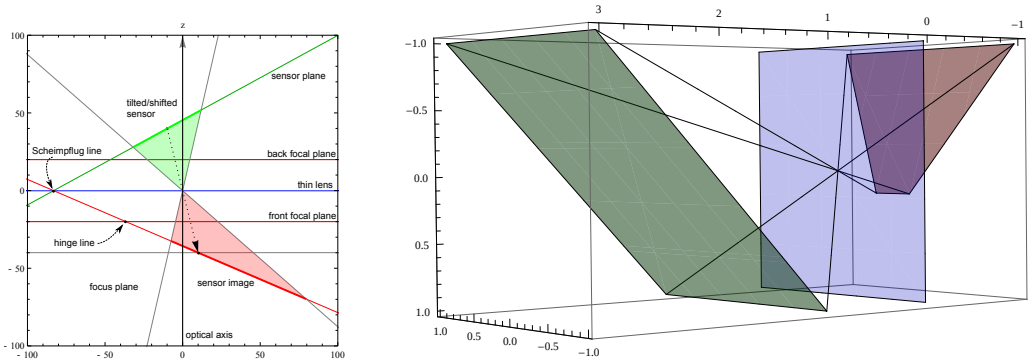
Table 2.1: Tabular front-to-back description of a Double Gauss lens. r_c is the radius of curvature, t denotes element thickness, i_r is the refractive index, and d_a is the aperture diameter.

element surface or stop. A *stop* is a planar opaque element with a transparent hole (aperture) inside. And finally a *diaphragm* is a physical implementation of a stop with variable aperture size. Note that each lens element or stop has an aperture, eg. for a lens element surface of a spherical cap shape the aperture is equivalent to the base circle.

Lens systems can be defined via tabular descriptions compatible to what can be found in optical design literature. See an example of a Double Gauss lens – tabular specification of elements surfaces [41] in table 2.1 and ray tracing in fig. 2.11a. The elements (spherical caps or circles) are sorted by their intersection with the optical axis (apex or center) from front to back (towards the z direction). Each spherical cap has the signed radius of curvature r_c specified (positive value means a convex surface from the front and negative conversely). Each element has also specified the diameter of aperture d_a (for a spherical cap the aperture is its base circle), thickness t (z offset to the apex of the next element) and material of the medium beyond the surface, eg. specified by its refractive index i_r.

A property of material relevant in this context is that the refractive index is wavelength-dependent. In practice it can be approximated by a constant value correct only for a single frequency, a linear function or by a more complex non-linear function.

To proceed further we have to define several additional terms. *Aperture stop* is the stop whose image from an arbitrary point on the optical axis (on either side of the lens) subtends the smallest angle with the optical axis. Its image when looking from the object half-space is called the *entrance pupil*, while when looking from back it is called *exit pupil* (see fig. 2.11b). When imaging the aperture stop it is crucial to use only the elements between the aperture stop and the viewing point. Moreover, the entrance and exit pupils are images of each other when using the whole optical system for imaging [66]. Those two pupils play an important role in most camera models. For off-axis viewpoints there can be an *effective pupil* [68], an image of multiple apertures limiting the cone of light. Its shape is closely related to vignetting.



(a) 2D (b) 3D

Figure 2.12: The situation for a tilt-shift sensor with a thin lens. A tilted sensor (green) and its image (red) via a thin lens (blue).

In many real lenses the aperture stop is a diaphragm with variable aperture size. Its purpose is to limit the amount of light passing though the lens system in order to control the total power of the incoming light (thus brightness of the image), the depth of field and also the effect of aberrations. The power that can pass depends linearly on the area of the entrance pupil (thus also the aperture stop), so that it depends on the diameter of the entrance pupil by a square root: $\Phi \sim A \sim d^2$. In order to let half the power though the lens system the aperture diameter must be decreased by a factor of $\sqrt{2}$. The entrance pupil diameter d relative to the focal length f of the lens is called the *f-number* (here denoted by n): $d = f/n$, eg. $f/8$, which is a wide-spread term a notation in the photographic community.

Due to technological constraints real variable-size iris diaphragms only approximate the circular shape, usually with regular polygons or similar shapes. The shape of the aperture stop has several implications on the image formation both in terms of geometrical and wave optics and also as an artistic tool.

Some complex lenses can be approximated by thick lenses. Details are provided in [41]. Also a detailed explanation of various sources of vignetting is in [68].

2.2.4 Tilt-shift configurations

So far we have assumed that the sensor is oriented perpendicularly to the optical axis with its center being aligned with the axis. In general, configurations with a tilted and/or shifted sensor are clearly possible. Physical examples include tilt-shift lenses, where the lens system is moved against the rest of the camera, and view cameras consisting of two separately movable boards for lens and for sensor. In this section we will discuss the implications of allowing tilt-shift configurations in the previously described camera models.

First we might ask a question: Why to bother with tilt-shift configurations? The main reason is that they provide some very interesting and also useful visual effects. Shifting (translating) the sensor in the z direction is commonly used for focusing, but shifting in the xy plane (perpendicular to the optical axis) can produce dramatic changes in perspective – without changing the camera or lens position. This is utilized mainly in architectural and technical photography.

Tilting (rotating) the sensor is, however, far more interesting as it is capable of

changing the focus plane and the depth of field – again keeping the original camera position. With a tilt it is possible to focus on an almost arbitrary plane in the scene. A shift is then used in conjunction with tilt to compensate the visible region of the scene. In today's photography a popular usage of tilt-shift is in fake miniatures – photographs of cityscapes from bird's view with a very shallow depth of field thus conveying the visual appearance of being small-scale – and also for drawing attention to the focused subject.

Note that the optical axis is determined by the lens system and moving just the lens system (thus also changing the world-camera transform) is equivalent to moving the rest of the camera inversely and leaving the world-camera transform intact. Therefore we can in the models consider only the second case – tilted and/or shifted sensor – even though in practice lenses are tilted more often.

The principles of tilt-shift camera configurations and methods of focusing are presented eg. in [63, 51] in terms of trigonometry which might not be quite intuitive. In the following text we will describe the tilt-shift configurations of a thin and thick lens models from the point of view of transformations in homogeneous coordinates.

The basic insight is that points and sets of points (eg. lines, planes or arbitrary objects) on each side of the lens are in related to their conjugates by the lens transformation[5]. In particular the sensor plane is the conjugate of the focal plane. Nothing prevents from tilting the sensor plane in which case the focal plane still keeps being the conjugate.

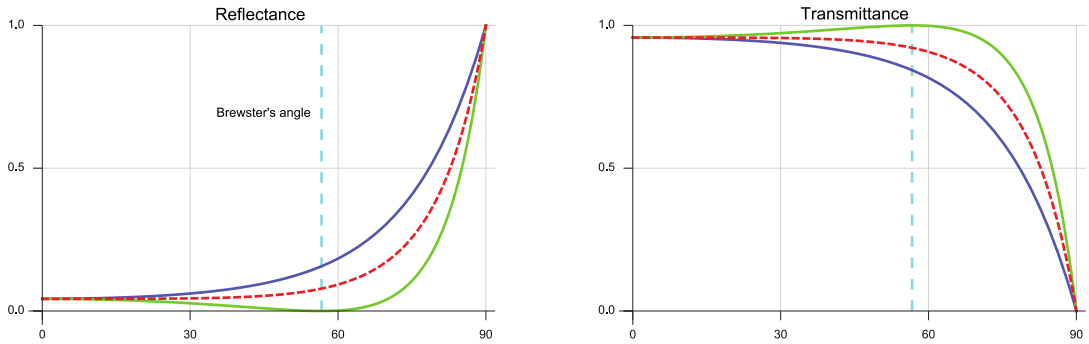
By tilting the sensor plane it becomes no longer parallel to the back principal plane, so that both planes intersect at a line. The same holds for the focal plane and the front principal plane. In case of the thin lens model there is only one principal plane and both lines of intersection are the same, since any point on the principal plane is imaged to itself. This line is called the *Scheimpflug line* in literature [63]. For thick lenses with non-zero thickness there are two such lines, each on one principal plane, and the sensor plane intersects with the focal plane on a separate line in the region between the principal planes.

There is also another even more interesting line, called the *hinge line* [51]. Its property is that it is invariant to any translation of the sensor plane (or in other words it stays invariant if the sensor plane orientation is invariant). It lies on the front principal plane from which follows that images of its points are directions (or points at infinity). They correspond to all directions on the sensor plane (ie. perpendicular to its normal). This line is the axis of rotation of the focal plane when the sensor is shifted in any direction (not only in the xy plane but also in the z direction). The behavior of the hinge line resembles the behavior of vanishing points in perspective [62].

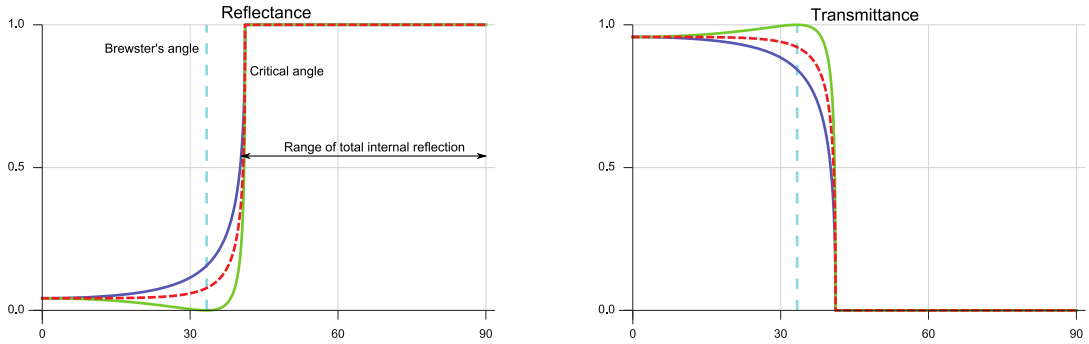
Finally when the sensor is rotated by some axis of rotation A the focal plane is rotated around the image A^* of A. Note that in contrast to [47] it is perfectly possible for a sensor with non-zero shift that the focal plane is aligned with the z axis. On the other hand only a small fraction of light comes to the sensor as it needs to be oriented almost completely from the lens aperture.

The problem of focusing on a particular focal plane in the object half-space can be solved simply by transforming the focal plane via the lens transformation to the

[5]We assume linear transformations in the homogeneous space, but a generalization is possible even to non-linear transformations in complex geometric lenses provided the lens is able to focus an image of a single point to another single point.



(a) Interface towards the denser material (eg. air to glass).



(b) Interface towards the sparser material exhibiting total internal reflection.

Figure 2.13: Fresnel equations describe reflectance and (its complement) transmittance as functions of the angle of incidence. Functions for S-polarization (blue), P-polarization (green) and their average, an approximation of unpolarized light, (red) are shown. The interface is between two dielectric materials with real-valued refractive indices. The illustration is based on plots courtesy of Dr. Alexander Wilkie.

needed sensor plane. For non-tilted configurations the focal and sensor planes are defined by a single parameter (z depth), for tilted configurations the planes are defined by three points. So for an "auto-focus" feature in both tilted and untilted configurations it is only needed to pick at most three points in the object half-space and transform them to the image space. The advantage of image synthesis is that we have the depth information and need not to rely only on the image information.

Note that tilt-shift configurations change the perspective (a tilt-shift sensor together with a pinhole produces a different perspective frustum than an untilted sensor). Only a non-physical generalized depth of field could produce variable blur field without any changes in the perspective. Also note that depth of field in the context of tilt-shift configurations have to be defined differently as the CoCs resulting from a circular aperture might be arbitrary conic sections.

2.2.5 Further models

The complex geometric camera model [41] does not try to simulate all the important phenomena of physical cameras and does not go into the area of wave optics. Several newer and even more complex models have been recently published [39, 69, 72]. They address also the following phenomena.

The previous model [41] assumed that rays are perfectly transmitted during re-

fraction and no light is absorbed or reflected. In reality the materials used for optical elements both reflect and refract light and the ratio of reflectance and transmittance is described by the Fresnel equations (see fig. 2.13). Also media such as glass have non-zero absorbance so that the light intensity is attenuated during transmission.

All rays reflected inside the lens system, with total internal reflection on lens element surfaces or bounced off the lens barrel, have been discarded. Such rays from very bright light sources within the field of view or even outside can produce *lens flare* or ghosts – multiple variously distorted images of the aperture stop. Lens coatings are used in practice to minimize all the reflections affecting the appearance of the lens flare. For physical and technological reasons they cannot filter all the reflected light so variously colored lens flares are possible in practice.

Also for a more accurate simulation of chromatic aberrations better models of dispersive optical materials and spectral rendering are necessary.

Some models take diffraction into account [39, 69, 59]. Fraunhofer approximation for far-field areas is able to produce a *star-burst* or *glare* pattern observable in photography at very small aperture sizes, while Fresnel approximation is good at near-field "ringing" patterns. This can be efficiently computed via the fractional Fourier transform [54].

Completely wave-based optical simulation frameworks are also possible [73] where effect like diffraction are just natural consequence of the rendering model.

2.2.6 Fourier optics and PSF

In Fourier optics [30] an imaging system is described as a linear system where the image of the scene is given by convolution of the light sources in the scene with a *point-spread function* (PSF), an impulse response of the system.

The imaging system is usually described as a focused system with an aperture for a single point light source or a planar object field at the focal plane and a planar image (sensor) field of irradiance. The aperture is defined by a pupil function which represents visibility. The point spread function then depends only on the aperture shape and size. More precisely it is interrelated to the pupil function by a Fourier transformation.

A complex object composed as a sum of many light sources is a due to linearity of the system a sum of images of single points. Thanks to the convolution theorem convolution in the spatial domain can be reduced to multiplication in the Fourier frequency domain. Unfortunately, this is only applicable when the PSF is constant, which is not true in practice.

So far we have only treated visibility through the lens aperture and not in the scene. When some objects in the scene get occluded by other ones the PSF can get clipped and thus becomes scene-dependent, too. Also the visibility through the lens might be dependent on the position on the sensor leading to effective pupil function. This way we have a convolution of the scene (or more precisely the plenoptic function of the scene) with a very complex spatially-varying kernel, the visibility function.

Chapter 3

Methods of depth of field rendering

The goal of all the rendering methods is to (approximately) compute the radiant power incoming from the scene to each pixel (the measurement equation). In this thesis we are only interested in transforming the incident radiance function of the scene into an image on the sensor, so we will not go in details of simulating light transport in the scene.

We will briefly summarize the various important approaches to depth of field rendering and describe in more detail the relevant methods which served as the basis for our implementation – in particular some reference methods and some state-of-the-art interactive methods. For more information the reader should consult the existing surveys of DoF rendering methods and camera models used in computer graphics [6, 7, 24, 8, 43]. As the field of DoF rendering rapidly evolves the latest methods are not covered in those surveys.

There are several criteria to distinguish the nature of various rendering algorithms. An in-depth summary can be found in Levoy's introductory essay to point-based rendering [31] which helps to understand the differences and connections between seemingly diverse algorithms.

One criterion is in the scene representation. Distributed ray tracing and rasterization belong to the group of rendering algorithms which operate on geometrically represented scenes. On the other hand post-processing methods (together with point-based rendering methods) operate on sample-based representation of the scene [31], eg. layered depth images [65]. Such representations can either originate in image synthesis or by capturing real world data (such as light fields or depth images from depth cameras or range scanning).

Another criterion distinguishing rendering algorithms in how they solve visibility, ie. deciding which scene primitives contribute to each pixel and vice versa. Since the visibility problem can be reduced to sorting there are two main approaches: image-based algorithms such as ray tracing gather contributions from geometry primitives and find the nearest intersections with geometry per-pixel. In contrast, z-buffer rasterization (an example of object-based algorithms) sorts the geometry primitives and finds contributions of each visible primitive to image pixels. The algorithms differ in the order of the nested loops.

3.1 Reference methods

The basic off-line reference method is Monte Carlo distribution ray tracing. The basic interactive reference method is image-based multi-view accumulation.

3.1.1 Distribution ray tracing

The basic ray tracing method [71] just traces a single ray from each pixel through a point-sized pinhole into the scene and computes the direct lighting at the point of intersection of the ray with an object. To handle reflections or refractions at reflective or transparent surfaces the rays can be traced recursively. However, the method computes only direct lighting. It cannot handle a full lens system.

In order to support pixel anti-aliasing, depth of field due to finite sized lens aperture, non-perfect reflections or refractions or soft shadows it is needed to compute multidimensional integrals. A good way to accomplish this approximately is via Monte Carlo numerical integration. As the rays are traced according to some non-singular distributions of directions the method is called distribution ray tracing [19].

Ray tracing handles visibility implicitly by sorting the intersections with scene geometry in each ray and taking the nearest intersection. Thus it is free of artifacts present in some other methods, such as a lack of partial occlusion (intensity leakage, depth discontinuity artifacts), discretization artifacts, incorrect blurring or compositing, etc. [7]. The drawback of this method is that for each lens sample a separate ray must be traced and the complexity of the intersections scales with the amount of scene geometry [33].

Monte Carlo integration

Given a definite integral of function f we can approximately compute its value by evaluating the function at N random or pseudo-random samples within its range according to some distribution p. The estimate is then given by averaging the sampled with weights according to p [70]:

$$I = \int_\Omega f(x)\,\mathrm{d}x \approx \hat{I}_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}.$$

With the increased number of sampled the estimate $\hat{I}_N$ converges to I. The approximation error is in form of a high-frequency noise. A commonly used technique to reduce the noise with the same number of samples is stratified sampling. The sample domain is divided into disjoint regions completely covering it and random samples are drawn from each region independently [70].

Combination with lens models

Since distribution ray tracing is able to integrate rays going through a lens with a finite aperture it can be equipped with any of the previously described lens models. We assume that the incident radiance function of the scene can be computed by the underlying ray tracing algorithm and the only thing we would like to focus is its transformation in the lens model and projection on the sensor.

Whichever lens model is used we can always sample the entrance or exit pupils. They might be circular or bounded by a circumscribed circle. One of the standard approaches [41] is jittered sampling [28] with concentric square to disk mapping [67]. For a non-circular aperture this could be combined with rejection sampling.

Sequential ray tracing of the geometric lens system

Since a lens system is the only way the light can pass from the scene to the sensor and the exit pupil usually subtends a small solid angle when viewed from a sensor pixel it does not make sense to treat the lens system a part of the scene and sample all incoming directions from a pixel. Indeed, [41] proposed a rendering method, described here, in which rays are traced in the lens system separately from the scene. If one takes into account that the light rays pass the lens elements in a fixed sequence it greatly simplifies finding the next primitive to intersect, leading to a O(1) cost. This is not possible if the lens system is treated being a part of the scene and the elements being stored in a huge acceleration structure.

