Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 290

Rendering of Densely Recorded Light Fields

Sebastian Zahn

Course of Study: Informatik

Examiner: Jun.-Prof. Dr.-Ing. Martin Fuchs

Supervisor: Dipl.-Ing. Alexander Wender

Commenced: November 11, 2015

Completed: May 13, 2016

CR-Classification: I.3.3, I.3.6, I.4.10, I.4.2

Abstract

Light fields present an alternate approach for producing images of a high degree of realism, by capturing real world data in the form of images, or by traditional techniques like raytracing a synthetic scene. In both cases, the produced data can be utilized to render images from positions which were previously not recorded or with different camera parameters and configurations. The resolution and spatial density, at which such light fields are recorded, influence the mass of produced data that has to be handled.

This work focuses on densely recorded light fields and attempts to produce synthesized images computed from the available data, defined by a camera moving through space. Synthesized cameras are also able to change their aperture size and focus setting. Rendered cameras behave according to the thin lens model. A method for extraction of relevant light field images is proposed. For rendering of the data, two different approaches are evaluated. The first approach collects rays which are present in the light field in synthetic sensor plates. In an alternative approach, rays are collected in a standard hash map, and rendered by constructing and querying a kd-tree. Both approaches yield a set of properties which make them useful in different scenarios, and can also be combined to an hybrid renderer. The proposed system is intended to run on several machines in parallel.

Kurzfassung

Lichtfelder bieten einen alternativen Ansatz zur Erstellung von Bildern mit hohem Realitätsgrad, indem Bilddaten durch echte Kameras aufgenommen, oder diese von synthetischen Szenen erstellt werden. In beiden Fällen können diese Daten verwendet werden, um neue Bilder aus völlig neuen Positionen oder geänderten Kameraparametern- und Konfigurationen zu erstellen. Die Auflösung, und die räumliche Dichte, mit der das Lichtfeld aufgenommen wurde, beeinflussen die Masse an produzierten Daten, die es zu verarbeiten gilt.

Diese Arbeit konzentriert sich auf dichte Lichtfelder, und versucht synthetische Aufnahmen aus den vorhanden Daten zu produzieren, definiert durch eine Kamera welche sich durch den Raum bewegt. Synthetische Kameras sind auch in der Lage, ihre Blende und Fokuseinstellung zu verändern. Diese Kameras verhalten sich nach dem Modell dünner Linsen. Eine Methode zur Bestimmung von relevanten Bildern des Lichtfelds wird vorgeschlagen. Zur Erstellung der neuen Bilder werden zwei verschiedene Ansätze evaluiert. Der erste Ansatz sammelt Strahlen des Lichfelds in synthetischen Sensorplatten. Alternativ werden Strahlen in einer Hash Map gesammelt, und nach Aufbau

eines kd-Baums gewünschte Strahlen angefragt. Beide Ansätze haben Eigenschaften, welche sie in verschiedenen Szenarien verwendbar machen, auch als kombinierter Hybrid-Renderer. Das vorgeschlagene System ist darauf ausgelegt, parallel auf mehreren Rechnern zu laufen.

Contents

List of Figures

List of Tables

1 Introduction

Commonly, realistic renderings of scenes are produced by providing a geometric representation of the scene, as well as material definitions of the surface of this geometry and their reflective properties [GGSC96]. Afterwards, it is possible to simulate light traversal through the scene to obtain a 2-dimensional synthetic image. However, modeling light interacting with the scene's surfaces becomes increasingly complex with the desired level of realism. One may determine the color of a pixel of the desired image by casting rays into the scene and determining the color of the object being hit first, essentially traversing the path of a light ray in the opposite direction [Gla89]. Advanced effects like reflection (diffuse and specular), refraction, shadows with a penumbra or depth of field resulting from non-pinhole cameras can be approximated.

To acquire realistic results, a large number of rays have to be cast and therefore many intersection test have to be performed, leading to long rendering times. Effects like mat reflections caused by surfaces which are not reflecting incoming light in a single direction, but scattering it instead lead to an exponential rise of necessary rays with every intersected mat surface. However, Kajiya described an alternative ray tracing algorithm called *path tracing*, which reduces this problem [Kaj86]. Still, his approach of evaluating the *rendering equation* requires many samples, and therefore many rays traversing a scene.