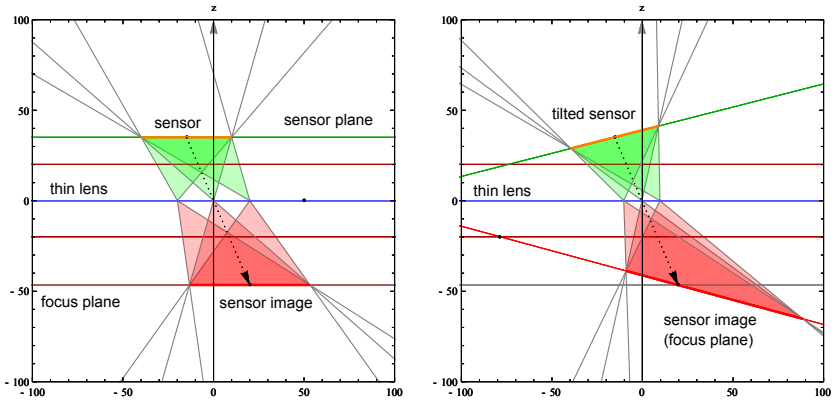
There are two primitives for representing lens elements surfaces – spherical caps and circles. Both acting as a basic primitive (sphere, plane) clipped by an additional condition. Other surface primitives are possible, eg. parabolic caps for aspheric elements or image-based apertures. The formulas for ray-circle and ray-cap intersection are described in Appendix 1.

Algorithm 1 Pseudocode of sequential ray tracing of a lens system:

Input: incoming ray R_i
Output: outgoing ray R_o
  for all lens surfaces S **do**
    intersection $I \leftarrow$ intersect R_i with S
    if (no intersection) or (I outside aperture of S) **then**
      return ray blocked
    end if
    $N \leftarrow$ normal of surface S at I
    $D_r \leftarrow$ refract ray R_i at I with normal N
    if ray refracted **then**
      $R_o \leftarrow (I, D_r)$
    else
      return ray blocked by TIR
    end if
    $R_i \leftarrow R_o$
  end for
  return R_o

The algorithm of lens system ray tracing (see algorithm 1) can be used either for ray generation in backward ray tracing (from sensor to scene) or for forward ray tracing, what changes is the order of lens elements. If one assumes a backward order, given a ray incoming at the back-most surface the algorithm finds a corresponding ray which leaves the front-most surface or reports that the ray was blocked inside. Note that no internal reflections are taken into account.

An incoming ray is created by taking a sensor pixel sample and a lens sample. For ideal lenses the lens samples should be placed on the exit pupil (for backward ray



(a) Thin lens, shift only (b) Thin lens with tilt-shift

Figure 3.1: 2D projection of multiple perspective pyramids for different points on the lens aperture – thin lens model without and with tilted sensor. Note that all the object-space pyramids have to intersect at the sensor image on the focus plane to bring this in focus.

tracing), ie. the image of the aperture stop from the back side since only the aperture stop limits visibility through the lens. However complex lenses introduce vignetting, so that multiple apertures limit visibility. A safe solution, though possibly not very efficient, is to place the lens samples on the surface of the back element or better at its aperture. It is guaranteed that no rays will be missed at the cost that many rays might be blocked (eg. in case of a very small aperture stop).

Ideally the lens samples should be placed on the effective pupil, ie. the projection of the visibility function on some plane. This is solved to some extent in [68].

Using a wavelength-dependent refractive index leads to spectral rendering which can provide more realistic results (such as chromatic aberrations) at the cost of some more complexity. A trade-off can be made that the model is evaluated at just three frequencies corresponding to red, green and blue channels.

3.1.2 Image-based multi-view accumulation

The multi-view accumulation (MVA) method [32] exploits two facts. First, the light propagation though the thin lens acts as integration of many pinhole views with fixed sensor and variable center of projection which can be located anywhere on the entrance pupil [40]. Second, rendering of each perspective projection can be accelerated on a GPU with highly optimized rasterization.

Many pinhole images are rendered, accumulated and averaged. Each pinhole image is rendered with different camera position and projection matrix with camera position being stochastically sampled within the area of the entrance pupil. Behind the construction of the frusta is following: when one puts a point-sized off-axis aperture to the thin lens there is a pyramid-shaped cone of light from the aperture point to the rectangular sensor (see fig. 3.1a for illustration). The lens transform this pyramid into another pyramid of rays incoming from the object-space such that the rays are not refracted only if the aperture point is at the optical axis.

Since a thin lens transforms a fan of rays from a single point into another fan that intersects at a single point all the object-space pyramids share a single rectangle

which is invariant to the position of the aperture samples. Anything on this rectangle will be thus rendered sharply on the sensor. It is not surprising that it is the image of the sensor transformed by the thin lens and lying precisely on the focus plane.

This results in creating off-axis frusta [17]. Also note the relation to stereo pair imaging (for 3D vision) [16] which is a special case when the scene is rendered from only two view point and the images are displayed separately. For thin lens model without tilt-shift configuration the frustum is given by:

$$\begin{aligned} M &= M_{frustum}(x'_r, x'_l, y'_t, y'_b, z_n, z_f) \cdot M_{translate}(-x_s, -y_s, 0), \quad \text{where} \\ (x'_r, x'_l) &= -\frac{z_n}{z_{focal}}(x_r, x_l), \quad (y'_t, y'_b) = -\frac{z_n}{z_{focal}}(y_t, y_b) \end{aligned}$$

The original method utilized accumulation buffers for averaging the frames. Although they have greater bit precision than standard buffers they are not suitable for accumulating many images due to numeric errors. With today's hardware it is better to use rendering into floating-point textured via frame-buffer objects (FBO).

The result converges to the reference solution but quite slowly. Some highly visible artifacts are present during convergence as the aperture samples are constant for all pixels in each accumulated image. Due to the fact that multiple views are accumulated the visibility is handled correctly without any further artifacts other than aliasing at in-focus areas.

For producing a single output frame the original method rendered all the pinhole images and then displayed the result. In order to see the accumulation as it progresses it is possible to display the average computed from the partial sum or maintain a moving average in the accumulator.

Extension to tilt-shift configurations

The extension of this method to tilt-shift configuration was shown to be possible [10]. Unfortunately, they did not provide any details on the deriving the frusta from the geometrical situation.

In the non-tilted case we have identified that the important thing is that the frusta shared the rectangle on the focus plane which resulted in rendering those points in sharp focus (fig. 3.1b). For just shifted sensor we only need to shift the bounds of the near rectangle in addition to the shift from the lens sample to create an off-axis frustum perspective matrix.

In case the sensor plane is not parallel to the principal plane of the lens the sensor's image is a general quadrilateral instead of a rectangle (fig. 2.12b). This corresponds to the intersection of the tilted focus plane to the basic frustum with the center of projection in the entrance pupil center.

The resulting frusta should be constructed appropriately in order to maintain this quadrilateral invariant to changing the center of projection for lens samples. The near plane must be selected such that its intersection with the object half-space frustum is a rectangle. Unfortunately, besides this necessary condition we do not have the exact mathematical model.

3.2 Interactive methods

Almost all interactive methods for depth of field rendering work on sample-based representations of the scene, most often on one or more layered depth images [65], and try to solve the convolution of the plenoptic function of the scene with the visibility function in different ways. One of the exceptions is a variation on the accumulation buffer [35] which blends multiple perspective views that approximate a complex geometric lens model.

Since the convolution kernel is spatially-varying there exist two strategies of computation – gathering and spreading (or scattering) [43, 55]. In gathering an output element is computed as a linear combination of input elements with weights defined by the kernel. Conversely in spreading each input element is decomposed into a linear combination of output element using the kernel. A constant kernel used in gathering and spreading context produces the same results. In contrast, the same spatially varying kernel leads to different result in both contexts.

Also the methods differ in how they treat visibility or even if they do. Methods like image-based ray tracing (IBRT) [46, 47] evaluate the visibility function implicitly as a result of tracing of the lens systems and the scene. On the other hand filtering approaches expand visibility into visibility through the lens system and visibility in the scene. The first is described by the PSF, while the latter is resolved by digital compositing (alpha blending) [58] of layers.

Until recently there was a lack of thorough understanding of the duality between gathering and spreading filters [43] in the area of depth of field rendering. It has been shown that CoC-shaped PSFs in the area of depth of field rendering correspond to simple kernels for spreading. A simple kernel for spreading might be dual to a very complex kernel for gathering and vice versa. Since gathering filters mapped better to the computational model of GPUs they were used almost exclusively for interactive DoF rendering. Concurrent reading that is typical for gathering filter is no problem, while concurrent writing was not possible on GPUs until they recently supported atomic additions. Still it is not as efficient as reading due to the need for locking on a hardware level somewhere within the GPU.

Unfortunately, simple spreading varying kernels were used for gathering leading to bad artifacts. Many methods were proposed to solve this problem [8]. Also there were some methods which utilized rendering rasterized sprites that were supported even by older GPUs.

An interesting kind of DoF rendering methods attempts to solve the visibility problems by anisotropic diffusion [44, 52], that by doing this it simulates Gaussian PSFs, quite different from photographic lens PSFs, and produces an unrealistic bokeh. Gathering filters and anisotropic diffusion are currently used in the state-of-the-art computer games to simulate DoF since they are very fast, albeit not so realistic.

Recently, new algorithms for spreading with complexity reduced from quadratic to linear or even constant for some specific PSF types were published [43]. It has been shown that implementation on today's GPUs is possible.

3.2.1 Comparison of ray tracing and filtering methods

We can compare to the two approaches to DoF rendering. Ray tracing methods solve visibility correctly but have problems with optimal sampling. Filtering methods offer optimal sampling but have to explicitly solve visibility within scene (occlusion).

Getting a large CoC sampled properly in ray tracing requires many samples to reduce noise to tolerable levels, while for in-focus areas a single sample might be sufficient. Unfortunately, for a given pixel on the sensor we are not aware of a method of efficiently generating lens rays ordered by decreasing contribution of light intensity. Eg. importance sampling successfully employed in image-based lighting [60] cannot be readily used due to possibly complex ray transformation within the lens and limited visibility.

On the other hand spreading filters can rasterize the PSF according to the sensor resolution, spreading the light exactly to the affected pixels without noise present in Monte Carlo methods. The cost is that the occlusion has to be solved by other means.

Another view is on supported PSFs. Ray tracing methods can support arbitrary lens models with various PSFs, but have to evaluate the propagation of rays in the lens systems which might be costly. Filtering methods are limited by the complexity of PSFs and their efficient representation, evaluation and spreading.

Ray tracing of the original scene and multi-view accumulation has also the advantage over image-based methods that they provide implicit anti-aliasing via multi-sampling. Methods working with discretized images eg. have no information of an exact position of a high-light smaller than a pixel. Thus a bokeh pattern from such a highlight might be slightly translated in the image-based methods compared to a more accurate result of the multi-sampling methods. Also CoC from light sources outside the pinhole field of view might be missing there.

3.2.2 Layers and their extraction

Before we can describe the DoF post-processing methods themselves we need to learn more about their input data.

In a pinhole image (perspective projection) only the parts of the scene which are directly visible from the center of projection can contribute to the output image. On the other hand for a lens with a finite aperture even parts of the scene which are occluded in the central pinhole view can become visible in other views and thus take part in the output image. Since both the directly visible and occluded parts of the scene cannot be represented in a single image, they must be stored in several layers.

Each image represents a 2D table of samples of the incident radiance function from the scene to the center of projection. Each sample might be then understood as a single light source. Except that the sampled color (or precisely radiance) from the scene is not enough for depth of field rendering since the effect of a light source on the image also depends on its depth. Thus each layer consists of a color image and a depth image. Usually the layers store the results of frustum transformation normalized to the $[0.0; 1.0]^3$ cube.

The sampled radiance is valid only for a single direction. Assuming that the exitant radiance of scene surfaces does not vary too much when changing the viewing direction a little the sampled radiance can approximate the true radiance from

another viewpoint (on the front element of the lens) quite well. This problem can be solved with deferred shading [46] where surface properties are sampled and radiance from given viewpoint is computed later.

There are two approaches in extracting layers from the scene – depth interval layers [64, 4] and depth-peeled layers [25] – each with its pros and cons.

In depth interval layer extraction the scene is divided into disjoint intervals of depth and each layer contains the surfaces visible in that layer. This results in that the whole images are ordered by depth which could be exploited in some methods.

On the other hand depth peeling produces layers where each pixel is ordered by depth independently. The first layer contains what is visible directly, the second what is hidden after the first layer and so on. This results in fewer layers, since the number of layers is limited only by the depth complexity of the scene. The difficulty is that a patch of pixels from one surface might be interspersed in many layers.

In general the depth peeled layers provide a more compact representation than depth interval layers, since there is fewer empty areas. Thus a lesser number of layers is needed, saving some memory.

The layers can be rendered by a GPU scan-line rasterizer or with a ray tracer modified with additional depth checks, resp. taking k-th intersections instead of the first ones.

For accurate rendering of strong bokeh it is necessary that the color images in the layers are HDR images, eg. represented with floating-point numbers. The resulting output image might then be tone-mapped to LDR.

Depth peeling in practice

Depth peeling extracts layers ordered in each pixel by visibility from the camera. The basic method is multi-pass depth peeling [25], described in more detail in [14]. Extraction of each layer, except the first one, is dependent on the depth image of the previous layer. The first layer is rendered with the usual z-buffering, ie. a single depth test compares the depth z_f of a fragment from an object with the current value in the z-buffer z_b. If the fragment is further (hidden), $z_f >= z_b$, it is discarded, otherwise the (color) frame buffer and z-buffer are updated. In the following layers two depth tests are performed using the previous depth image as a secondary read-only z-buffer. First, the fragment depth is compared to the previous depth image and is discarded if it is nearer, $z_f <= z_{i-1}$. Such fragments have been already stored in previous layers. Next, the usual depth test with the z-buffer can be done.

Clearly, this way we need to do as many passes as is the target number of layers. Some newer methods exist which produce two layers in each pass or there are even single-pass methods [13, 49, 50]. According to [47] multi-pass methods are more flexible than single-pass depth peeling methods and thus are more suitable in context of preparing data for IBRT or other DoF rendering methods.

Rasterization might not be the only source of depth-peeled layers. Although no description of such a method was found in literature ray tracers might also be theoretically modified to produce depth-peeled layers. The renderer would produce n output layers in a single-pass, each with output from ray intersection of different order. Besides color images also depth images would be produced.

The depth images usually contain values normalized into interval [0.0; 1.0] (near to far plane). An information to convert it to absolute depth might be needed to

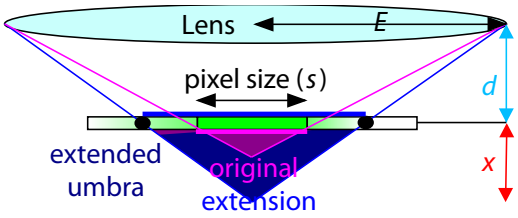


Figure 3.2: Situation in the extended umbra depth peeling, a sketch courtesy of Lee et al. [47].

be passed along. Storing absolute depth, eg. as a float, is possible but some height field intersection algorithms might still expect depth range [0.0; 1.0]. The conversion formulas between camera-space depth and z-buffer depth can be inferred from the perspective matrix with the additional rescaling from the [-1;1] to the [0;1] interval:

$$\text{cameraToBuffer}(z_c, n, f) = \frac{f(z_c + n)}{z_c(f - n)}, \quad \text{bufferToCamera}(z_b, n, f) = \frac{fn}{z_b(f - n) - f}$$

Extended umbra With the knowledge of the entrance pupil size (or its bounding circle in case of non-circular pupil) it is possible to predict which areas of the further layers will not be visible to any lens ray going from such a pupil. If we interpret pixels in the layered depth images as tiny squares and the pupil as an area light the space in the full shadow (umbra) will be unreachable to any such a lens ray. Fragments within such umbras need not to be extracted by depth peeling. This reduces to introducing an offset to the depth peeling fragment test [47].

They go even further by reinterpreting the geometry to extend the umbra to the maximum size without creating artifacts from the lack of data in IBRT. The result of both modifications is they avoid shading unnecessary fragments and allow to skip those areas easier later during intersection testing.

Extracting depth interval layers in practice

Having floating-point z-buffers and frame-buffer objects extracting layers of disjoint depth intervals is similar. The only thing is how to choose the interval spacing. It seems that best results can be obtained not with uniform spacing with respect to depth but rather to CoC size [8, 46]. Since there is no dependency in extracting layers into disjoint depth intervals (in contrast to depth peeling) it is possible to render multiple layer in one pass [46]. In contrast to depth peeling it is possible that from two objects, one occluding the other, within one depth interval will be rendered only the first visible. However there is a high probability that the occluded object would be culled in extended umbra depth peeling. So that artifacts from missing geometry should be rare.

3.2.3 Image-based ray tracing

We now describe the main method implemented in the thesis project. The idea is to synthesize new views on the scene from a single-view data and do Monte Carlo sampling. Either depth interval layers and depth peeled layers can be taken as the view-dependent scene representation, but the details of their processing differs.

Compared to multi-view accumulation synthesizing new views in IBRT does not require rendering the whole scene over again but achieves it purely with depth layer images. Also per-pixel lens aperture sampling is possible. Compared to filtering methods IBRT solves visibility by ray tracing instead of compositing. Since it is in fact a gathering method it maps well to a GPU.

The first method [46] works on depth interval layers and assumes the thin lens model, while the second one [47] on depth-peeled layers and is generalized to complex geometric lens system. The basic structure of the DoF rendering process is the same:

1. the layered depth images are extracted according to the paraxial approximation of the lens model - an object half-space frustum is constructed using the sensor extents and entrance pupil center
2. image-based ray tracing is performed on the layers
3. the resulting image is displayed

During the IBRT phase rays are traced backwards from the sensor, through the lens (using exit pupil samples) to the scene represented by layered depth images treated as height fields. Contributions of computed radiance from all rays into each pixel are averaged.

When an intersection is found the incident radiance can be evaluated. Basically, we can just take the shaded color from the color image in the corresponding layer. Better results can be obtained by view-dependent deferred shading [47]. In this case the surface properties instead of plain color are extracted into the layers, and shading is performed with those parameters and the direction of the incident ray.

The interesting parts are computing intersections of a ray with a multi-layer height field and generating rays from the lens, in particular computing the pseudo-random lens samples and evaluating the lens model.

Height field intersection

The idea of ray-tracing height fields is not new. It arose naturally eg. in rendering terrain from measured elevation data [18]. Then it appeared again in relief mapping [57, 56, 3]. The input data were originally assumed to be a plain single layer. Multi-layer height fields were supported in [56] with the assumption that the height fields represent closed objects where the terms "inside" and "outside" make sense.

There are several approaches to doing the intersection. Mostly they interpret a discretized height field as a surface obtained by bilinear interpolation. The first method [57] treats the values of the height field on the projection of the ray on the height field as a 1D function and searches for intersection by a binary search. This gives wrong results in case there are more than one intersection under the ray. To alleviate this it adds before binary search a phase of linear search in constant increment which roughly identifies the intersection and the binary search perform a refinement.

For a height field with large depth discontinuities and rays of large angles of incidence even the linear search phase might give wrong results. This can be to great extent solved by traversing the rasterized projection of the ray on the height field – *ray footprint*. A small error is still possible for some special height fields which vary non-linearly even within a single pixel. Unfortunately the complexity is now linear to the length of the ray footprint, instead of logarithmic as for the binary

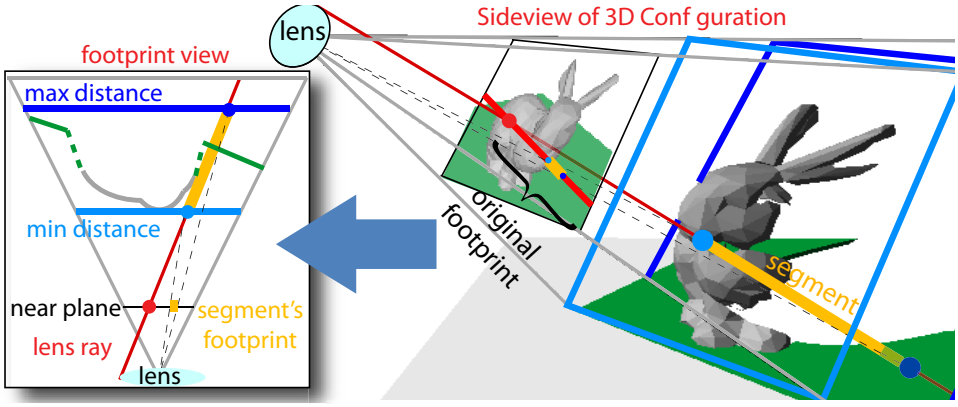


Figure 3.3: Ray footprint clipping, a sketch courtesy of Lee et al. [47].

search method. The binary search can be made robust at the cost of some heavy precomputation which prohibits real-time usage [3].

In context of IBRT we can assume the rays are almost perpendicular to the depth planes in the height field since the front element of the lens is quite small. This allows us to use a more robust method then in relief mapping without excessive amount of error.

Intersection with depth interval layers is simpler. The height field in each layer is from definition bound by some depth interval. Thus the ray can be divided into segments, each tested against a particular depth interval. Due to all the assumptions in [46] they use a simple method of finding intersection similar based on binary search.

In order to accelerate the intersection testing for blank areas it is possible to skip some tests for rays that pass in front of the geometry at that layer [46]. All lens rays for a given sensor pixel go through a CoC at given depth. For thin lens this can be computed analytically, for complex lens system this can be approximated by a bounding box. As the CoC size increases monotonically with distance from the focal plane the maximum bounding box within a depth interval is at one of its ends. Nevertheless, if the minimum depth of the current height field within the larger CoC bounding box is beyond the current depth interval the layer is empty in this region, so there cannot be any intersection, thus it can be skipped.

The minimum depth within a rectangular region of a depth image can be efficiently evaluated using N-buffers [23]. This data structure is similar to mip-maps with the difference that all levels are of the same size. Instead of providing an average value over a region it can provide minimum or maximum values over a rectangular region. Both construction and evaluation of N-buffers is very efficient and can be easily implemented on a GPU.

Intersection with depth-peeled layers is a bit trickier. In case the objects are not closed we can no longer assume that the neighbor pixels belong to the same object or an object in a similar depth – there can be large depth discontinuities. So that interpolation and a simple binary search might give wrong results. Again assuming the lens rays are almost perpendicular [47] showed that the exact computation is not necessary. N-buffers can be utilized again but in a slightly different manner. For a given ray we can find minimum and maximum values of the height field layers under the bounding box of its footprint. With those depth bounds the ray can be clamped (see fig. 3.3). Two N-buffers – with min/max – values can be used for

finding minimum/maximum depth within a region of a height field.

However, bounding footprint of each ray separately would be not too efficient. Instead one can compute the footprint of the whole bundle of all lens rays for a pixel and clamp them all at once.

After the rays are clamped to sufficiently short footprints each ray can be tested for intersection pixel by pixel. For multiple depth layers each has to be compared to the ray depth. The details on height field intersection algorithms are provided in the section 4.1.4.

There is also another interesting approach for accelerating height field intersection with run-based pixel traversal [37]. The question is if it is possible to modify it for our height field data.

Lens ray generation

Before rays can be intersected with the height field they have to be generated. For each pixel we have to generate rays according to the required lens sample count. Each ray starts from within the pixel area. Ideally this should be sampled by jittered unit square samples for anti-aliasing. Since the layered depth images are already discretized we cannot get more information from the scene, so in practice this sampling might be useless and the center of the pixel might be sufficient. The pixel samples have to be transformed to the camera space according to the sensor size, position, shift and tilt. This can be done via a 3×3 transformation matrix.

Also the lens have to be sampled. In general we should sample the effective pupil. Since it is hard to compute for complex lenses we might sample the back lens element surface. For thin lenses we can sample the exit pupil. In case the aperture is located on the principal plane or nearer to the sensor the exit pupil is the same as the aperture.

Having the ray from the sensor to the lens it can be transformed via the lens model to the ray towards the scene. Since the height field layers are in the frustum space the ray has to be also transformed via the frustum transformation to this space.

3.2.4 Spreading filters

Spreading filters are dual to gathering filters known from convolution [43]. A spreading filter behaves conceptually similarly to a light tracer, except that it does not work with individual rays but rather with whole ray bundles going through the lens. Each pixel of the source image is treated as a point light source[1] and its PSF is projected onto the sensor. For a thin lens model with a circular aperture this might be seen as cutting a cone of rays with the sensor plane producing a conic section. The PSF is computed using a lens model controlled by the depth image.

This is similar to linear filtering known from Fourier optics. The assumption there is, on the other hand, that point light sources are all in the same depth, so that no occlusion can happen. To solve occlusion naively for an ordinary scene this would mean slicing the scene into as many layers of the same depth as there are light sources of different depth. Each layer would be filtered independently and the

[1] no matter whether in the original scene it emitted or just transported the light

then layers would be alpha-composited from back to front. Obviously for common scenes the number of layer would be huge and each of them would be almost empty.

Nevertheless, the layering approach gives a theoretical background to what is used in practice. The scene is partitioned into a small number of disjoint depth interval layer, each of them being filtered separately and composited. The computation is practically feasible, although discretization problems on interval borders have to be addressed [12, 11].

Since the memory consumption of depth interval layers is not ideal and due to the mentioned artifacts it was proposed to filter layers extracted similarly to the depth peeling process, while doing a per-pixel comparison [48]. The output of filtering a single layer was divided into three layers by relative visibility to the next layer, thus solving partial occlusion to some extent.

Traditional brute-force spreading filters [59] have computational complexity linear to the PSF area, ie. quadratic to its radius. Also, in contrast to gathering filters, they exhibit scattering memory writing patterns, until recently not well supported on GPUs. For a long time, their long execution times put off spreading filters to non-interactive uses. An example of a trick of doing scattering on a GPU was the usage of point sprites [48]. Still the quadratic complexity was prohibitive.

Recently, some fast spreading filters were introduced which successfully reduce the complexity for some special but useful classes of PSFs to be linear to the PSF perimeter or even constant. This is very suitable even for great amounts of blur. Also thanks to the advances in GPU architectures scattering write patterns are now supported. Parallel implementations on GPUs are now possible [43].

Also it should be noted that for filtering a scene consisting of light sources (again either emitters or transporters) in different depths the PSF size and shape might be variable. This prevents Fourier transform to be applied to compute convolution via multiplication in the frequency domain, thus reducing the quadratic complexity of convolution to $\mathcal{O}(N \log(N))$ for a Fourier transform and its inverse. Nevertheless, some methods exist doing this for multiple layers assumed to have an approximately constant PSF [64]. Fast spreading filters support variable-sized PSF natively.

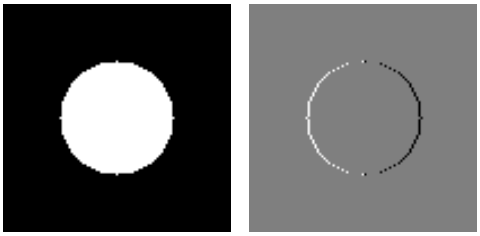
For depth of field rendering the PSF shape and size can be controlled by a lens model, eg. the CoC size computation in the thin lens model, and a depth image. The PSF must be rasterized, either in a precomputation step or on-the-fly. Care must be taken to maintain the correct intensity levels in the presence of discretization [48].

The resulting spreading algorithm for a single layer looks like this:

- for each pixel in the input image:
  - read the input light intensity I
  - find out the PSF size and shape
  - get the corresponding PSF rasterization
  - for each pixel in the rasterized PSF:
    - find out the proper weight W
    - write the IW at the corresponding position in the output image

Fast spreading filters

The problem of ordinary spreading filters is that for spreading th PSF of a single input pixel too many output pixels can be written. Even if scattering is done on a



(a) Circle (b) Circle diff

Figure 3.4: Discretized circular PSF and it horizontal finite difference (in false colours).

GPU (if it is supported) it still incurs locking. The goal is to reduce the number of written pixels, while still producing the same output image.

The key are several observations made by [34] and [43]. Spreading PSFs of many input pixel is equal to summing multiple 2D discrete functions. Differentiation of a non-constant function followed by integration leads to the original function, ie. $\int f' = f$. Both differentiation and integration are linear, ie. $(f + g)' = f' + g'$ and $\int(f + g) = \int f + \int g$, leading to $f + g = \int(f' + g')$. Differential calculus can be successfully replaced by finite difference calculus for 2D discrete images. And the key observation is that n-th derivative of a 2D polynomial of degree n is almost zero, except for a small set of non-zero values, still it holds all the information needed to reconstruct the original function.

The result is that for spreading polynomial PSFs, including constant-valued rectangles, tent functions, etc., it is sufficient to spread the n-th discrete difference of the original PSF and then perform n time integration of the whole image [45, 34].

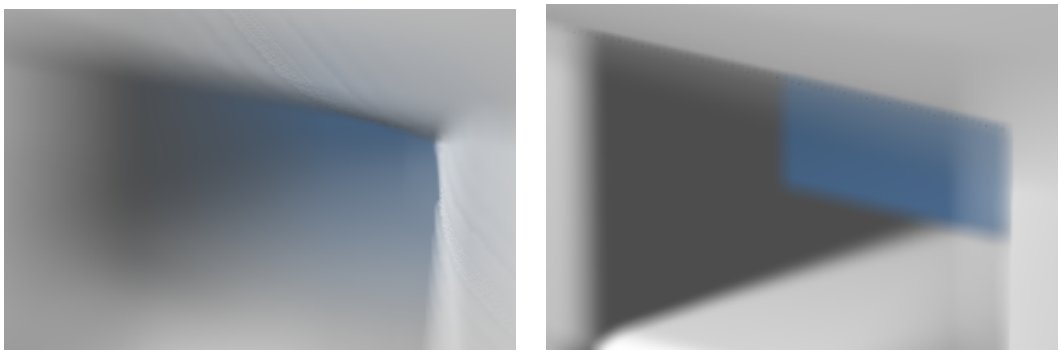
Also for 2D PSFs which are separable (into two 1D functions) both integration and differentiation can be made separable, ie. consisting of a horizontal and vertical 1D phases.

Discrete integration can be implemented by well-known summed area tables (SAT) and even an efficient GPU implementation is possible with the parallel prefix-scan approach [38].

In case the PSF is a constant box function, its 2D separable discrete difference contains just four non-zero values corresponding to the corners. This is the reason why such a *rectangle spreading* filter can produce arbitrarily large rectangle-shaped PSF in constant time.

To represent PSFs more similar to photographic diaphragms, such as circles, regular polygons, etc., there might be no separable representation. At least it is possible to differentiate such functions in one direction (eg. horizontally). This leads to *perimeter spreading*. Only the edges are spreaded which results in linear complexity with respect to the PSF radius.

Hybrid spreading The perimeter spreading filter can mimic the aperture shape closely, but it depends linearly on the PSF radius. The rectangle spreading filter is fast but is limited to rectangular PSF shape, which produces bad-looking bokeh. Fortunately, for low-contrast areas of the source image the shape of the PSF has significantly lower visual impact than for high-contrast and high intensity areas. Thus it is possible to make a hybrid filter which applies the high-quality perimeter



(a) Focus at foreground (b) Focus at background

Figure 3.5: Visibility artifacts due to a single layer (example from spreading filters applied on the cubic structure test image). Intensity leakage (3.5a) – background is blurred on top of in-focus foreground. Depth discontinuity (3.5b) – no background can be seen under out-of-focus foreground (a lack of partial occlusion), producing a sharp edge.

spreading only for areas where it makes sense and fast rectangle spreading for the rest [42].

For a given pixel in the original image the hybrid filter selects one of the two filters via a heuristic criterion based on local contrast. For high-contrast, high-intensity pixels it selects the high-quality filter, while for the rest the lower-quality fast filter. One of the local contrast metrics can be the thresholded difference between the current pixel intensity and average intensity of its neighborhood of some size. This is similar to thresholding an edge-detected image. The average value of a pixel neighborhood can be efficiently evaluated from a SAT of the original image.

Layers The original paper [45] shows that the fast spreading filters can be applied to a set of depth-interval layer and then alpha-composited in order to solve visibility, and thus to avoid intensity leakage and depth discontinuity artifacts (see examples in fig. 3.5).

On the other hand, the usage of fast spreading filters for depth-peeled layers should be explored. Recall their advantage of the need for fewer layers compared to depth-interval layers. Per-pixel layers [48] seem to be suitable for spreading filters since in the original a kind of spreading was done, albeit by different means. Unfortunately, [42] only mention per-pixel layers without providing details whether they in fact used per-image or per-pixel layers.

Per-pixel layers seem not to be fully compatible with precomputed rasterized PSF differences since some per-pixel computations has to be made. Note that for deciding the output for a pixel of the rasterized CoC we need to know the source and destination pixel depths. The information on source pixels is cannot be stored during the phase of spreading of PSF differences. A way to combine those two methods which seems to be possible is the following procedure:

1. take the original rasterized PSF
2. for each of its pixels make decision to which output layer it should go, which produces three clipped PSFs
3. differentiate each of them

4. spread each of them into the corresponding layer

Ie. spreading could be done after deciding the output layer and for each part of the PSF separately. Solving this problem was out of the scope of the thesis. Some more research is certainly needed in this area.

3.3 Proposed ideas

Although we concentrated mainly on creating an interactive depth of field renderer, during the course of the thesis project some ideas and questions came to mind which might eventually become a basis of some further research. The main question that arose was how to support complex lens models in the specialized interactive methods, in particular in image-based ray tracing and fast spreading filters.

Treating a complex lens as a black-box function and the consequences for ray tracing are suggested. A prototype has been created showing that the this direction is viable and is worth being explored further. Also the possibilities of supporting a complex lens within spreading filters is discussed.

3.3.1 Lens ray transfer function

Complex lens system differ from thin or thick lenses in that there is no analytical formula for figuring out how incoming rays get transferred by the lens. Instead, the ray traversal is simulated by ray tracing though the lens elements. This computation is obviously more expensive as it encompasses a sequence of ray-sphere intersections and refractions. It is linearly dependent on the number of element surfaces. According to [47] ray generation for simple lens models is not a bottleneck in IBRT (rather memory transfer bandwidth is), but the measurements comparing performance for simple and complex lens models are not available. On the other hand, more complex models might be harder to implement on a GPU. Also tracing rays in lens elements with aspheric surfaces or even with a gradient refractive index might be far more computationally expensive. Note that a ray transfer must be computed for each sample in each pixel on the sensor. It is natural to try to accelerate this computation.

The other motivation is that it might be very useful to conceptually treat the complete lens system as a black box which just accepts incoming rays and produces outgoing rays with the internals being hidden. Lens designers have no motivation to publish their design data and publicly available are mainly historic designs. Also the black box concept does not implicitly constrain any particular internal ray traversal model and its implementation.

Discussion

The natural idea is to represent the ray transfer through a lens system by a mathematical function. Generally, we will refer to it as a *lens ray transfer function* (LRTF). This is a similar concept eg. to radiance fields mentioned in the theoretical chapter. In case we assume perfect non-scattered transfer such function can be a mapping from incoming rays to outgoing rays. For absorbed rays there might be undefined values. When taking scattering into account we would end up with a

function quite similar to BSSRDF[2] [70] which describes the amount of light energy transferred for every pair of (incoming and outgoing) rays.

Sticking with the simpler mapping concept a question arose how to parametrize the rays. In general a ray in a 3D space is defined by a 3D position and 2D direction[3]. Fortunately incoming or outgoing rays must intersect the front or back lens surface, thus constraining a single ray position on a 2D surface. This reduces the total dimension to four degrees of freedom. Since the lens surfaces are finite also the range of the parameters is finite.

We will discuss the possible parameterizations further. Before that we outline the representation and usage of the LRTF. The ultimate goal is to be able to efficiently evaluate the function without the need for the full ray tracing of the lens. The most straightforward way of representing a general function is sampling it into a table, either by precomputation or by measurement. A 4D function with 4D values still presents a problem with high dimensionality. Thus we must find structure in the function, based on some assumptions on the lens system it represents, which can be exploited to compress the function. At best the function could be decomposed into some analytical parts and some coefficients. A trade-off must be found to reduce the number of coefficients while maintaining the evaluation not too complex.

On simple assumption which can reduce the range of the LRTF by one dimension and which holds for most of the photographic lenses is rotational symmetry around the optical axis. Thus a rotation transformation can be factored out. This leads us to a 3D range and a possibility of sampling the LRTF in a 3D table of 4D vectors which is already feasible. To be precise iris diaphragms (important for a great effect on the bokeh shape) are usually not circular but also do not modify the direction of rays, they only have effect on the absorbed rays, ie. the undefined values. Thus their effect could be also factored out.

The minimum number of samples for acceptable reconstruction depends the frequency spectrum of the original function as described in the Nyquist theorem [29]. Thus at best each of the LRTF components should be smooth and without discontinuities. As in common lens system the rays can get clipped by several apertures the sharp transitions between defined and undefined areas cannot be eschewed in general. A suitable parametrization should also take sampling in consideration in order to sample important parts of the function well.

It should be noted that [68] briefly suggested the idea of representing the ray transfer with a function and the possibility of evaluating it with a texture lookup, but completely independently and without any details.