In the case of image sequences of moving cameras, the computational effort required may rise even further.

In recent times, another approach in the form of *light fields* gained popularity. Conceptually, the radiance of all light rays in a scene is expressed as a 4-dimensional function L, which can be evaluated via a position and direction (with some constraints on valid positions). An image can be interpreted as a 2D slice of L, meaning that L can be constructed by providing all corresponding slices [LH96]. In a later step, one may evaluate this function to obtain radiance of an arbitrary ray, which contributes to a novel image. This approach also enables the recording of light fields of real-world scenes- it does not matter if the images are synthesized or captured by a real camera. Note that depth information is not necessary to extract valid new views- In principle, one may only evaluate L with the desired parameters.

The 4D function can be recorded by taking many images of the scene from viewpoints located around it.

Note that if L is constructed from real world data (e.g. images taken with a camera), global illumination effects resulting from many light bounces [Kaj86] are captured too.

One of the main advantages of light fields is that fast rendering can be achieved, given that evaluating L is not computationally expensive. The user is then able to explore the light field from different vantage points, essentially moving around the captured scene.

The light fields to be rendered in this approach have the property of being dense, meaning that they consist of many more images than regular light fields, captured in a spatially dense manner. Such a dense light field consists of millions of images resulting in multiple TB of data [Sie15]. Dense light fields can not be fit into the random access memory of typical render nodes. In this work it is attempted to exploit the light field's density to achieve high quality renderings of new views.

1.1 Motivation

In order to create new renderings, the light field has to be processed in some way. However, the large amount of data also has to be limited to a subset of the data, and may be additionally compressed in a lossy fashion, without compromising image quality of produced renderings too much. The requirement of rendering camera movements instead of still images introduces further complications.

This work builds on the light field capturing process described by Siedelmann [Sie15]. Dense light fields require specialized hardware for recording and storing, because of the large amount of data that is to be processed per time unit. In turn, such light fields can be used to synthesize new images of high quality, as shown by Siedelmann. However, the large amount of data presents some drawbacks, since spatially small parts of the light field already contain a large amount of data, which has to be processed.

1.2 Goals

Image Sequences

The rendering system should be capable of producing sequences of images defined

by a camera which moves through space arbitrarily, however constraint to areas where light field data is available.

Synthetic Camera
Cameras should support finitely sized apertures. This is necessary to produce realistic depth of field effects.

Parallelization
The system should be capable of distributing the required data and computational effort among multiple nodes.

Progressive Results
Intermediate results should be returned periodically.

Data reduction
Optionally, data should be compressed by a lossy compression scheme to reduce the amount of data that has to be passed to nodes.

1.3 Challenges

The main challenge is the size of the light field- for example, the light field presented by Siedelmann [Sie15] has a total (lossless compressed) size of 11 TB, and twice the uncompressed size. Even if only a fraction of this available data is used for rendering, main memory requirements would still be infeasible. Data can be restricted to those parts a camera movement requires for rendering, however this could still yield a large amount of data. This data which is to be processed should be split up further to enable parallelized processing. Depending on the required image quality, data may further be reduced to some subset.

The available dataset is composed of a stream of images, as well as information about their capture position and rotation in space, a representation that has both advantages and disadvantages for rendering, depending on the synthetic camera's attributes and movement.

Geometric approximations of the scene may help with rendering [GGSC96], but are not available for the given dataset, i.e. no depth information is available or computed for rendering. If the dense light fields full resolution is used, this might not even be necessary.

Many approaches for light fields enable real time rendering at interactive rates. Achieving this is difficult for dense light fields, since the data requirement for full exploitation of the available dataset is way higher than for usual light fields. The renderer presented by

Siedelmann [Sie15] already uses thousands of light field images to accomplish realistic synthetic aperture rendering for a single new view. This makes dense light fields unsuited for interactive rendering if their full resolution is to be exploited, however, it is still possible to use the available data in offline rendering.

2 Fundamentals

This chapter contains information regarding light fields, optics and cameras, as well as some useful concepts applied in this work.

2.1 Light Fields

The *plenoptic function* is a function of seven dimensions, in explicit of position (3-dimensional x, y, z), direction (spherical coordinates θ, ϕ), time t and wavelength λ, denoted as $P(\theta, \phi, \lambda, t, x, y, z)$, describing radiance at the given position, direction and wavelength, at the given point in time. Evaluating P for a given set of parameters would return every possible view acquirable by the human eye or a camera [AB91]. P can be restricted to being temporally static by removing t. Similarly λ can be restricted to some fixed wavelength(s), reducing P's dimensionality to five. Considering some object in world space, light rays reflected of its surface can be assumed carrying constant radiance, if such a ray is not intersecting anything along its path. By assuming surrounding space around an object or scene being empty, and restricting positions for evaluation of the dimensionally reduced version of P denoted as P', choosing an alternate parameterization can further reduce the dimensions by one [GGSC96; LH96]. Query rays can be constructed against this parameterization, and used to evaluate the function. This 4-dimensional function is called *light field* (Alternatively, Gortler et al. termed this function *lumigraph*, although their work differs slightly in detail).

2.1.1 Parameterization

Consider the ray r:

$$r(\lambda) = o + \lambda \cdot d$$

With o denoting the ray's origin, and d its direction. Parameterizing in this context corresponds to finding four variables that describe the ray. Note that a ray r' with the same (or opposite) direction, but different origin, is parameterized to the same variables. Therefore those variables can be used to identify every ray in space.

Two-Plane Parameterization

The *two-plane parameterization* is commonly used, for example by Gortler et al. [GGSC96], as well as Levoy and Hanrahan [LH96].

Two parallel planes are placed in space, see Figure 2.1. By placing two perpendicular axes, points on the plane can be identified. Additionally, a ray intersecting both planes can be parameterized by its intersection points with both planes (s, t), and (u, v) respectively. Note that in this work, the front plane is denoted as the st-plane, and the back plane as the uv-plane. This arrangement of planes is sometimes also called *light slab*.

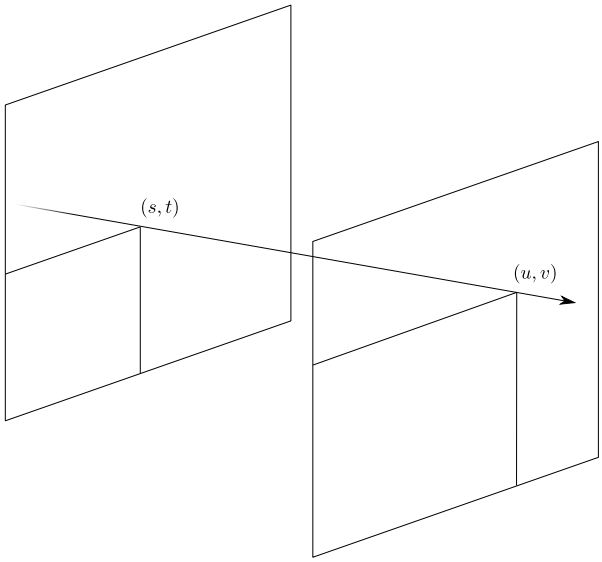


Figure 2.1: The two-plane parameterization. The ray intersects both planes and is parameterized by both intersection points.

Spherical Parameterization

Spherical Parameterizations of rays can be obtained by intersecting a bounding sphere S, and describing the intersection as spherical coordinates (θ, ϕ). Additionally, the direction of a ray is described by intersecting a second sphere S', having its origin at the intersection (θ, ϕ). Spherical coordinates (θ', ϕ') then identify the ray's intersection with S' [Ihm97].

The ray r striking through a sphere S can also be parameterized by the two intersection points where r enters and leaves the sphere volume. This parameterization is known as *sphere-sphere parameterization* [TRK07].

Alternatively, r can be parameterized by intersecting both a sphere S and the plane P containing S's center point and having r's direction as its surface normal. This is known as *sphere-plane parameterization* [TRK07].

Light fields represented in a spherical parameterization show lower amounts of artifacts due to the uniformity of spheres [CLF98; TRK07]. Note however, that spherical coordinates result in denser parameterization at the poles.

2.1.2 Capture

In principle, light fields can be obtained by rendering many views of the scene from different viewpoints [GGSC96; LH96]. For example, to obtain a two-plane parameterized light field, view points are chosen on the discretized st-plane. The camera's view direction is perpendicular to the st-plane. To make sure the camera captures the uv-plane, its viewing frustum is moved accordingly. A standard ray tracing program can be used to obtain the light field's radiance information.