The LRTF and its tabular representation is not the only possible way of capturing the lens behavior. In optics [15] the theory of wavefront and ray aberrations deals with representing how a particular lens differs from an ideal lens. The good thing is that wavefront aberrations can be efficiently represented by coefficients of Zernike polynomials. This could save quite an amount of memory and reduce sampling and interpolation artifacts. Also the coefficients have a physical meaning, each one controls the amount of a single optical aberration or a family of aberrations, such as the spherical aberration.

The functional representation of ray transfer behavior of lens systems might not be only useful for acceleration purposes. It also allows rendering with data based on

[2]Bidirectional surface scattering reflectance distribution function.

[3]Consider spherical coordinates.

measurements of ray transfer in physical lenses (without having the geometrical a material design data). Moreover the functional lens descriptions could be modified after capturing or be synthesized non-physically. Being able eg. to interactively control the amount of various aberrations might be quite interesting.

Further generalizing the lens models results in higher dimensionality, but probably some redundancy could be exploited for compression. We could theoretically allow eg. dependency on wavelength or zoom lenses with variable element positions.

The basic parametrization

We describe the basic parametrization which has been implemented in the software prototype but has some drawbacks.

The LRTF describes the behavior of a lens. It is a 4D to 4D mapping from incoming ray to outgoing rays. Each ray can be described by four parameters: hemispherical coordinates (θ, ϕ) of the ray origin on the back or front hemispherical cap surface of the lens and hemispherical coordinates of the ray direction (in a frame local to the ray origin with hemisphere pole aligned with the surface normal – pointing outside the lens).

For lenses with rotational symmetry around the optical axis the LRTF is also rotationally symmetric and the input position ϕ might be separated. Then the evaluation consists of three phases:

1. rotate the input (position and direction) so that its position ϕ is 0
2. evaluate the LRTF (by a texture lookup)
3. rotate the output using the original position ϕ

This parametrization assumes that the rays are defined by positions of the back or front element surfaces themselves. This means that transforming the ray direction to the local frame and back needs a general coordinate transformation. In case the ray could be defined by a point on a plane instead of a curved surface, the transformation could be simplified to a plain rotation and also representing the direction in the local frame would be simpler.

A better parametrization

The previous parametrization has a singularity at the pole of the hemispherical coordinates for ray direction. This results in poor sampling.

A better candidate for representing a direction on a hemisphere seems to be hemispheric stereographic projection [20] of the direction onto a circle. It could also avoid the usage of trigonometric functions. The cost could be some memory inefficiency (the space within a bounding square outside the circle would be unused).

Effective pupil sampling

Being able to evaluate the lens ray transfer function efficiently is only halfway towards efficient ray tracing. As pointed out by [41] and [69] in particular the effective pupil[4] should be sampled in order to minimize the number of absorbed rays. [69] precomputed for each pixel for a single static lens-sensor configuration a bounding circle of the effective pupil projected on some plane and then sampled those circles.

[4] the image of the visible window towards the scene from a particular pixel position

In context of interactive rendering this should be precomputed in a similar way to the LRTF. Using bounding circles for approximating the effective pupil leads to a very compact values - only its 1D radius and 2D position of the center on the projection plane is needed. The function depends, however, on the viewer's (pixel) 3D position which could be unbounded. Thanks to assuming rotational symmetry of the lens we can also get rid of one dimension. In order to make the position range finite we can constrain the sensor to lie within a finite area, either a box or probably more better a frustum (to compensate that the effective pupil might not change as much at large distances).

This leads to a 2D table of 3D vectors, which can be easily stored on a GPU into a texture. In case of a great asymmetry a 3D texture would be probably needed. The benefit would be that such a precomputation would not be necessary for each frame when only the sensor moves. Changing the aperture size, on the other hand, would need a recomputation. A question is if the recomputation of whole table is really necessary or if it is possible to reuse a single table with multiple aperture sizes.

A more interesting open problem is how to efficiently represent the exact effective pupil function.

3.3.2 Spreading with PSFs of complex lens models

This subsection discusses some directions for further research which have not been yet implemented and tested in practice.

The full spreading algorithm [59] uses the PSF of the thin lens model with diffraction at a circular aperture taken into account. The PSF is assumed to be invariant on the xy position of the light source and to be dependent on its z depth and intensity (which is just a scaling factor). Thus it is possible to sample the PSF into a set of 2D textures at multiple depths, producing a single 3D texture.

In contrast complex lenses with optical aberrations and/or vignetting lead to spatially variant PSFs. This would increase the dimensionality by two degrees of freedom. In case we assume rotational symmetry of the lens and thus the PSF around the optical axis this would add only one degree of freedom at the cost of introducing some aliasing artifacts from the rotation of discrete samples. Still the memory requirements might be great. The PSFs in such a setting would not be precomputed by an analytical model but via light tracing (forward ray tracing) of point light sources though the complex lens. Tilt-shift configurations would lead to even more variable PSFs, as discussed in the following subsection.

A possible alternative would be to project the PSFs onto some basis and rasterize them on-the-fly or exploit similarity between the PSFs to compress them.

An analytical model or ray tracing of complex lenses might not be the only source of PSF data. In vision-realistic rendering the wavefront aberrations of a human eye can be measured, fitted with Zernike polynomials and converted to PSFs [5, 4]. The PSF was again assumed to be varying only with depth of the light source, thus it was measured only at a single point of the eye. Also the conversion of wavefront aberrations to PSFs could be quite computationally intensive, which could prevent generating the PSFs interactively. This method could be potentially generalized to measuring wavefront data from real-world lenses to obtain PSFs for spreading.

An interesting open question is whether orthogonal polynomials could be used

instead of ordinary polynomials for representing PSFs in spreading filters. [45] reported that using higher-order polynomials (for more precise representation of complex PSF) is limited by numerical precision caused by a high range of values. On the other hand orthogonal polynomials keep their values in a small interval (eg. [0; 1]). It should be explored how they would cope with separability and with repeated discrete differentiation and integration.

Spreading for tilt-shift configurations

So far correct tilt-shift DoF rendering has been done either by multi-view accumulation or (image-based) ray tracing. Also a physically incorrect approximation based on filtering appears widely in consumer software. It just utilizes an artificial blur map which controls the PSF size, nevertheless the PSF remains the same as for the basic non-tilted variant. The results lack a perspective transformation due to non-constant sensor depth, too.

This section discusses some aspects of PSFs in tilt-shift configurations and suggests a more physically correct approach based on spreading.

For tilted configurations the PSFs from a circular aperture might not be circular. Given a cone of rays from a point light source P in the object half-space on the optical axis a thin or thick lens with a circular aperture centered at O transforms the cone into another double-cone of rays in the image half-space with the apex at P^*. The shape of PSF on the sensor is then formed by intersecting the cone with the sensor plane leading to be an arbitrary conic section, most often an ellipse. The intensity depends both on the irradiance distribution in the cone and the incident angle.

Note that for a point light P, off the z axis, the cone is skewed in the (xy plane), not rotated. Thus its sections maintain the aperture shape. Intersections with other planes also lead to conic sections, albeit scaled.

Representing such non-constant PSFs with quite variable shapes in a rasterized could be quite difficult for both fast and full spreading (ie. with and without PSF discrete derivatives). Otherwise they would have to be represented analytically and rasterized on-the-fly.

Remind that the lens transforms the incident radiance function of the object half-space into the incident radiance function of the image half-space. Both are dependent on angle and position. In case we assume the exit pupil subtends a small solid angle from the pixels on the sensor we might approximate the incident angle from the center of the exit pupil.

We can represent incident radiance function approximated this way in a region of the image half-space volume by computing a set of non-tilt images at uniformly sampled depths and without vignetting. Those can be stored in a 3D texture. Views from a tilted sensor can then be synthesized as slices from such a texture by interpolation. Afterwards the effect of vignetting can be approximately computed via the $\cos^4$ law [41] modified for tilted sensors. The sensor normal might not correspond to the optical axis. So for a direction of the ray from the exit pupil to the sensor pixel the angles of incidence to both the sensor normal and the optical axis might be different.

There is a cost for preprocessing for a single camera position and orientation – the rendering of many non-tilted slices. But afterwards new views with tilted or shifted sensor could be synthesized via interpolation from a 3D texture, trading-off the cost for spreading with complex PSFs for memory storage. Also the range

of tilting and shifting of the sensor is limited by the sampled region of the image half-space. Due to the preprocessing step the method is not suitable for dynamic scenes.

Solving visibility in fast spreading filters

So far visibility in DoF rendering methods based on filtering was either unsolved or solved by compositing of depth-ordered layers. Another possible approach which should be explored is the combination of spreading filters with ray tracing to solve visibility.

Given two objects in the scene from which the light goes towards the sensor some of the light from the further object might be blocked by the nearer object. This results in that the PSF of the occluded object get clipped. Ordinary spreading filters assume no occlusion and work with full PSFs. The proposed filters would be modified such that for a given source pixel (a light source) a PSF is retrieved and before being spread into the output image it is in addition clipped.

The process of clipping should be based on sampling the aperture and testing visibility of the light source by image-based ray tracing. As the input data for spreading are layered depth images they can be treated as height-fields. And the visibility test is just comparing the position and layer of the first intersection and the light source pixel. The result would be a mask for clipping the occluded areas off the PSF.

The difficulties which have to be overcame are the following. The height-fields should be interpreted better than with nearest-neighbor interpolation to reduce blocky artifacts on edges. The aperture sampling should match or slightly exceed the target rasterization of the PSF. As the number of visibility ray would depend on the PSF area a PSF with the rasterized size over some limit should be sampled coarsely.

This way all image layers could be spread into a single output image without layer compositing.

Chapter 4

Implementation

We focused mainly towards implementing an interactive DoF renderer, which is codenamed BokehLab. Along this we developed a comprehensive set of prototypes, mathematical models and visualizations to test some of the ideas outlined in the previous sections – the sequential ray tracer of complex geometrical lens systems and the fast spreading filters, just to name a few.

We have chosen to create the software in the C# language and Microsoft .NET Framework environment. Together they provide features which help rapid prototyping and focusing on the actual implementation instead of distracting the programmer with many technological complications. The utmost performance is not critical here and is rather limited by the GPU.

For utilizing a GPU the OpenGL toolkit was selected for being platform independent. The OpenTK project[1] then provides a mature OpenGL binding for .NET along with a vector mathematics library. The shader code is written in GLSL. Mathematical models and visualizations were mainly done in the Wolfram Mathematica software. 3D models are loaded by the Meshomatic[2] library.

4.1 Interactive DoF preview renderer

The main goal of the thesis was to implement an interactive renderer to preview the effects of depth of field for synthesized imagery. Two methods were selected: the accumulation of many pinhole views [32] and image-based ray tracing [47].

The first method is considered a reference one as it is simple and converges to the correct solution. On the other hand it might take a considerable time to converge, should be dependent of scene complexity and is not capable of handling complex geometric lens models.

The latter represents the state-of-the-art. With all the accelerations and on recent hardware it should be able achieve real-time rates even for very complex scenes. Moreover it was shown to support a simple geometric lens, a biconvex lens, to produce optical aberrations. Ray tracing is flexible enough to provides a room for being further extended.

Since both methods are not real-time on a several-years-old GPU, the first by itself and latter without all the proposed accelerations, we have enhanced both methods with incremental rendering. In practice this means that the estimate of

[1] `http://www.opentk.com/`

[2] `http://www.opentk.com/project/Meshomatic`

the converged image is accumulated over many rendering cycles and the moving average is continuously displayed during the accumulation. This enables users to quickly navigate through the scene and manipulate with the camera parameters as well as to obtain a result of a longer computation. Moreover the GPU might limit the time for one rendering cycle (eg. to 1 sec) and longer computations must then be split otherwise the GPU driver restarts itself.

The interactive renderer is based on GPU rasterization via OpenGL. In addition to the DoF preview the original pinhole view, ie. the plain output of the rasterizer, can be displayed for comparison.

4.1.1 Overall organization

The interactive renderer is located in a single project, `BokehLab.InteractiveDof`. The main class is the `InteractiveRenderer` which handles the OpenGL and input events, coordinates the various rendering modules and definitions of the scene and camera.

The rendering methods are split into several classes, rendering modules, as they consist of multiple separate phases that can be invoked separately. Another purpose of rendering modules is handling their life cycle – initialization, enabling, disabling, destroying. This way it is possible to switch between the rendering methods at run-time.

The `Navigation` class holds the extrinsic camera parameters such as position and orientation and enables their interactive manipulation. The `Camera` class holds the intrinsic parameters and the camera model. Currently the thin lens model with a tilt-shift sensor is utilized. It provides the IBRT with general parameters for its own lens model and the corresponding perspective matrices for rasterization.

The scene is represented by the `Scene` class which is responsible for providing the GPU with data for rendering a single frame. It can be invoked directly or from the various rendering methods.

4.1.2 Multi-view accumulation

The multi-view accumulation method described in section 3.1.2 was implemented in two different ways. The original [32] utilized the accumulation buffer, a hardware buffer which can be used for efficiently adding the contents frame buffer with some weight. Each rasterized view has its own modified perspective and model-view matrix.

There are three possible approaches to computing the average of n frames:

1. add each frame with weight $1/n$
2. add each frame with weight 1 and finally multiply the buffer by $1/n$
3. make a linear interpolation of each frame and the buffer: multiply the buffer by $1 - 1/i$, add a frame with weight $1/i$, for $i \in [1; n]$

The third approach is used in the `BufferAccumulator` class since it maintains the moving average in the accumulation buffer.

The accumulation buffer has problems with numerical precision when accumulating many samples since it supports only integer pixel format, even though with higher bit-depth than a plain frame buffer. For 24 or 66 images presented in the

original paper the precision was probably sufficient but with hundreds or more images needed to remove all visible artifacts it is not. Moreover integer representation is not quite suitable for HDR rendering. A solution is in using different buffers with floating-point pixel format. Fortunately today's hardware offer a suitable tool – Frame Buffer Objects (FBO) that enable rendering into ordinary textures. 32-bit floating point pixel format provides enough precision even for thousands of accumulated frames.

The accumulation with FBO (in class `FboAccumulator`) is a little bit more complex than the trivial `BufferAccumulator` but is able to overcome the numerical precision problems. It needs four textures and a shader program and works in two phases. One texture acts as the render target for the current frame, two textures maintain the moving average and the last texture is a depth buffer. First the current frame is rendered into a texture which is then blended into the moving average via a fragment shader. As a texture cannot be read and written at the same time we must use two textures in a ping-pong way. The current moving average is read from one texture and the resulting blend with the current frame is rendered into the other texture. Their roles are swapped after rendering each frame to be accumulated.

The camera model used in this implementation is the thin lens model with circular aperture at the principal plane and with a sensor supporting only a shift. Perspective frustums for each view have centers of projection at sample points within the entrance pupil in this case coinciding with the aperture. The aperture samples are precomputed with jittering and concentric mapping and reused for each frame.

Incremental rendering

For the aforementioned reasons the accumulation is generalized to span multiple displayed frames. The general accumulation algorithm is thus separated from the actual underlying mechanism (accumulation buffer, FBO) and located in the `IncrementalRenderer`. It is possible to specify the number of accumulated frames per each rendering cycle. In case it is equal to the total number of views it reduces to the plain non-incremental rendering. This class has been reused to support incremental rendering even for the image-based ray tracing method. Lens pupil samples are precomputed and also shuffled to maintain a balanced sample distribution even temporarily.

4.1.3 Depth peeling

Depth-peeled layers consisting of color and depth images are the basic input data for IBRT. Since they can be also processed in different ways than with IBRT, such as directly visualized or potentially post-processed in some other methods, it is not a bad idea to separate the depth peeling phase into a separate class, the `DepthPeeler`.

The basic multi-pass front-to-back depth peeling described in section 3.2.2 is implemented as follows. In each pass a single layer is rendered into the corresponding color and depth texture attached to the FBO. Since the first layer cannot be occluded the scene is rendered as is. For the latter layers a second depth test is needed. This can be easily done in a fragment shader with the previous layer's depth texture serving as the second z-buffer. The fragment shader `DepthPeelerFS` just filters out all fragments in front of the previous pixel with the GLSL keyword `discard`.

The depth peeling process has been greatly simplified with the advent of the render-to-texture capabilities and since the depth textures could be represented with floating-point numbers instead of integers. The original method utilized shadow mapping for the secondary z-buffer [25] and the non-linear mapping of the z-buffer values had to be compensated in order to get rid of depth discretization artifacts due to interpolating in different spaces.

Since the depth images are single-channel multiple layers can be packed into one image after depth peeling in order to reduce memory transfers in further processing, such as IBRT. When reading a texture value on a GPU up to four channels can be read at once. This was suggested by [47, 56]. The implementation is in the `DepthPackerFS` fragment shader.

We can also convert the depth values from 32-bit floats to 16-bit half floats to save some memory. Some GPUs support only 32-bit float depth buffers but do not have problem with half-float read-only textures. After the depth peeling phase the depth precision is not that critical. Probably even an integer pixel format would suffice.

Our depth peeler supports extracting an arbitrary number of layers. Since in practice four layers are enough in IBRT, the implemented depth image packer and IBRT intersection routines work with exactly four layers. Both can be easily extended to also support an arbitrary number of layers. The multiple color and depth layers are stored and then easily indexed using OpenGL's array textures.

Umbra (or extended umbra) depth peeling [47] was not implemented due to time constraints. It would probably bring a considerable performance improvement (especially for smaller blur amounts) and reduce the number of necessary layers. On the other hand they would introduce undefined depth values, which complicates both depth peeling and N-buffers.

Since only a single fragment shader can be active at a time, in order to support multiple materials those shades had to be automatically equipped with the depth-peeling tests at the beginning. The `MaterialShaderManager` class is responsible for this.

4.1.4 Image-based ray tracing

Given the image layers produced by depth peeling and camera parameters the IBRT renderer produces and displays an output image with depth of field. It consists of the `ImageBasedRayTracer` class, which coordinates the rendering and does some precomputation of samples, and a fragment shader `IbrtFS` with the actual IBRT code.

Just as in depth peeling a single full-screen quad is rendered with both matrices set to identity. Thus the screen is parametrized by texture coordinates $[0; 1]^2$, just like the sensor in the camera model. The IBRT computation for a single pixel comprises of several steps:

1. the parametric pixel position is transformed to a position on the sensor in camera space
2. for each lens sample:
    - a sample position L on the exit pupil in camera space is computed
    - a sample position within the pixel area P is computed

- a ray from the sensor to the lens P to L is transformed via the lens model to the ray going from the lens to the scene L' to $P*$
- the intersection of this lens ray with the height field layers is searched for
- the color obtained from each lens ray is accumulated

3. the average color from all lens rays is output

For synthesizing new views on scene the most important parts are ray generation and height field intersection.