Real world light fields are more difficult to acquire, since precise information about the camera's location and rotation (the extrinsic parameters) in each recorded image are necessary. Illumination of the real world scene must be constant over time, since light field capture usually takes some time. Additionally, the capturing camera's aperture should be small, to maximize depth of field such that the entire scene is in focus [LH96].

Images can be obtained by a hand-held camera [GGSC96], as well as in a more organized approach like a planar gantry [LH96]. Another method is the use of a camera array [Vai07; WJV+05], making use of a high number of cameras placed in a rectangle. Since light fields can be captured immediately this way (as opposed to a single camera moving slowly over time), this enables the implementation of a real time system as proposed by Yang et al. [YEBM02].

Light fields can also be captured by a single *plenoptic camera* as presented by Ng et al. [NLB+05]. A *microlens array* is introduced between the camera system's main lens and the sensor plate. Rays focused by the main lens onto the array are scattered on the array's lenses, producing a sharp image of the rays originating on the main lens (i.e. rays that are incoming from the outside, refracted by the main lens) on the sensor.

It is also possible to capture the entire scene by moving the camera around the scene (or, moving the entire scene while keeping the camera static), as it is done by Siedelmann [Sie15]. In his approach, the camera is mounted on an arm, looking at the center of the scene. This arm can be moved vertically along the hemisphere whose radius is given by the length of the arm. The scene is placed on a turntable, which moves at a steady rate

during capture. Note that this method is equivalent to moving the camera around the scene.

Gortler et al. used markers visible in the scene to obtain the intrinsic and extrinsic camera parameters [GGSC96]. Siedelmann used a similar approach, with a large number of visible markers [Sie15].

Levoy and Hanrahan filtered their light fields to remove aliasing appearing during rendering caused by high frequencies [LH96].

2.1.3 Compression

Light field compression is closely related to image compression. Therefore, many image compression algorithms can be directly used or modified to work on light field data. Compression rates are usually high, since light field data is highly redundant in each of the four dimensions [LH96]. Since it is generally not possible to know which parts of the light field dataset is accessed, random access to data is advantageous. Levoy and Hanrahan proposed a *vector quantization* scheme to compress their light fields. This lossy compression technique maps vectors of data to a set of vectors called the *codebook*. The codebook vectors are chosen such that they represent the uncompressed data as closely as possible. Compression is done by mapping data vectors to the index addressing the vector in the codebook that is approximating the given data as closely as possible. By indexing the codebook with a given index, the representing vector can be obtained, enabling random access to the dataset. Gortler et al. propose to use a compression technique similar to JPEG and MPEG [GGSC96]. Camahort et al. used JPEG compression in their work [CLF98].

Many light field systems use wavelets as part their compression scheme, see Section 2.5 for an introduction. A 4D wavelet compression scheme was used by Magnor et al. to compress the light field [MEG00]. Wavelet decomposition was applied in the image dimensions and neighboring pictures. Peter and Straßer proposed a method to compress and progressively obtain the compressed wavelet coefficients when needed [PS01]. Both approaches use a hierarchical tree structure to organize the coefficients, and allow progressive decoding.

Note that most image compression algorithms are lossy, and compression usually takes place after the light fields has been captured. The dense light field dataset used in this work is compressed in a lossless fashion. Siedelmann's proposed scheme works in an online fashion, directly compressing recorded raw images at very high speeds, to overcome bandwidth limits introduced by storage media [Sie15]. He also suggested that the light field should be compressed via offline algorithms after the light field has been recorded.

2.2 Optics

Light field cameras have to be modeled before the radiance information recorded in the light field's images can be utilized to generate new views. Additionally, synthetic cameras have to be defined by some suitable parameters, and should be able to simulate certain effects and properties.

2.2.1 Homogeneous Coordinates

Homogeneous coordinates are an alternative representation of points, respectively vectors [Sze10], and can be used to apply translation and rotation to a vector/point by applying a single matrix multiplication [SM09].

Vectors in 3D are extended by one dimension

$$\tilde{v} = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{pmatrix} \tag{2.1}$$

where $\tilde{w}$ denotes the new component. By dividing $\tilde{v}$ through w, one obtains the *augmented vector*:

$$\bar{v} = \begin{pmatrix} \tilde{x}/\tilde{w} \\ \tilde{y}/\tilde{w} \\ \tilde{z}/\tilde{w} \\ \tilde{w}/\tilde{w} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{2.2}$$

This can be used to obtain the *inhomogeneous vector* $v = (x, y, z)^T$.

Homogeneous vectors $\tilde{v}$ which result in the same augmented vector $\bar{v}$ (and therefore the same v) are seen as equivalent. In the case of $\tilde{w} = 0$, no inhomogeneous vector can be obtained. These are called *points at infinity* [Sze10].

By augmenting the component $w = 1$ to any 3D vector, a transformation matrix

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.3}$$

can be constructed, where $a_{i,j}$ describe the components of a rotation matrix and t_i describe the components of a translation vector. Multiplying the augmented vector with M will rotate and translate the vector with a single matrix multiplication.

2.2.2 Pinhole Camera

Pinhole cameras are idealized cameras with an aperture size of zero. Incoming light rays fall through the aperture and hit the film positioned behind it, resulting in a flipped image of the observed scene [HSS97], see Figure 2.2. This produces images which are entirely sharp.

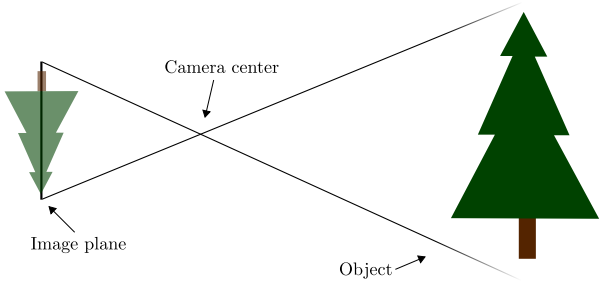


Figure 2.2: Depiction of a pinhole camera. Note that the object is flipped in image space.

2.2.3 Raytracing

It is easy to simulate a pinhole camera by constructing rays which pass through the pinhole's center point and hit the camera's pixels [Gla89]. Typically, when rendering via raytracing, the ray is cast in the opposite direction of the light path, i.e. the ray is cast outwards toward the scene. By obtaining this ray's radiance, images of the scene can be created.

2.2.4 Perspective Projection

A pinhole camera can also be modeled as a perspective projection, mapping 3-dimensional points in the scene to 2-dimensional points, which could correspond to pixels in the image.

As seen in [Sze10], this can easily done by dividing a 3D point p by its z-component i.e.

$$p' = \begin{pmatrix} x/z \\ y/z \\ 1 \end{pmatrix} \tag{2.4}$$

after it has been transformed with the camera's *extrinsic* parameters $[R|t]$. To obtain a pixel coordinate, the point resulting from the perspective projection has to be transformed using the camera's sensor plate properties. The matrix K is called *calibration matrix*, and also regarded as *camera intrinsics*. Following `OpenCV`'s [CV15] conventions, K looks like

$$K = \begin{pmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

with f_x and f_y describing focal lengths for their respective dimension, and c_x, c_y the intersection of the optical axis with the sensor plate. The parameter s describes the possible skew of the sensor plate, however, it is commonly zero [Sze10]. Note that it is more practical to express all variables described above in pixel units instead of millimeters, because this will map transformed points to pixel indices.

Multiplying the intrinsic and extrinsic matrix yields the *camera matrix*

$$P = K\,[R|t]\,. \tag{2.6}$$

This matrix contains the world space to view space (camera space) transformation of world space points via $[R|t]$, and the subsequent transformation to image space. Note that the resulting vector is expressed in homogeneous coordinates, so division by w (respectively z) is necessary.

Summarizing, world space points $p_{ws} = (x, y, z)^T$ can be transformed to image space pixels by multiplication with P, i.e.

$$p_{is} = P \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ w \end{pmatrix} \tag{2.7}$$

and subsequent division of p_{is} by w.

Distortions Caused by Lenses

Unfortunately, lenses of real cameras introduce *radial* and *tangential* distortion to their recorded images. This results in lines appearing curved in the recorded images, leading to a slight "fisheye"-effect. Correction can be applied by remapping distorted pixels (x_d, y_d) to pixels (x, y). For example, a model using low-order polynomials remaps pixels by multiplying a polynomial

$$x = x_d(1 + \kappa_1 r_c^2 + \kappa_2 r_c^2)$$
$$y = y_d(1 + \kappa_1 r_c^2 + \kappa_2 r_c^2)$$

where $r^2 = x_d^2 + y_d^2$ describes the radial distance of a point, i.e. the distance from the image center, and κ_1, κ_2 are the *radial distortion parameters* [Sze10]. Note that the exemplary distortion model described above is applied before multiplication with the intrinsic matrix K.