In this camera model we support tilt-shift sensors with tilt around the sensor center about the X then Y axis. The transformation from the parametric screen quad coordinates to the sensor space can be easily split into transforming the texture coordinates into a sensor in the canonical position and orientation followed by a rotation and then a translation. Such a rigid-body transformation can be precomputed into a 3×3 matrix.

For transforming the rays from the sensor to the rays to the scene the thin lens model is currently applied. The ray tracing method is general enough to support more complex lens models. It has been shown [47] that eg. a biconvex lens represented by two height fields can be evaluated interactively.

The main advantage of IBRT over multi-view accumulation is in the ability to sample each pixel with a different set of pupil samples. Repeating the same samples in each pixel leads to highly visible artifacts in area of high blur (many copies of the original image) which can be reduced only with a high number of samples. Different samples produce high-frequency noise which is more tolerable by a human viewer. One technologically challenging thing was in finding a trade-off between providing the lens model with sufficiently quality exit pupil samples and not killing the performance.

Compared to CPU code the shader code has no access to a pseudo-random generator. It is non-trivial to create one and if such it could take much of the performance needed for ray tracing. We must not forgot than jittering and concentric sampling would have to be also done for each sample in the shader.

The opposite extreme would be to precompute all the jittered lens sample sets for each pixel in a big 3D texture. Unfortunately this would be very memory intensive. For a 1024^2 pixel image with 64 samples represented as half-float pairs this would result in a texture of size $1024^2 \cdot 64 \cdot 2 \cdot 2 = 256$ MB, much larger than all the layered depth images together.

In our implementation we decided to reduce the second approach to tiles of constant size, eg. 64^2 px. The samples are precomputed once and then reused for all the rendered frames. Since they are already jittered and mapped to a unit disk all the shader needs to do is only to read them from the texture rescale them to the pupil size.

For the case of incremental rendering only a part of the 3D texture with all the samples is accesses in each iteration.

Probably a more memory-saving approach would be to construct sample pairs combinatorially just like in [46]. The goal of this implementation was simpler code than tuned performance, however.

Height field intersection

Having computed a ray from the lens towards the scene we can find its intersection with the height field layers. As the height fields are clipped to a frustum the intersection can happen only between the near and far planes. The ray can be thus clipped to those extents by intersecting it with the planes. Also the intersection algorithms work in the frustum space, ie. a unit cube, so the ray must transformed as well. Multiplying with the perspective matrix is not necessary in this special case of point on the near or far plane.

We have implemented several algorithms for height field intersection and chosen a kind of per-pixel traversal. The details are presented in the next section. A simple acceleration via N-buffers has been implemented to provide interactive performance. Each lens ray is clipped separately in one or more iterations. Better would be to take all lens rays for a pixel, compute a bounding box of their footprints and clip them to the min/max depth extents with a constant number of N-buffer queries as in [47].

4.2 Height field intersection

One of the most important routines for DoF rendering methods based on image layers is computing the intersection of a ray with a height field. It is necessary for image-based backward ray tracing or forward light tracing and also can be useful to enhance spreading with testing visibility in via forward light tracing. This algorithm is crucial to be implemented correctly and efficiently since it is then evaluated heavily.

During the project development we considered several algorithms before finding the one most suitable for the purposes of the implemented DoF rendering method. One brand new algorithm was proposed, although being finally replaced by a modified version of a simpler existing algorithm. Still, it exhibits some quite nice features so it is described here. Also some ideas from this algorithm are reused in the modification of the final algorithm. In summary the following algorithms were considered:

1. [46] – modified method of false position, suitable for multiple depth interval layers with discontinuities only at the borders with undefined areas. It assumes that all pixels in each layer have bounded depth and depth intervals are disjoint.
2. [56] – linear search followed by binary search, parallel test for being inside/outside an object. It assumes that the rendered objects were closed, eg. planar discontinuous objects are not supported well.
3. [3] – robust binary search, very expensive precomputation, single-layer.
4. [3] (naive) – 2D version of the voxel traversal [2], single-layer.
5. [47] – they do not provide any details, only mention a naive per-pixel traversal, parallel multi-layer intersection test and focus rather on acceleration. They probably implicitly refer to the naive method in [3].

An algorithm suitable for intersection with height fields used in our chosen IBRT method must comply with the following assumptions:

- there can be one or more depth layers

- due to depth peeling values are ordered by layers ascendantly only per pixel, not per whole image
- there might be steep discontinuities at the borders between objects in different depths but in the same visibility layer
- there might be non-closed objects
- when using N-buffers for acceleration there might be areas with some special "undefined" value where there can be no intersection
- if a pixel in some layer contains the undefined value all the subsequent layers are also undefined
- the depth values are in the depth buffer space [0.0; 1.0]
- the rays will go from front to back (depth 0.0 to 1.0) and will be nearly perpendicular to the XY plane

All the mentioned algorithms (except [47] which provided no information on this) treat the each height field layer as a function obtained from the array of point samples by bilinear interpolation. This in fact prevents seeing beyond the first layer by discontinuities, effectively resulting in the further layers being useless.

Also methods based on stepping in the ray parameter space rather then traversing its rasterized footprint have problems with those discontinuities. Too spare steps result in missed intersection, while too dense steps only waste texture reads. Rasterized traversal provides an optimal number of intersection tests at the cost of linear complexity to the footprint length. In contrast linear search has constant complexity and binary search logarithmic one. Since the lens rays can be clipped using N-buffers [47] to get a small footprint, rasterized traversal is more suitable.

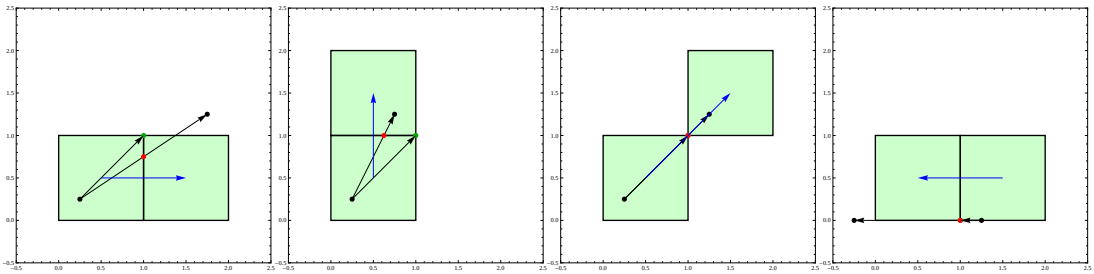
The other problem is with intersection tests themselves. The algorithms working with a single layer can simply consider the layer to be varying linearly and find the point where the ray goes beyond the surface. Also multi-layer relief mapping assumes closed objects, thus it is possible to test whether the ray is currently inside or outside and find the first surface where it goes from outside to inside an object. On the other hand continuing to visit other layers when the ray goes through a discontinuity requires different tests.

Due to the discontinuity problems we restricted the algorithm to be used to treat the height field by nearest neighbor interpolation, ie. consider each pixel as a tiny square with constant depth.

First we developed a custom robust algorithm for rasterized ray footprint traversal equipped with a test for intersection with multiple layers. Then after studying the voxel traversal algorithm [2] we decided due to its simplicity to use its 2D variant for the footprint traversal. However it had to be modified for more robustness. The intersection test was also modified for being better suitable for data-parallel usage with in shaders. Both algorithms do not assume the height field layers to be continuous nor objects to be closed.

4.2.1 Custom rasterized intersection algorithm

We assume the scene is represented by one or more layers of rasterized color and depth images. Also we assume the layers were extracted via depth peeling [25], not in depth intervals [4]. This means that for each pixel all the values are defined from the nearest layer up to a certain limit. Undefined values might be only beyond this limit which contrasts with depth interval extraction where there might be undefined



(a) Right (b) Up (c) Across corner (d) Axis-aligned ray direction

Figure 4.1: Ray footprint traversal - selecting next pixel

gaps in arbitrary layers within a single pixel. The results of umbra or extended-umbra depth peeling [47] are acceptable. In case of depth-interval extracted layer the algorithm would have to be modified to skip undefined values for a particular pixel.

For the purposes of intersecting a height field with a ray we interpret each pixel in the height field as a square (or possibly a rectangle) element. We assume depth discontinuities produced by the nature of depth peeling, so bilinear interpolation of the surface is not a correct interpretation. At a discontinuity it would create a false vertical surface which would prevent seeing further layers.

The height field data originate from rasterizing the objects within a frustum transformed from world space into an NDC cube. It means that a pixel on the far plane corresponds to much larger region in the world space than a pixel on the near plane. This also means that rays to be intersected must be transformed from the world space by the frustum transform and clipped to the NDC cube depth extent.

One of the approaches to intersecting a height field is in projecting the ray onto the 2D plane of the depth image. The general strategy is then in traversing the ray projection and testing whether the ray intersected a particular pixel square. On the first such an intersection the traversal ends and the position is reported.

Rasterized ray footprint traversal algorithm

In order not to miss any ray-square intersection the rasterized footprint of the ray projection is traversed. Unfortunately common line rasterization algorithms are suitable for drawing visually pleasing lines, not for computing all pixels under a ray. For this purpose a custom algorithm have been created. Also note that for testing height field intersections we need to know not only the pixels under the ray but also intersections of the ray with edges of those pixels.

The outline of the algorithm is as follows. We start from the pixel under the starting point R_s of the ray R. In each step, being in a particular pixel P, we decide to which pixel in the 8-neighborhood should the traversal continue. The idea is to select the corner C of the pixel P approximately in the direction of the ray and decide upon the orientation of the angle $\angle PCR_e$ where R_e is the end point of the ray.

Eg. if the ray direction aims within the top-right quadrant the potential target pixels are upwards, rightwards and across the top-right corner in between. Let us call such a corner the *nearest corner*. The orientation of the angle $\angle PCR_e$ can be

computed from the cross-product $(P-C)\times(R_e-C)$ (which is a vector perpendicular to both). As both $(P-C)$ and (R_e-C) are 2D vectors the cross-product of their 3D extensions – $(P_x, P_y, 0)$ etc. – has the x and y components zero, thus its z component alone gives the orientation: $(A \times B)_z = A_x B_y - A_y B_x$. If it is positive the traversal should continue to the clockwise edge, if it is negative to the counter-clockwise edge and a zero value indicates the ray directly crosses the corner (see figure 4.1).

The traversal starts at the pixel under the ray starting point R_e, where the pixel corresponding to a point P is $\lfloor P \rfloor = (\lfloor P_x \rfloor, \lfloor P_y \rfloor)$ (thus $\lceil P \rceil$ belongs to another pixel). In case the starting point lies on a pixel edge and the ray points outside the starting pixel, it needs to be shifted such that the ray points inside. The end pixel is found the same way, except the ray direction has to be reversed.

$$\begin{aligned} \text{pixel}(P, D) &= \lfloor P \rfloor + (\text{s}(P_x, D_x), \text{s}(P_y, D_y)), \quad \text{where} \\ \text{s}(p, d) &= \begin{cases} -1 & \text{if } (\text{sgn}(d) < 0) \text{ and } (p = \lfloor p \rfloor) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

The nearest corner C for the starting pixel $S = \text{pixel}(R_e, D)$ depends on the orientation of the ray direction. Also care must be taken when the direction is axis-aligned, ie. $\text{sgn}(D_x)\,\text{sgn}(D_y) = 0$.

$$\text{nearestCorner}(S, D) = \begin{cases} S + 0.5(\text{sgn}(D) + (1,1)) & \text{if } \text{sgn}(D_x)\,\text{sgn}(D_y) \neq 0 \\ S + \text{sgn}(D) & \text{otherwise} \end{cases}$$

For each visited pixel we will recognize an *entry point* and *exit point*. An entry point is either the starting point of the ray or the intersection of the ray with the pixel where the ray enters it. Similarly an exit point is either the the intersection of the ray with the pixel where the ray exits it or the end point of the ray. Note that the exit point of one pixel is the same as the entry point to the next pixel.

After initialization the traversal proceeds in a loop until an intersection is found or the end pixel has been reached. In each step the direction to the next pixel is found, the exit point is computed, intersection is tested and the traversal proceeds to the next pixel. Finally the tail of the ray footprint (at the end pixel) is tested for intersection.

Computing the direction to the next pixel is different for a general ray direction and axis aligned direction. In the latter case this equals to $\text{sgn}(D)$. In the general case is depends of the orientation of the aforementioned cross product $(P - C) \times (R_e - C)$. In summary:

$$\begin{aligned} \text{nextDir}(P, C, R_e, D) &= \begin{cases} (\text{sgn}(D) + n)/2 & \text{if } a > 0 \\ (\text{sgn}(D) - n)/2 & \text{if } a < 0 \\ \text{sgn}(D) & \text{if } a = 0 \text{ or } \text{sgn}(D_x)\,\text{sgn}(D_y) = 0 \end{cases} \\ \text{where} \quad a &= ((P - C) \times (R_e - C))_z \\ n &= (-\text{sgn}(D_y), \text{sgn}(D_x)) \end{aligned}$$

Then the exit point have to be computed as the intersection of the ray with the edge leading to the next pixel or as the corner in case of going right across the

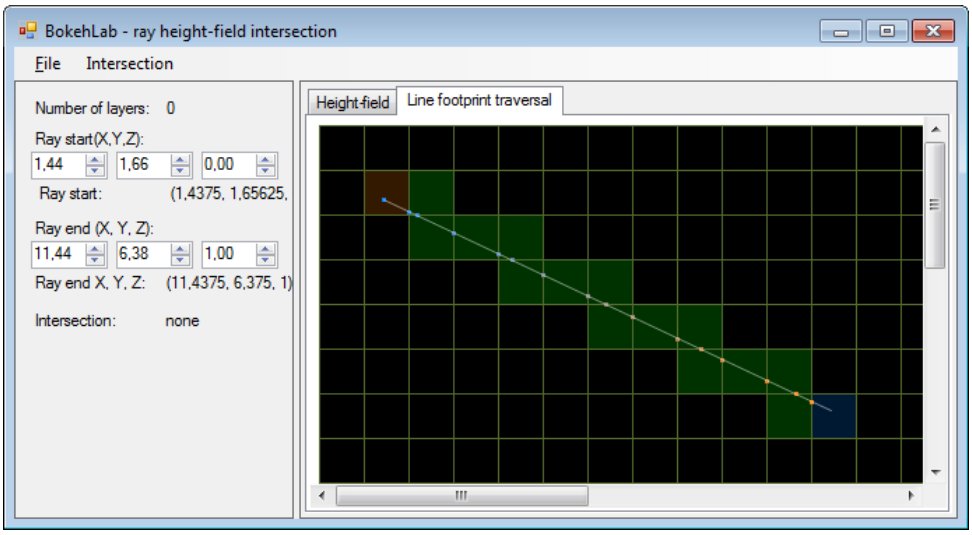


Figure 4.2: Ray footprint traversal - an interactive visualization

corner. N is the direction to the next pixel. Note that if $N = (0, 0)$ the main loop would have stop immediately without ever getting here.

$$\text{exitPoint}(P, C, D, N) = \begin{cases} P - D(P_y - C_y)/D_y & \text{if } N_x = 0 \text{ and } N_y \neq 0 \\ P - D(P_x - C_x)/D_x & \text{if } N_x \neq 0 \text{ and } N_y = 0 \\ C & \text{if } N_x \neq 0 \text{ and } N_y \neq 0 \end{cases}$$

The z coordinate of the exit point X in the original ray (which is needed for intersection testing) is then computed from the ray's parametric equation as:

$$X_z = \begin{cases} R_{s_z} + (X_x - R_{s_x})D_z/D_x & \text{if } |D_x| > |D_y| \\ R_{s_z} + (X_y - R_{s_y})D_z/D_y & \text{otherwise} \end{cases}$$

An interactive visualization of the ray footprint traversal through the pixel grid is available (see fig. 4.2) in the `BokehLab.Demo.HeightField` project.

Intersection testing

The result of testing the whole ray is either an intersection (where the pixel position and depth is reported) or no intersection. The situation in each visited pixel is as follows. The 2D ray projection enters the pixel in some point N and exits it in point X. For the ray start and end point the N and X point might not lie on a pixel edge. A height field depth image at that pixel provides a square at depth H_z. The original 3D ray intersects the square if and only if:

$$\text{sgn}(N_z - H_z) \neq \text{sgn}(X_z - H_z),$$

ie. the depth of the square is in between depths of the entry and exit points.

When there are multiple height field depth layers at that pixel are multiple squares of increasing depth $H[i]_z$ and each of them is tested in sequence (until an intersection is found). Since there is per-pixel depth ordering early termination is possible. In practice care has to be taken to ignore missing depth data and also terminate the pixel testing. Assuming the missing data are marked with depth 1.0 in case this depth is taken from the depth image we can safely terminate in intersection

testing for that particular visited pixel since all further layers have also value 1.0. When also assuming the ray direction has always positive z component D_z it is safe to early terminate even when $H[i]_z > X_z$ since all further height field values will be greater (conversely $H[i]_z < X_z$ for negative D_z. The multi-layer algorithm with early termination looks like this (L denotes the number of layers):

Algorithm 2 Multi-layer height field intersection:

for $i = 0 \rightarrow L - 1$ **do**
  if $(H[i]_z = 1)$ or $((H[i]_z > X_z)$ xor $(D_z < 0))$ **then**
    return no intersection
  end if
  if $\text{sgn}(N_z - H[i]_z) \neq \text{sgn}(X_z - H[i]_z)$ **then**
    return intersection
  end if
  $i \leftarrow i + 1$
end for
return no intersection

4.2.2 Simpler rasterized intersection algorithm

A simplified algorithm which is at last used in the IBRT shader was created by replacing the ray footprint traversal with 2D variant of the voxel traversal algorithm [2], modifying it for more robustness and equipping it with a parallel multi-layer intersection test.

The main idea of the algorithm is the following. A 2D projection of a 3D ray on a pixel grid intersects the grid at pixel edges. Each intersection leads the ray from one pixel to another. The ray intersects with vertical or horizontal edges. In case of a corner intersection both edges are intersected at once. The idea is to choose in each step an intersection with either edge, whichever is closer.

The algorithm precomputes some constants and then visits pixels in a loop until finding the end pixel. t_{max} maintains the ray parameters of the next intersection with a pixel edge in either axis, while t_δ is a constant parameter step for intersections in each axis. In the loop whichever component of t_{max} is smaller it gives the direction to the next visited pixel.

We maintain the discrete current pixel and end pixel positions, in the original algorithm computed as a plain floor of the continuous start and end positions. In case the ray starts at its boundary edge and the goes in the negative direction some pixels near the start might be visited needlessly, but worse the footprint traversal might miss the end pixel leading to an infinite loop! Thus it is necessary to shift the start and end discrete pixels the same way as in the previous algorithm's `pixel` function.

Another modification is to avoid singularities (division by zero etc.) when the ray direction is axis-aligned, ie. either component is zero. Is such a case we shift it by a small ε. The ray end must be corrected accordingly.