For example Farag et al. [EF03], describe a method for determining those coefficients. `OpenCV` uses the approach described in [Zha00].

2.2.5 Thin Lens Model

To model properties of non-pinhole cameras, a more complex model than the pinhole model can be used: The *thin lens model* [Sze10; ZZ09]. The lens has a diameter d, and is positioned with distance b and g between image plane and focus plane. Points lying on the focus plane are projected sharply onto the image plane. Note that in this model, the lens is modeled as a single plane, meaning that it is infinitely thin. The lens's properties follow the *lens law*

$$\frac{1}{f} = \frac{1}{g} + \frac{1}{b}. \tag{2.8}$$

Rays, that are parallel to the optical axis o will pass through the focal point p_f on the optical axis (after being refracted by the lens). p_f is distanced f from the lens on the image plane's side. See Figure 2.3 for a depiction of the model. Note that f approaches b when g is increased, leading to the angle between rays and the optical axis becoming zero. If a ray happens to strike through the lens's center point, it is not refracted, but simply passes through [ZZ09].

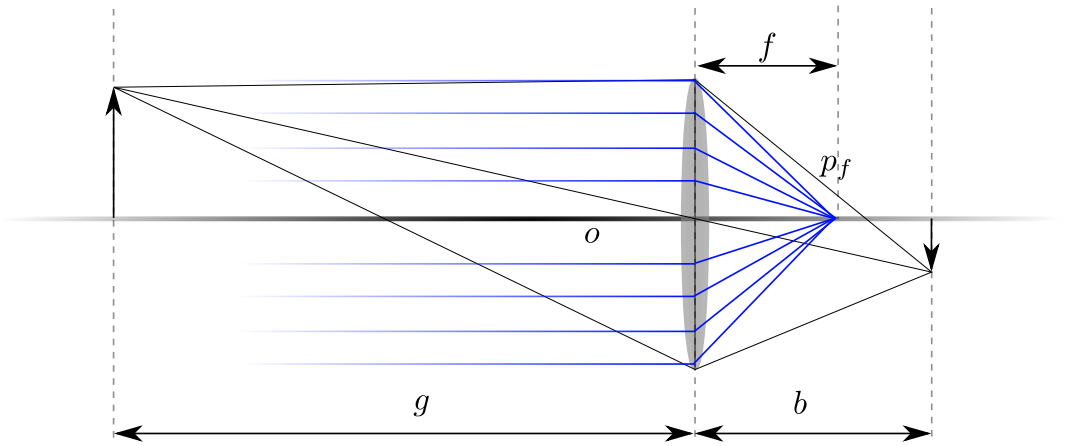


Figure 2.3: The thin lens model. Depicted in blue are rays originating at infinity, being refracted and intersecting the optical axis at the focal point p_f

2.2.6 Depth of Field

Points which lie beyond or in front of the focus plane cannot be imaged sharply. However, since sensor plates (synthetic and real camera sensors) consist of pixels with a certain resolution, points which are only offset away in a certain range Δg from the focus plane will appear perfectly sharp nonetheless. This range is called *depth of field* [ZZ09]. Let $\tilde{g}$ be the distance of a point placed in front of the lens, and Δg the distance of this point from the ideal focus plane. The *circle of confusions*' diameter z is given by

$$z = \frac{d}{b} \cdot \Delta x \tag{2.9}$$

where Δx describes the distance of the sensor plate to the ideal image plane [ZZ09]. See Figure 2.4 for a depiction. As long as the *circle of confusion* is smaller than the spatial pixel resolution, the image will remain sharp. Let r be the diameter of a pixel on the camera sensor (r therefore limits the sensor's maximum resolution). The circle of confusion is related to *bokeh*, which is the appearance of the circle of confusion in the image [WZH+10]. Such effects are oftentimes desired as they allow photographers to bring the attention of the viewer to certain objects in scenes (See Figure 2.5). The depth of field range [ZZ09] is given by