Apart from plain discrete positions we need to know the exact position of ray-edge intersections. This is equivalent to the notion of entry and exit points in the previous algorithm. The intersection with the current pixel can be done after

updating the exit point. An exit point of one pixel becomes the entry point of the following one. Also note that the loop stops just before visiting the last pixel, which must be then tested separately.

It remains to explain the intersection test done at each visited pixel. Treating the height field with nearest-neighbor interpolation reduces the single-layer intersection problem to testing intersection of a square with a segment from the entry point to the exit point. In other words if and only if the entry point is in front of the square and the exit point behind there is an intersection. We can repeat this test for several layers and get the color from the first intersected one.

Note that the height field is sampled at pixel centers is it necessary to get the layer depth also at a pixel center. There is one catch. The cost for the lack of interpolation in the height field is that any non-constant surface is discontinuous at pixel edges. For rays aligned with the z axis it leads to artifacts – small view-dependent holes within surfaces.

A simple workaround is to move the exit point an epsilon further. Experimentally was found that value around 0.01 works fine. For a single layer the modified test could look like this:

```
bool intersects = (entry.z < layerZ) && (layerZ <= exit.z + epsilon);
```

Since the input depth images are packed by four channels per texture we have to compare up to 4 layers at once. Instead of comparison operators we can indicate the intersection via the signum function. Intersection happens at the first layer with both signs positive, ie. with the indicator value equal to 2. From such a layer the color can be grabbed.

The GLSL code if this algorithm is provided in the section A.3 of the first appendix.

4.3 Sequential lens ray tracer

The sequential ray tracing of geometric lens models [41] has to be implemented in a C# library. It is used from several applications.

First, a prototype of image-based ray tracer has been created to test the various lens models, including thin lenses and pinholes. It is equipped with a simplified scene model, just a single textured rectangle. Even without intersecting height fields the effect of optical aberrations, vignetting and swirly bokeh can be seen. The sensor can be tilted and shifted. Jittered sampling of the back aperture with concentric mapping is used. It could be improved by sampling the exit pupil or the approximate effective pupil, however.

Second, a visualization of the ray tracing inside a complex lens system had been implemented. Finally, it is utilized in LRTF precomputation.

4.3.1 Lens ray transfer function

A rough prototype of computing the LRTF with the basic parametrization was implemented. It is possible to convert between rays and their parametrization, then a full 3D table can be precomputed and even utilized for image-based ray tracing.

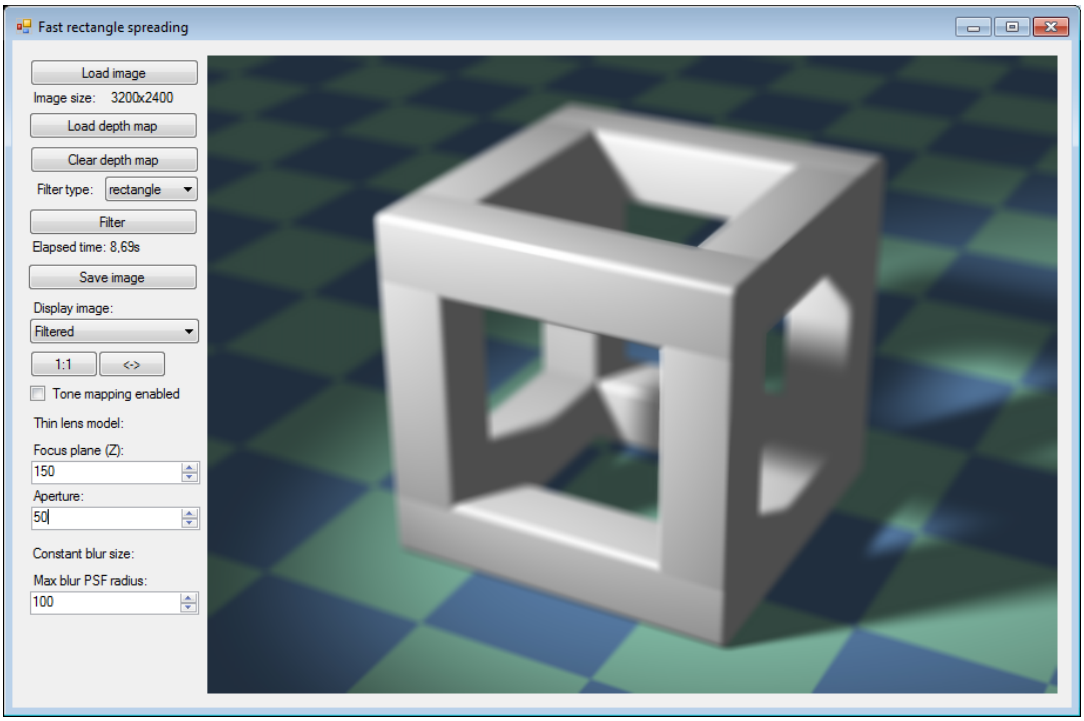


Figure 4.3: A demo application for fast spreading filters

4.4 Fast spreading filters

Originally we liked also to compare the abilities of image-based ray tracing to fast spreading filters. Later we decided to stick with IBRT for several reasons: IBRT is more extensible to generalizations to more complex lens models, such as tilt-shift configurations, and also its GPU implementation required only OpenGL, not also CUDA or OpenCL. For those reasons only an isolated CPU prototype of fast spreading filters is available, in the `BokehLab.Spreading` project.

Three kinds of fast spreading filters are implemented: rectangle spreading, perimeter spreading and hybrid of both controlled by a contrast criterion. Circular PSFs are generated to be used with perimeter spreading. The PSF size can be constant or controlled by the thin lens model backed by a depth map.

A GUI demo (see fig. 4.3) was created using the library which allows the user to save/load images to be spread and control the lens model parameters. Only a single layer is supported. A basic experimental integration with the output from OpenGL rasterization was made, outside the mentioned interactive demo.

4.5 Portable float map library

There arose a need for representing HDR images (with floating point pixel format) in C# code and their storage in files, mainly to support image-based ray tracing and spreading filters. A custom float-map image library, `BokehLab.FloatMap`, was created. Since some example HDR images are distributed in the simple Portable Float Map [22] file format this format was selected for storage of our float-maps. Some image processing routines were also created, along with an integration with LDR images (including a trivial tone-mapping). The PFM format was extended to

support the alpha channel for transparency in layers.

4.6 Additional models and visualizations

Apart from the main renderers a lot of auxiliary mathematical models and interactive visualizations were implemented during the course of the project creation in order to better understand the theory, to get the formulas right and to provide nice illustrations for the thesis. Since they are useful themselves we will describe some of them too. Some models are implemented in Mathematica, some in C#.

Transformation matrices

Transformation matrices such as perspective, thin lens, thick lens, frustum has been wrapped up with procedures for conversion from/to homogeneous coordinates. Thus the correct form of the matrices with respect to all local conventions could be found. Also they can be readily used in subsequent models and visualizations.

Thin lens transformation with ray fans

The classical model of thin lens with a fan of rays going through various points on the aperture was interactively visualized (fig. 2.6). The parameters as the position of the source object, focal length, aperture radius can be modified by the user. Thin lens and perspective transformations can be switched.

Entrance and exit pupils for thin lens

Computing the entrance and exit pupils has been implemented for thin lens model with a possibility to set the aperture stop on an arbitrary plane, not just the lens plane (fig. 2.11b). In case the aperture stop is behind the lens the entrance pupil its image, else the aperture stop itself is the entrance pupil. Conversely for the exit pupil. Also a viewpoint can be set with rays from it to the pupil visualized. It can be clearly seen that if the viewer in front of the lens looks at the entrance pupil the ray gets refracted and hits the corresponding point on the aperture stop.

Thin/thick lens with tilt-shift sensor

There is a interactive model of a thick lens with a sensor which can be arbitrarily tilted and shifted. The sensor and focal planes are shown. A single center of projection can be set (a lens sample) which defines the frusta for multi-view accumulation. There is a more detailed model in 2D (fig. 2.12a) and a simpler model even in 3D (fig. 2.12b).

CoC size functions

An interactive plot of circle of confusion size depending on various parameters is available. It can be clearly seen as the signed CoC radius is of hyperbolic shape and where is the asymptote for an object at infinity (see fig. 2.8).

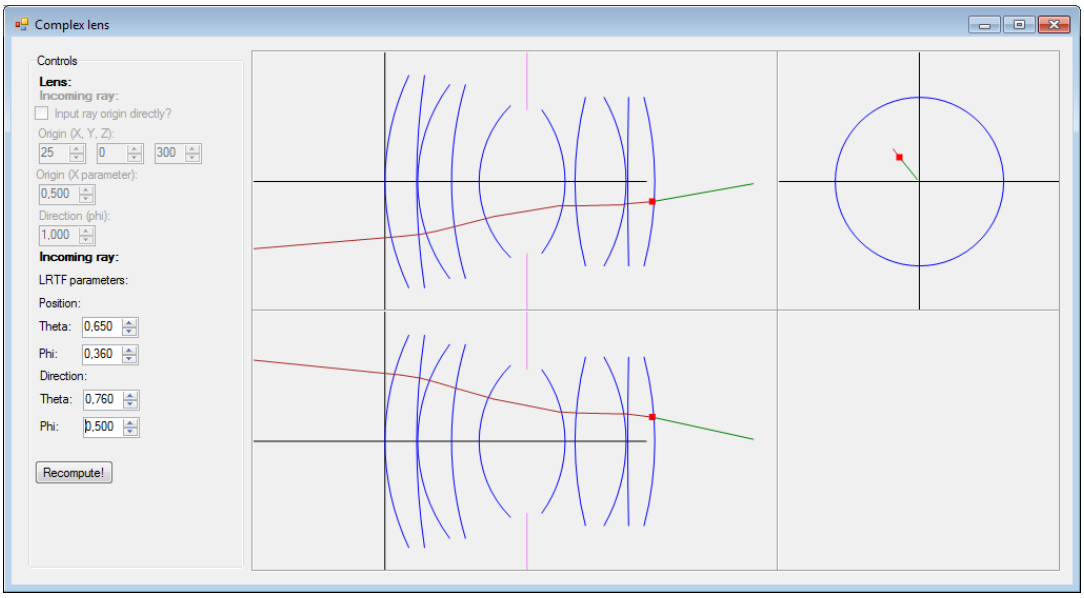


Figure 4.4: Interactive visualization of sequential ray tracing inside a complex lens system (Double Gauss lens on this image).

Limits of DoF and relation to CoC formation

More interesting than plain CoC size plots are the limits of depth of field both in object half-space and image half-space (see fig. 2.9). A diagram show how the limiting CoCs are formed on the sensor. It is widely known that DoF limits are asymmetrical in object half-space. They are also asymmetrical in image half-space. Moreover the corresponding limiting planes in object half-space and image half-space are images of each other via the lens transformation.

Ray propagation in a geometric lens system

During the implementation of the sequential ray tracing in the complex geometric lens system model [41] a nice visualization of the ray propagation inside the lens was created (fig. 4.4). It is available in the `BokehLab.Demo.ComplexLensTracing2d` project. One can select a ray incoming from the back side of the lens by setting a parametric position of the ray intersection on the back element and ray direction. The traversal is then shown from three orthographic views. Also the lens element surfaces are displayed according to their tabular specification.

Chapter 5

Results

5.1 Image-based ray tracing

We compared both DoF rendering methods used in the GPU implementation extended with the incremental rendering. Image-based ray tracing was compared to the reference method, multi-view accumulation, in several aspects.

We measured the total accumulation time, including time spent on displaying the intermediate results, depending on the scene complexity. The scene consisted mainly of a variable number of complex models (dragon, teapot) with high triangle count. An image of the full scene is shown on the fig. 5.3c.

The program was measured on two different GPUs (although both have 16 stream processors). The results are shown in the fig. 5.3. Both plots show that the total accumulation time in IBRT almost does not depend on the scene complexity, while in MVA it depends linearly. This means that IBRT is more suitable for larger scenes. The reason is simple, IBRT needs to rasterize the scene only for a constant number of layers. In contrast in MVA the scene is rasterized again for each lens sample.

We can also very roughly compare the CPU and GPU IBRT implementation. The GPU one works with the full scene and four layers, on the other hand it has precomputed tiled lens samples. Neither implementation is low-level optimized. Both IBRT and MVA give around one second for 64 lens samples. The IBRT on a CPU with the thin lens model gives 11.4 s (fig. 5.4a). This shows that the GPU can be successfully employed for acceleration of this kind of rendering algorithms.

The interactive IBRT implementation is capable of rendering a tilted focus plane and conic-section bokeh (see fig. 5.1). Both methods are capable of rendering the effect of partial occlusion and bokeh clipping (fig. 5.2).

We also compared the artifacts caused by an insufficient number of lens samples in both methods. We can see that IBRT with per-pixel sampling (fig. 5.2d) converges faster to the solution than MVA with per-image sampling (fig. 5.2c). Clearly for a human viewer high-frequency noise is more pleasant than many similar copies of the same image. Also from the temporal point of view convergence in IBRT is visually smoother.

5.2 Sequential lens ray tracing

We tested the sequential lens ray tracing method implemented on a CPU within a very simple IBRT framework. The scene is only a single planar image. The complex

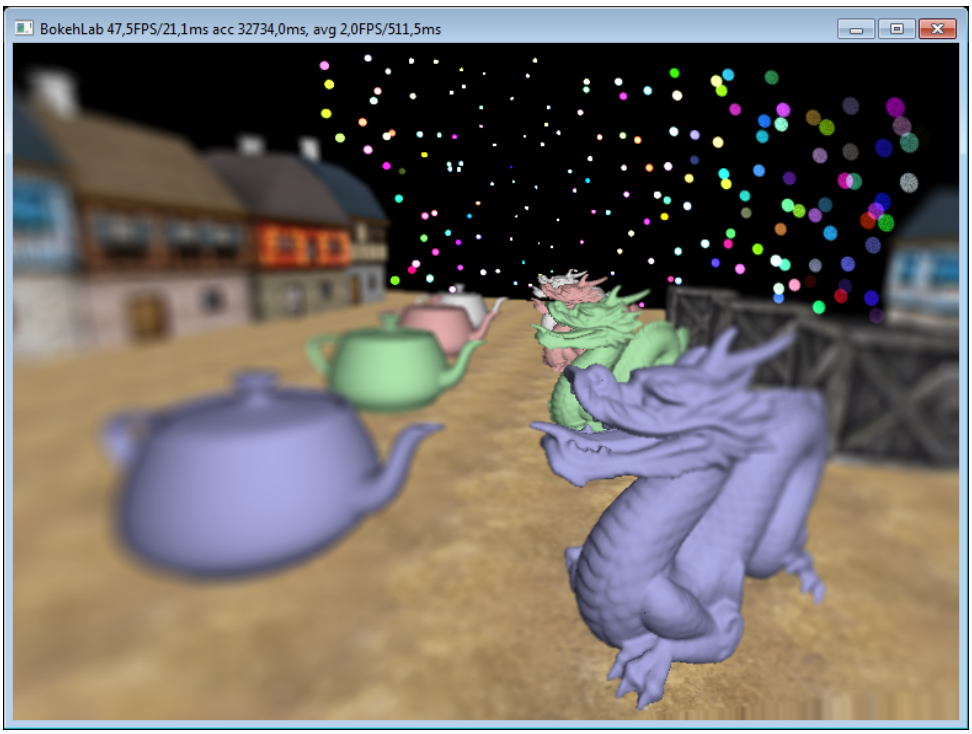
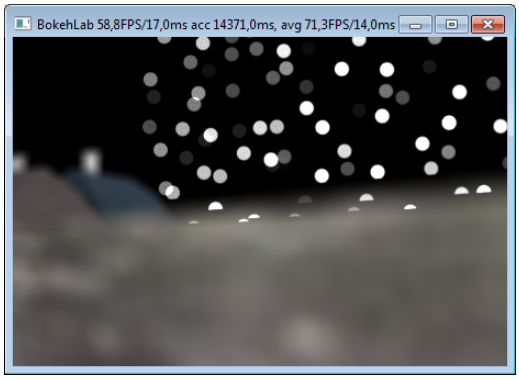
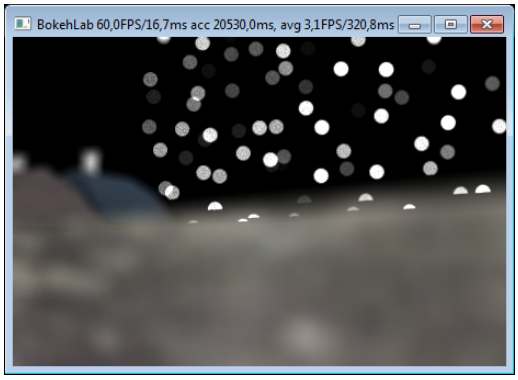


Figure 5.1: Example result of incremental IBRT with thin lens and tilt-shift sensor, 1024 lens samples.



(a) MVA 1024 samples

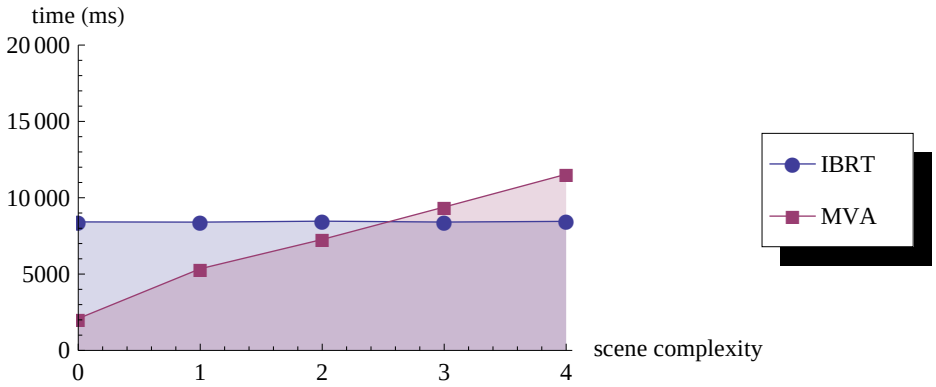


(b) IBRT 1024 samples

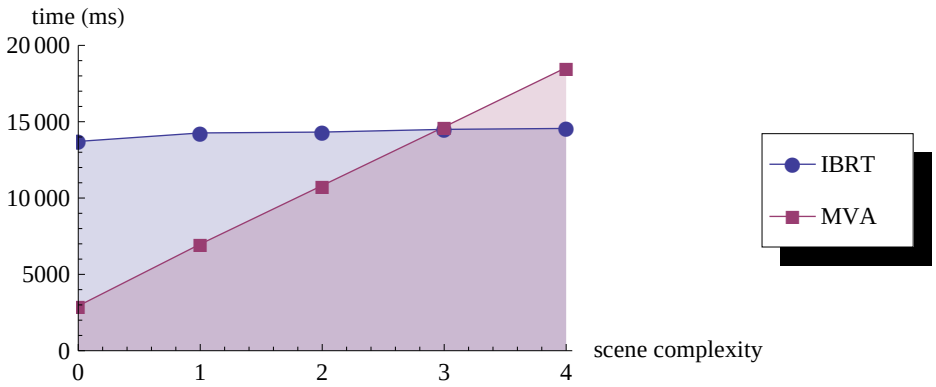
(c) MVA 32 samples

(d) IBRT 32 samples

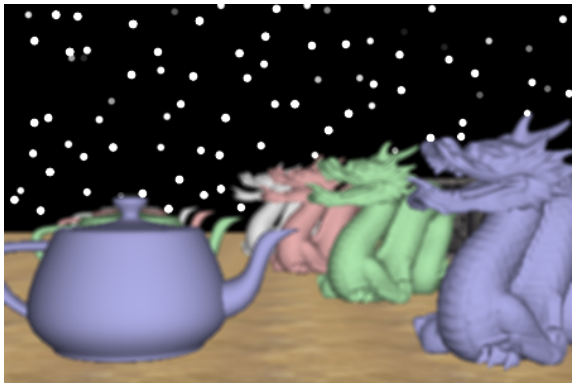
Figure 5.2: Partial occlusion in incremental image-based ray tracing (IBRT) and multi-view accumulation (MVA) at 450×300 px resolution.



(a) NVIDIA GeForce 9400GT



(b) NVIDIA Quadro NVS 140M



(c) The test scene with full complexity

Figure 5.3: Measurements of incremental image-based ray tracing (IBRT) and multi-view accumulation (MVA) both with an FBO accumulator and 1024 lens samples at 450×300 px resolution, averaged from about five runs each. The scene differs in the number of complex models (dragon, teapot) from 0 to 4. The triangle count for a single dragon is 202520, and 6400 for a teapot, other models are much smaller.

lens model was compared to the thin lens model and the pinhole model. For complex lenses the whole back surfaces were sampled. Details on rendering parameters, the resulting images and times are shown on the fig. 5.4. In addition rendering with the pinhole model took 8.2 s.

On the fig. 5.4e it shown a plot of rendering time depending on the number of lens element surfaces. In addition to complex lenses with non-zero surfaces there is pinhole model with the baseline time. The red points represent the actual measurements, while the blue line is a linear fit of the first two lenses (pinhole and biconvex) which are without vignetting. In the rest the rays can be absorbed which terminates the traversal earlier. Thus the rendering time here are below than the linear extrapolation. On the other hand the visible noise is stronger. A better sampling (such as the effective pupil sampling [68]) could help in this situation.

5.3 Lens ray transfer function

The basic parametrization of the LRTF (described in section 3.3.1) has been implemented to represent complex geometric lenses and to work in the lens ray generation stage of the IBRT. The function has been sampled into a 3D table by ray tracing the original lens in a precomputation step. Then in the IBRT phase the values of the function were looked up from the table and tri-linearly interpolated. In addition some coordinate transformations are needed due to the parametrization.

The goal here was only to show whether it is in principle capable of rendering images without the need for sequential lens ray tracing. Some artifact from the non-ideal parametrization were expected. Besides comparing the output of LRTF-rendered images to the reference one (done by full lens ray tracing) we wanted to compare the results from multiple LRTF tables with various sample density.

The resulting images are shown in the figure 5.5. The first (fig. 5.5a) is the reference image rendered with full sequential ray tracing of a Double Gauss lens. The rest are rendered by evaluating the LRTF tables with decreasing resolution.

All render times are similar (120 sec for the reference, 150 sec for the LRTF images). This CPU implementation is by no means optimized and exactly the texture lookup and interpolation operations are expected to be optimized heavily on GPUs.

We can see in the resulting images that LRTF tables with resolution 64^2 to 256^2 (fig. 5.5d, 5.5c, 5.5b) are capable of reproducing the image which is the evidence that the approach is viable! Also all the LRTF images exhibit an artifact in the view direction (the center of the image). It is caused by the non-ideal parametrization, more precisely using the discretized spherical coordinates for representing direction of a ray suffers from a singularity at the pole of the sphere which is undersampled. Numerical errors arising from this lead to differently directed rays which results in noise and blur. It is clearly visible in the images produces with the lower-resolution LRTF tables (fig. 5.5f, 5.5e).

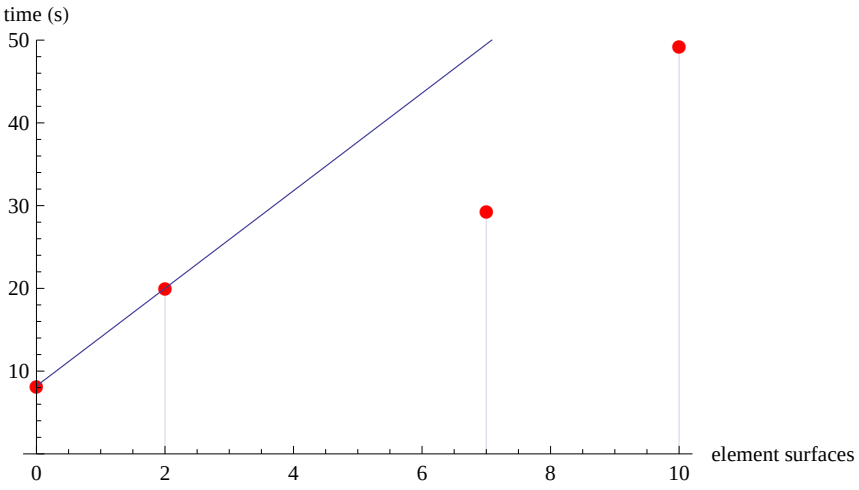
In summary, the resulting images show what was expected. The precomputation approach should be further examined, better parametrization defined and a GPU implementation should be made and compared to the other methods.

(a) Thin lens, 11.4 s

(b) Biconvex, 20.0 s, 2 surfaces

(c) Petzval, 29.4 s, 7 surfaces

(d) Double Gauss, 49.3 s, 10 surfaces



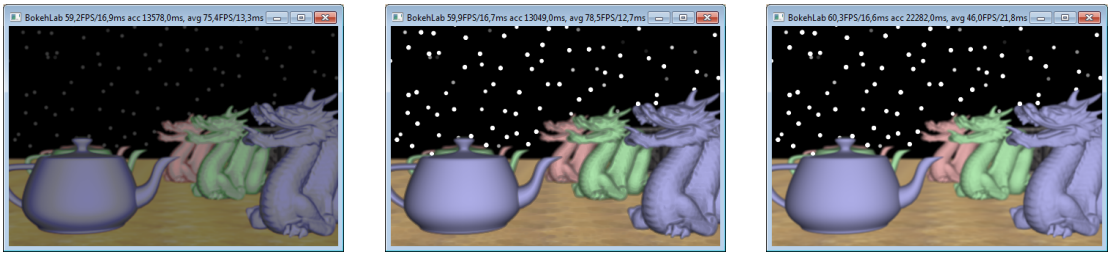
(e) Rendering time vs. lens complexity

Figure 5.4: Image-based ray tracing done on a CPU with a single planar image as the scene, sequential lens ray tracing, 64 lens samples, image size 450 × 300 px, tone-mapped. The thin lens model (fig. 5.4a) acts as a reference (it is implemented via a matrix transformation). The sensor was tilted and shifted variously in the examples to show the visual effects which does not affect the rendering times.

(a) Reference (b) LRTF 256^2 (c) LRTF 128^2

(d) LRTF 64^2 (e) LRTF 32^2 (f) LRTF 8^2

Figure 5.5: Image-based ray tracing done on a CPU with a single planar image as the scene, image size 450×300 px, Double Gauss lens, 64 lens samples.



(a) Accumulation buffer (b) FBO 16-bit float (c) FBO 32-bit float

Figure 5.6: The comparison of accumulation in accumulation buffer and with framebuffer objects. Multi-view accumulation, 1024 samples.

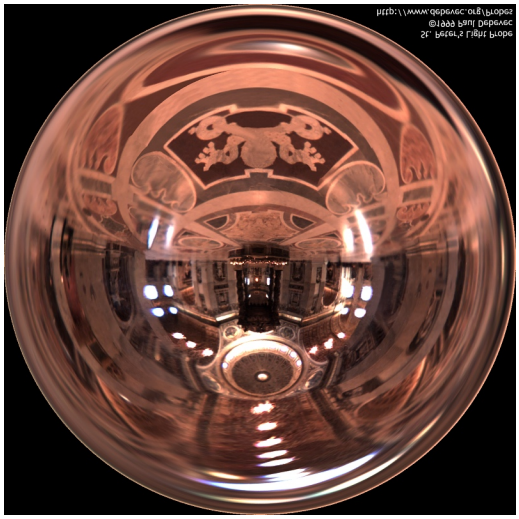
5.4 Accumulation buffer vs. FBO

In the fig. 5.6 we can see the results of multi-view accumulation of a high number of samples into buffers of different numeric precision. We can see that the accumulation buffer with (probably 12-bit) integer format (fig. 5.6a) is entirely not suitable for such an accumulation and HDR rendering (for bokeh) in particular. The 16-bit floating-point buffer (fig. 5.6b) is better but still exhibits some darkening of the image. Finally the 32-bit buffer (fig. 5.6c) shows no artifacts.

5.5 Fast spreading filters

We tested the prototype of the fast spreading filters applied on some HDR and LDR images. There are three filters compared – rectangle spreading, perimeter spreading with a circular PSF and a hybrid of both.

One set of images (fig. 5.7) shows the usage of spreading filters on a large HDR image to produce bokeh. As we can see the, even not optimized, constant-time rectangle spreading is extremely fast (when compared to the CPU-based IBRT).



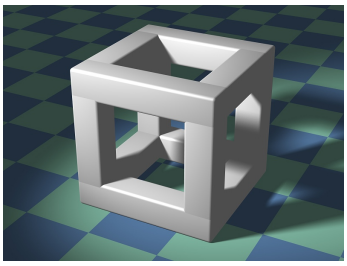
(a) Original

(b) Rectangle spreading, 1.25s

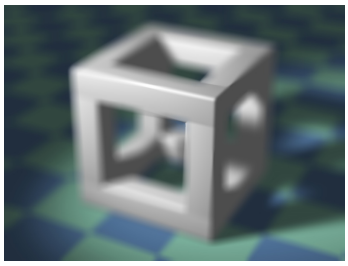
(c) Perimeter spreading, 103.93 s

(d) Hybrid spreading, 9.81 s

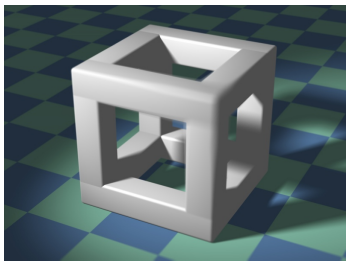
Figure 5.7: Fast spreading filters with constant-size PSFs with 100 px radius applied on an HDR light probe from St. Peter's dome at 1500^2 px resolution. The results are tone-mapped.



(a) Original



(b) Front focus, 19.27s



(c) Back focus, 17.06 s

Figure 5.8: The hybrid fast spreading filter with a circular PSF controlled by a depth map and the thin lens model, applied on the Cubic structure LDR image at 3000×2000 px resolution.

Even the hybrid algorithm with a circular PSF is much faster than comparable IBRT. The cost is that visibility is not solved here.

The second set of images (fig. 5.8) shows the results from spreading filter with the PSF radius controlled by the thin lens model and a depth map. Even for a huge image resolution the performance is notable. On the other hand artifacts arising from the lack of layers can be seen at depth discontinuities.

Chapter 6

Conclusion

Summary An extensive work has been done towards implementing an interactive depth-of-field preview renderer. It is based on the state-of-the-art method of image-based ray tracing, which utilizes the power of current GPUs, while offering some of the flexibility of distribution ray tracing. For comparison the reference method, the multi-view accumulation is provided. Both methods were extended to perform incremental rendering even on older hardware and without full optimization. The accumulation technique was revised to support accumulation of more samples.

The IBRT method depends on several techniques, such as depth peeling, height-field intersection, N-buffers, etc. Their applicability to the specific data needs were analyzed and they were implemented accordingly. A custom robust ray footprint traversal algorithm was proposed and an existing one was extended to act as a base for a data-parallel height-field intersection algorithm. The interactive IBRT method is capable of rendering a preview of depth of field in ordinary and tilt-shift camera configurations and is prepared for extending with further lens models.

A prototype of fast spreading filters was implemented, which supports rectangle spreading, perimeter spreading and a hybrid of both. For easier work with HDR images a custom float map library was implemented.

The sequential lens ray tracing method was explored and a non-interactive prototype was implemented along with a visualization of ray tracing inside a complex lens.

The theory was studied and summarized and an extensive bibliography on DoF rendering methods and related problems was compiled. Several interactive visualizations are available. Also the existing rendering methods were compared.

The sketch of the lens ray transfer function and its precomputation was introduced along with a basic prototype. Several points of further research were identified and suggested.

From this point of field the goals of the thesis were achieved.

Future work As the area of depth of field rendering is quite extensive not everything can be explored and implemented within a single thesis. In particular the software can be extended by the existing techniques described in literature and optimized (eg. with extended umbra depth peeling, better ray clipping via N-buffers, sequential lens ray tracing within the shader, correct radiometry, etc.).

Also there are some further directions which are worth a research. In particular, representing the behavior of complex lenses without the internal geometric design

offers many possibilities. To our knowledge the potential of future methods lies in better sampling (of the effective pupil of the lens, in particular) in ray tracing and in solving visibility more accurately in fast spreading filters, possibly by combining both approaches. Also the method in [35] should be revised for applicability on current hardware.

Acknowledgments We thank to the authors of several resources used in this project. OBJ models and textures: *dragon* – Stanford University Computer Graphics Laboratory[1], *medieval street*, *crate* – TurboSquid[2], *teapot* – Martin Newell (original model), Dr. Charalambos Poullis (OBJ file)[3], HDR light probes – Paul Debevec[4]. The Fresnel equation plots are courtesy of Dr. Alexander Wilkie. The photograph of the Kamelot band is courtesy of Jan Otruba, used with permission. The *cubic structure* image and depth map are courtesy of Dominic Alves (CC BY-SA 3.0).

[1] `http://graphics.stanford.edu/data/3Dscanrep/`
[2] `http://www.turbosquid.com/`
[3] `http://www.poullis.org/software.html`
[4] `http://ict.debevec.org/~debevec/Probes/`

Bibliography

[1] Adelson, E. H., and Bergen, J. R. The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, M. S. Landy and A. J. Movshon, Eds. MIT Press, Cambridge, MA, 1991, pp. 3–20.

[2] Amanatides, J., and Woo, A. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87* (1987), pp. 3–10.

[3] Baboud, L., and Décoret, X. Rendering geometry with relief textures. In *Graphics Interface '06* (2006).

[4] Barsky, B. A. Vision-realistic rendering: simulation of the scanned foveal image from wavefront data of human subjects. In *Proceedings of the 1st Symposium on Applied perception in graphics and visualization* (New York, NY, USA, 2004), APGV '04, ACM, pp. 73–81.

[5] Barsky, B. A., Garcia, D. D., Klein, S. A., Yu, W. M., Chen, B. P., and Dalal, S. S. Rays (render as you see): Vision-realistic rendering using hartmann-shack wavefront aberrations, 2001.

[6] Barsky, B. A., Horn, D. R., Klein, S. A., Pang, J. A., and Yu, M. Camera models and optical systems used in computer graphics: part i, object-based techniques. In *ICCSA'03: Proceedings of the 2003 international conference on Computational science and its applications* (Berlin, Heidelberg, 2003), Springer-Verlag, pp. 246–255.

[7] Barsky, B. A., Horn, D. R., Klein, S. A., Pang, J. A., and Yu, M. Camera models and optical systems used in computer graphics: part ii, image-based techniques. In *ICCSA'03: Proceedings of the 2003 international conference on Computational science and its applications* (Berlin, Heidelberg, 2003), Springer-Verlag, pp. 256–265.

[8] Barsky, B. A., and Kosloff, T. Algorithms for rendering depth of field effects in computer graphics. *Proceedings of the 12th WSEAS international conference on Computers 2008* (2008).

[9] Barsky, B. A., and Kosloff, T. Three techniques for rendering generalized depth of field effects. *Proceedings of the Fourth SIAM Conference on Mathematics for Industry: Challenges and Frontiers (MI09)* (2009), 42–48.

[10] Barsky, B. A., and Pasztor, E. Rendering skewed plane of sharp focus and associated depth of field. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches* (New York, NY, USA, 2004), ACM, p. 92.

[11] Barsky, B. A., Tobias, M. J., Chu, D. P., and Horn, D. R. Elimination of artifacts due to occlusion and discretization problems in image space blurring techniques. *Graphical Models 67* (November 2005), 584–599.

[12] Barsky, B. A., Tobias, M. J., Horn, D. R., and Chu, D. P. Investigating occlusion and discretization problems in image space blurring techniques. *International Conference on Vision, Video, and Graphics* (2003).

[13] Bavoil, L., and Myers, K. Order independent transparency with dual depth peeling.

[14] Bogomjakov, A. *GPU-assisted Geometry Processing for Novel View Synthesis from Depth Video.* PhD thesis, Technion - Israel Institute Of Technology, January 2008.

[15] Born, M., Wolf, E., and Bhatia, A. *Principles of optics: electromagnetic theory of propagation, interference and diffraction of light.* Cambridge University Press, 1999.

[16] Bourke, P. Calculating stereo pairs. July 1999.

[17] Bourke, P. Offaxis frustums - opengl. July 2007.

[18] Cohen-Or, D., Rich, E., Lerner, U., and Shenkar, V. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics 2* (September 1996), 255–265.

[19] Cook, R. L., Porter, T., and Carpenter, L. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18* (January 1984), 137–145.

[20] Coxeter, H., and Coxeter, H. *Introduction to geometry.* Wiley Classics Library. Wiley, 1989.

[21] de Greve, B. Reflections and refractions in ray tracing. November 2006.

[22] Debevec, P. Pfm image file format.

[23] Décoret, X. N-buffers for efficient depth map query. *Computer Graphics Forum 24*, 3 (2005).

[24] Demers, J. *GPU Gems.* Addison-Wesley, 2004, ch. Depth of Field: A Survey of Techniques.

[25] Everitt, C. Interactive order-independent transparency, 2001.

[26] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. *Computer graphics: principles and practice (2nd ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[27] Glassner, A. *An Introduction to ray tracing.* Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Academic Press, 1989.