$$\Delta g \simeq \frac{rfg^2}{df^2},\ g \gg f. \tag{2.10}$$

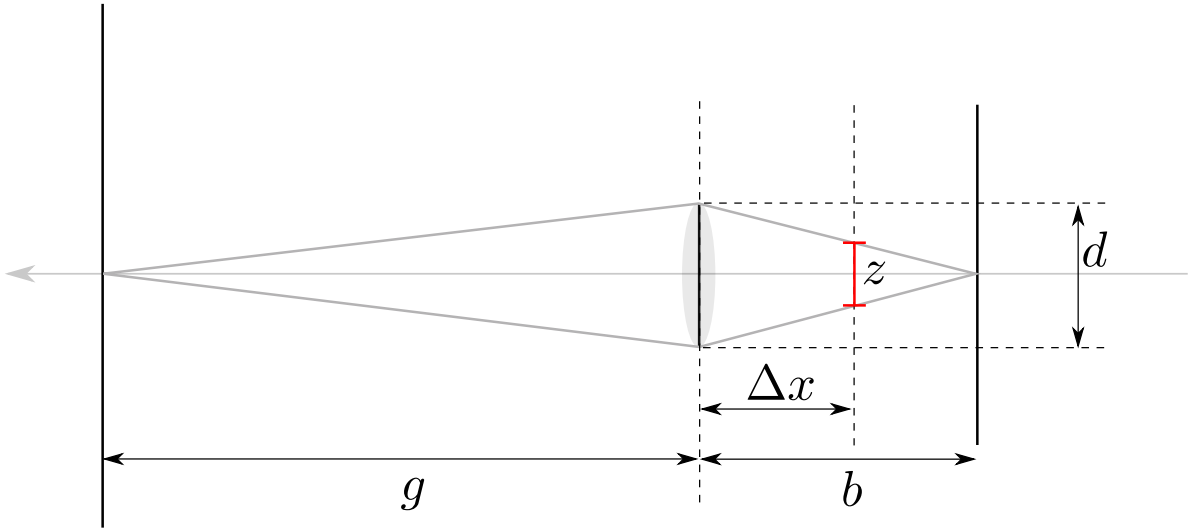
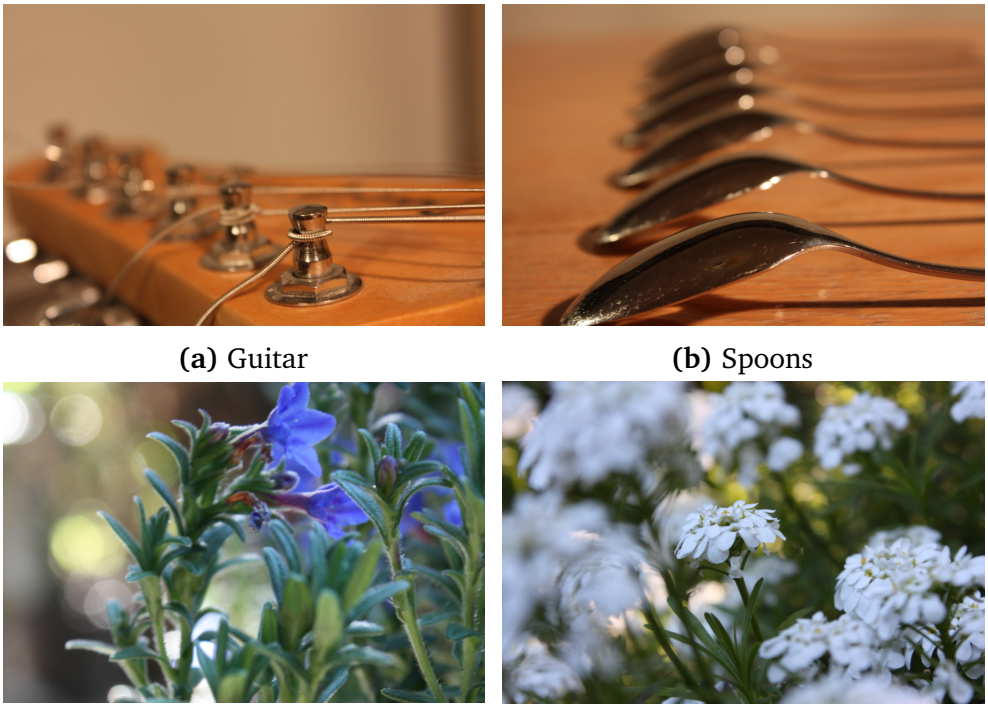


Figure 2.4: Circle of confusion with diameter z occurring when the image plane is moved forward. The ideal image is still imaged at distance b.