[28] Glassner, A. S. *Principles of Digital Image Synthesis.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[29] Gonzalez, R. C., and Woods, R. E. *Digital Image Processing*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[30] Goodman, J. *Introduction to Fourier optics*. McGraw-Hill physical and quantum electronics series. Roberts & Co., 2005.

[31] Gross, M., and Pfister, H. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[32] Haeberli, P., and Akeley, K. The accumulation buffer: hardware support for high-quality rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1990), ACM, pp. 309–318.

[33] Havran, V. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[34] Heckbert, P. S. Filtering by repeated integration. *SIGGRAPH Comput. Graph. 20* (August 1986), 315–321.

[35] Heidrich, W., Slusallek, P., and Seidel, H.-P. An image-based model for realistic lens systems in interactive computer graphics. In *Proceedings of the conference on Graphics interface '97* (Toronto, Ont., Canada, Canada, 1997), Canadian Information Processing Society, pp. 68–75.

[36] Held, R. T., Cooper, E. A., O'Brien, J. F., and Banks, M. S. Using blur to affect perceived distance and size. *ACM Trans. Graph. 29* (April 2010), 19:1–19:16.

[37] Henning, C., and Stephenson, P. Accelerating the ray tracing of height fields. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2004), GRAPHITE '04, ACM, pp. 254–258.

[38] Hensley, J., Scheuermann, T., Coombe, G., Singh, M., and Lastra, A. Fast summed-area table generation and its applications. *Computer Graphics Forum 24* (2005), 547–555.

[39] Hullin, M., Eisemann, E., Seidel, H.-P., and Lee, S. Physically-based real-time lens flare rendering. *SIGGRAPH 2011* (2011).

[40] Kerr, D. A. The proper pivot point for panoramic photography. February 2008.

[41] Kolb, C., Mitchell, D., and Hanrahan, P. A realistic camera model for computer graphics.

[42] Kosloff, T., Tao, M., and Barsky, B. A. Depth of field postprocessing for layered scenes using constant-time rectangle spreading. *Graphics Interface 2009* (2009).

[43] Kosloff, T. J. *Fast Image Filters for Depth of Field Post-Processing.* PhD thesis, EECS Department, University of California, Berkeley, May 2010.

[44] Kosloff, T. J., and Barsky, B. A. An algorithm for rendering generalized depth of field effects based on simulated heat diffusion. Tech. Rep. UCB/EECS-2007-19, EECS Department, University of California, Berkeley, Jan 2007.

[45] Kosloff, T. J., Hensley, J., and Barsky, B. A. Fast filter spreading and its applications. Tech. Rep. UCB/EECS-2009-54, EECS Department, University of California, Berkeley, Apr 2009.

[46] Lee, S., Eisemann, E., and Seidel, H.-P. Depth-of-field rendering with multiview synthesis. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH ASIA) 28*, 5 (2009), 1–6.

[47] Lee, S., Eisemann, E., and Seidel, H.-P. Real-time lens blur effects and focus control. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH'10) 29*, 4 (2010), 65:1–7.

[48] Lee, S., Kim, G. J., and Choi, S. Real-time depth-of-field rendering using point splatting on per-pixel layers. *Computer Graphics Forum (Proc. Pacific Graphics'08) 27*, 7 (2008), 1955–1962.

[49] Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. Efficient depth peeling via bucket sort. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 51–57.

[50] Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. Single pass depth peeling via cuda rasterizer. In *SIGGRAPH '09: SIGGRAPH 2009: Talks* (New York, NY, USA, 2009), ACM, pp. 1–1.

[51] Merklinger, H. M. *Focusing the View Camera.* 2010.

[52] Michael Kass, Aaron Lefohn, J. O. Interactive depth of field using simulated diffusion on a gpu. Pixar Technical Memo 06-01.

[53] Nasse, H. H. Depth-of-field and bokeh. Tech. rep., Carl Zeiss, Camera Lens Division, 03 2010.

[54] Ozaktas, H., Kutay, M., and Zalevsky, Z. *The fractional Fourier transform with applications in optics and signal processing.* Wiley series in pure and applied optics. Wiley, 2001.

[55] Piponi, D. Two tricks for the price of one: Linear filters and their transposes, 2009.

[56] Policarpo, F., and Oliveira, M. M. Relief mapping of non-height-field surface details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), I3D '06, ACM, pp. 55–62.

[57] Policarpo, F., Oliveira, M. M., and Comba, J. a. L. D. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 155–162.

[58] Porter, T., and Duff, T. Compositing digital images. *SIGGRAPH Comput. Graph. 18* (January 1984), 253–259.

[59] Potmesil, M., and Chakravarty, I. A lens and aperture camera model for synthetic image generation. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1981), ACM, pp. 297–305.

[60] Reinhard, E., Ward, G., Pattanaik, S., and Debevec, P. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[61] Renner, E. *Pinhole photography: rediscovering a historic technique*. Focal Press, 2000.

[62] Salomon, D. *Transformations and Projections in Computer Graphics*. Springer-Verlag, 2006.

[63] Scheimpflug, T. Improved method and apparatus for the systematic alteration or distortion of plane pictures and images by means of lenses and mirrors for photography and for other purposes, May 1904.

[64] Scofield, C. 2 1/2—d depth-of-field simulation for computer animation. 36–38.

[65] Shade, J., Gortler, S., He, L.-w., and Szeliski, R. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 231–242.

[66] Sharma, K. *Optics: principles and applications*. Academic Press, 2006.

[67] Shirley, P., and Chiu, K. A low distortion map between disk and square. *journal of graphics, gpu, and game tools 2*, 3 (1997), 45–52.

[68] Steinert, B. Simulation of real photographic phenomena in computer graphics. Master's thesis, Universitat Ulm, 2009.

[69] Steinert, B., Dammertz, H., Hanika, J., and Lensch, H. P. A. General spectral camera lens simulation. *Computer Graphics Forum 30*, 6 (2011), 1643–1654.

[70] Veach, E. *Robust monte carlo methods for light transport simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[71] Whitted, T. An improved illumination model for shaded display. *Commun. ACM 23* (June 1980), 343–349.

[72] Wu, J., Zheng, C., Hu, X., and Xu, F. Lens dispersion simulation using dispersive lens model and spectral rendering method. In *ACM SIGGRAPH ASIA 2010 Posters* (New York, NY, USA, 2010), SA '10, ACM, pp. 48:1–48:1.

[73] Ziegler, R., Croci, S., and Gross, M. H. Lighting and occlusion in a wave-based framework. *Comput. Graph. Forum 27*, 2 (2008), 211–220.

[74] Žára, J., Beneš, B., Sochor, J., and Felkel, P. *Modern Computer Graphics*, 2 ed. Computer Press, Brno, 2005.

Appendix A

Algorithmic and mathematical details

A.1 Homogeneous coordinates

Homogeneous coordinates are a concept from projective geometry. They allow representing point at infinity consistently and also allow representing more transformations in matrix form [26]. For an n-dimensional linear space there is a $n+1$-dimensional space of homogeneous coordinates. Eg. for $n = 3$ we have homogeneous coordinates (x, y, z, w). 3D points are represented by vectors with $w \neq 0$ and those with $w = 0$ represent points at infinity or directions. The important thing is that there are many homogeneous coordinates differing just by a scaling factor which represent a single finite point, ie. $(x, y, z, w) \sim (ax, ay, az, aq) \forall a \in \mathcal{R}$.

The normal and homogeneous coordinates can be converted from one another just by scaling by a constant factor. Homogeneous coordinates of a point (x, y, z) are $(x, y, z, 1) \sim (ax, ay, az, a)$ and in the other direction $(x, y, z, w) \sim (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$ corresponds to the point $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$.

Homogeneous coordinates are also useful for representing translations or perspective transformations in a 3D space compactly within 4×4 matrices.

A.2 Intersection of ray with a hemispherical cap

In the sequential ray tracing it is needed to intersect rays which hemispherical caps representing the lens element surfaces. A hemispherical cap is just a hemisphere clipped by a plane perpendicular to the optical axis, the z axis. The circle at the clipping border corresponds to the element's aperture. A lens element is defined by the sphere radius and this aperture radius. The basic routine is just to find a ray-sphere intersection. After an intersection is found is remains only to test whether its xy projection lies within the aperture radius.

There are several ways to perform a ray-sphere intersection. One algebraic approach is to express both objects by parametric equations, put both into equality and find solutions of a quadratic equation. Another way which arises from the geometric situation is utilized in our program. It is described in [74]. Whichever the intersection-finding method is, there can be always three possible results: the ray either intersects the sphere at two points, the ray touches the sphere at one point

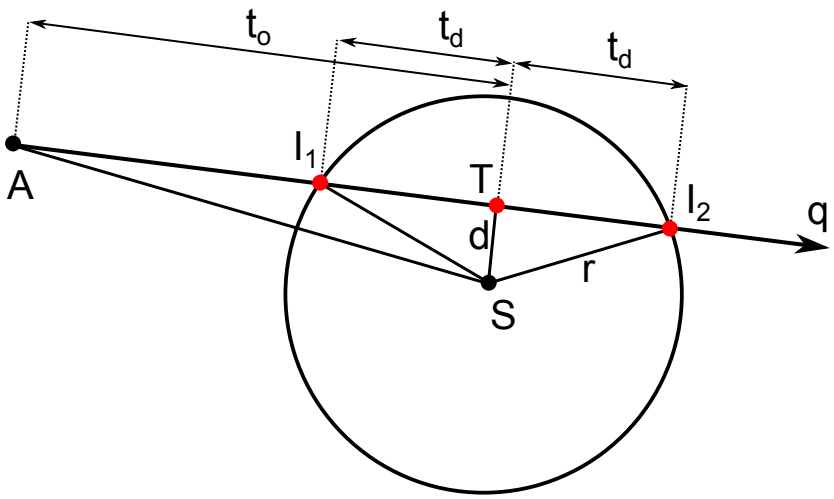


Figure A.1: Ray-sphere intersection – a 2D slice. The ray from A in direction $\vec{q}$ intersects the sphere centered at S with radius r at two points, I_1, I_2. T is the midpoint between the two intersections.

(being a double intersection), or the ray misses the sphere.

Let us have a sphere centered at S with radius r and a ray with origin A and normalized direction $\vec{q}$. The situation of two intersections I_1, I_2 – a 2D projection onto a plane defined by the ray and the sphere center – is shown in fig. A.1.

We find the midpoint T, the center of the segment I_1I_2. It is the projection of the sphere center relative to the ray origin $S - A$ onto the ray direction $\vec{q}$. The ray parameter t_0, equal to the length of the $T - A$ vector (since the $\vec{q}$ is normalized), can be obtained as $t_0 = |T - A| = |(S - A)\dot{\vec{q}}|$. The triangles STI_1 and STI_2 are right angled since T is a projection of S onto the ray. So that $t_d = |T - I_1| = |T - I_2|$ and $t_d^2 = r^2 - d^2$, where $d = |T - S|$. d^2 can be computed as $d^2 = |S - A|^2 - t_o^2 = (S - A)\dot{(}S - A) - t_o^2$. Finally $t_d^2 = r^2 - (S - A)\dot{(}S - A) + |(S - A)\dot{\vec{q}}|$. There can be three results:

$$t_d^2 \begin{cases} > 0 & \text{two intersection at} A + (t_o \pm t_d)\vec{q} \\ = 0 & \text{one intersection at} A + t_o\vec{q} \\ < 0 & \text{no intersection} \end{cases}$$

Case must be taken with numerical precision, especially equality of floating-point numbers must be performed with some epsilon tolerance.

A.3 Height field intersection

As the height field intersection algorithm is the cornerstone of the IBRT we provide the complete GLSL code of the simpler rasterized intersection algorithm described in section 4.2.2.

```
vec2 getPixelCorner(vec2 pos, vec2 relDir) {
  vec2 corner = floor(pos);
  float eps = 0.00001;
```

```
  if ((relDir.x < 0.0) && (pos.x - corner.x < eps)) corner.x -= 1.0;
  if ((relDir.y < 0.0) && (pos.y - corner.y < eps)) corner.y -= 1.0;
  return corner;
}

vec3 intersectHeightFieldPerPixel(vec3 startPos, vec3 endPos) {
  vec3 start = vec3((startPos.xy * screenSize), startPos.z);
  vec3 end = vec3((endPos.xy * screenSize), endPos.z);
  vec3 dir = end - start;
  float epsilon = 0.0001;
  if (abs(dir.x) < epsilon) { dir.x = epsilon; }
  if (abs(dir.y) < epsilon) { dir.y = epsilon; }
  vec2 rayStep = sign(dir.xy); // {-1,1}^2
  vec2 currentPixel = getPixelCorner(start.xy, rayStep);
  vec2 endPixel = getPixelCorner(start.xy + dir.xy, -rayStep);
  vec2 boundary = currentPixel + max(rayStep, 0.0); // {0,1}^2
  vec2 rayDirInv = 1.0 / dir.xy;
  vec2 tMax = (boundary - start.xy) * rayDirInv;
  vec2 tDelta = rayStep * rayDirInv;
  vec3 entry = start;
  vec3 exit = start;
  vec3 color = vec3(0.0);
  while (currentPixel != endPixel) {
    if (tMax.x < tMax.y) {
      exit = start + tMax.x * dir;
      tMax.x += tDelta.x;
      currentPixel.x += rayStep.x;
    } else {
      exit = start + tMax.y * dir;
      tMax.y += tDelta.y;
      currentPixel.y += rayStep.y;
    }
    if (testIntersection(currentPixel, entry, exit, color)) {
      break;
    }
    entry = exit;
  }
  testIntersection(currentPixel, entry, endPos, color);
  return color;
}

// code for exactly four layers
bool testIntersection(vec2 currentPixel, vec3 entry, vec3 exit,
    inout vec3 color) {
  vec2 depthTestPos = (0.5 + currentPixel) * screenSizeInv;
  vec4 layersZ = texture2D(packedDepthTexture0, depthTestPos);
  float epsilonForDepthTest = 0.01;
  ivec4 comparison = ivec4(sign(layersZ - vec4(entry.z))
```

```
    + sign(vec4(exit.z + epsilonForDepthTest) - layersZ));
  if (comparison.x == 2) {
    color = texture2DArray(colorTexture, vec3(depthTestPos, 0)).rgb;
    return true;
  } else if (comparison.y == 2) {
    color = texture2DArray(colorTexture, vec3(depthTestPos, 1)).rgb;
    return true;
  } else if (comparison.z == 2) {
    color = texture2DArray(colorTexture, vec3(depthTestPos, 2)).rgb;
    return true;
  } else if (comparison.w == 2) {
    color = texture2DArray(colorTexture, vec3(depthTestPos, 3)).rgb;
    return true;
  }
  return false;
}
```

Appendix B

User's guide

B.1 System requirements

The C# programs need for running the Microsoft .NET Framework 3.5 or newer to be installed. They were tested on Windows 7 Professional 64-bit and Windows XP SP3 32-bit. A GPU supporting OpenGL 2.1 with proper drivers is needed for running the interactive DoF renderer and other parts utilizing a GPU. The program was successfully tested on NVIDIA Quadro NVS 140M, so this or a newer GPU should be sufficient. The Mathematica notebooks require Wolfram Mathematica 8 or Wolfram Mathematica Player.

B.2 Installation

In case the required dependencies are installed, it is sufficient to copy the content of the CD to a writable directory and run the particular programs.

B.3 Usage

B.3.1 Interactive DoF renderer

The program can be started via the `BokehLab.InteractiveDof.exe` executable. It consists of a single OpenGL window and is controlled by keyboard and mouse commands. Their list can be shown in the help dialog window by pressing the `F1` key. In addition `F11` toggles between windowed and full-screen mode and `F12` displays a dialog windows with status information (camera, navigation, etc.). Continuous information on current FPS, frame time, total accumulation time and average FPS and frame time is displayed in the window title.

Rendering modes

There are several modes of rendering – plain rasterization, multi-view accumulation, image-based ray tracing and several visualizations of internal structures. The keyboard commands are following:

`F2` plain rasterization (pinhole view)
`F3` (incremental) multi-view accumulation

`[/]` decrease/increase the total number of samples
`Tab` toggle incremental rendering

`F4` image-based ray tracing
`F5` incremental image-based ray tracing

`[/]` decrease/increase the total number of samples

`F6` visualization of layered depth images

`Tab` select layer type: color/depth/packed-depth
`O/P/U` select previous/next/first layer

`F7` visualization of N-buffers

`Tab` select channel mask: min/max, min, max
`O/P/U` select previous/next/first layer

For demonstration purposes the scene is hard-coded and some of its parts can be user-controlled, such as the scene complexity.

`F8` toggle showing more complex models (dragon, teapot, etc.)
`F9` change the number of shown complex models
`F10` toggle showing white or colorized stars

Navigation and camera parameters

Both extrinsic (position, orientation) parameters and intrinsic ones (lens model and sensor parameters) of the camera can be controlled, some of them either by keyboard or mouse. The navigation is similar to what the user expects from common computer games.

`W/S/A/D/Q/E` move forwards/backwards/left/right/down/up
Up/down/right/left arrow change the orientation – look up/down/right/left
Mouse left button + drag change the orientation
`Shift+[WSADQE]` move more precisely
`Shift+R` reset the navigation parameters

Thin lens model and the tilt-shift sensor parameters are a bit more elaborate.

`Page up/Page down` increase/decrease focus plane distance (focus forth/back)
Mouse right button + drag up/down focus forth/back
Mouse wheel up/down focus forth/back
`Home/End` increase/decrease aperture radius
`Plus/Minus` increase/decrease focal length
`Delete/Insert` increase/decrease angle of view (zoom in/out)
`X/Z` increase/decrease sensor tilt around X axis
`V/C` increase/decrease sensor tilt around Y axis
`N/B` increase/decrease sensor shift in X axis
`,/M` increase/decrease sensor shift in Y axis
`R` reset the camera parameters

Appendix C

CD contents

The enclosed CD is organized as follows:

- `bin/`
  - `BokehLab.InteractiveDof.exe` – interactive GPU DoF renderer (IBRT, MVA), visualization of depth-peeled layers, N-buffers
  - `BokehLab.ImageBasedRayCasting.exe` – CPU-based prototype of a simpler IBRT renderer with sequential lens ray tracing and LRTF evaluation
  - `BokehLab.Spreading.GUI.exe` – CPU-based interactive prototype of fast spreading filters
  - `BokehLab.Demo.ComplexLensTracing2d.exe` – visualization of sequential ray tracing inside a complex lens
  - `BokehLab.Demo.HeightFieldIntersection.exe` – visualization of the ray footprint traversal algorithm
- `src/`
  - `bokehlab/` – BokehLab source codes
  - `mathematica/` – Mathematica notebooks - models, prototypes, etc.
- `thesis/`
  - `latex/` – LaTeX sources of the thesis
  - `pdf/` – PDF output
- `results/`
  - resulting images, measurements, etc.