The f-number is defined as [Sze10; ZZ09]

$$N = f/d. \tag{2.11}$$



(a) Guitar **(b)** Spoons

(c) Flower 1 **(d)** Flower 2

Figure 2.5: Images taken with a Canon EOS 450D with large aperture setting. Figure a shows a guitar's head with tuning pegs. Sharpness decreases rapidly as objects are more distanced to the focus plane. Figure b shows an arrangement of spoons. Highlights are imaged as uniformly blurred circles. This becomes even more apparent in Figure c: Objects very far away appear as blurry circles. In Figure d, the focus plane is placed at the sharply imaged flower. Objects in front and behind the focus plane appear out of focus.

Usually, the f-number is expressed as "f/N". Given a lens's focal length f, the aperture's diameter d can be obtained from a f-number setting. The f-number sequence commonly found on real cameras decreases image brightness by a factor of $\frac{1}{2}$ with increasing N in the sequence [ZZ09].

2.3 Rendering Techniques

In the context of computer graphics, different approaches for synthesizing a novel image from some input data exist, be it a description of geometry, or a set of images. In contrast to some light field rendering methods, usually some kind of 3D information has to be present in addition to the images (e.g. depth information).

Another well established approach are raytracing methods, however these techniques are usually applied to synthetic scenes. The light field renderer presented in this work should support finitely sized lenses. Distributed raytracing is one method to obtain the resulting depth of field effect.

2.3.1 Image-Based Rendering

Since the light field dataset used in this work consists of images with camera extrinsics, image-based rendering techniques could be used to obtain new views.

One popular approach for rendering novel views from present images is *view interpolation* as proposed by Chen and Williams [CW93]. The first step corresponds to computing two *morph map*, one for each image in a pair. These mappings describe the offset of a pixel's position in image space to the other image. Pixels in novel views are computed by computing an interpolated offset vector and moving the source image's pixels by the interpolated vectors. To interpolate views from multiple images, a graph is constructed and images are assigned to nodes. Edges are bidirectional and are associated with the morph maps between their images. From the location of the desired camera, it is possible to interpolate the morph maps and image pixels. Holes may appear since applying the offset may map multiple pixels to a single pixel. Additionally, for each pixel, the depth values have to be available such that the morph map can be computed, in case of real world camera images, they have to be acquired before view interpolation is possible, for example via stereo matching algorithms [Sze10].

An alternative to view interpolation was presented by Seitz and Dyer: *view morphing* [SD96]. Their methods consists of *prewraping* both input images, such that they become parallel to the line connecting the cameras that took the images. Afterwards pixels are interpolated, and the resulting image projected to the desired image plane. Some *correspondence* information (given some pixel in image A, locate where this point is imaged in some image B [SS; Sze10]) in both images is necessary, but no information about the geometry of the viewed scene. Correspondence can be obtained by letting users mark corresponding features in the input images. Note that depth information can help with visibility determination of projected pixels in the rendered image, similar as in the view interpolation approach.

2.3.2 Distributed Raytracing

Cook et al. proposed *distributed raytracing* to simulate effects like smooth shadows, motion blur, diffuse reflections and depth of field [CPC84]. Standard raytracing simulates

a pinhole camera by casting rays from image plane points through the camera's center point towards the scene. In distributed raytracing, instead of casting single primary and secondary rays, multiple rays are cast, with their direction (and possible origin in case of lens sampling) distributed such that some desired effect is simulated. Spatial oversampling is commonly used to minimize aliasing, these extra rays can also be used to accomplish new effects.

From this works perspective, it is relevant how depth of field effects are rendered. In the first step, a ray is cast from the sensor plane through the center of the lens towards the focus plane, where it is intersected in some point p. Then, a point p_l is selected on the lens, and the color for ray $r(\lambda) = p_l + \lambda(p - p_l)$ is computed [CPC84; PH10]. The pixel's color value is the averaged color of all sampled rays [HSS97], approximating the integral which describes the pixels color value. See Figure 2.6 for a synthetic view obtained by this technique.

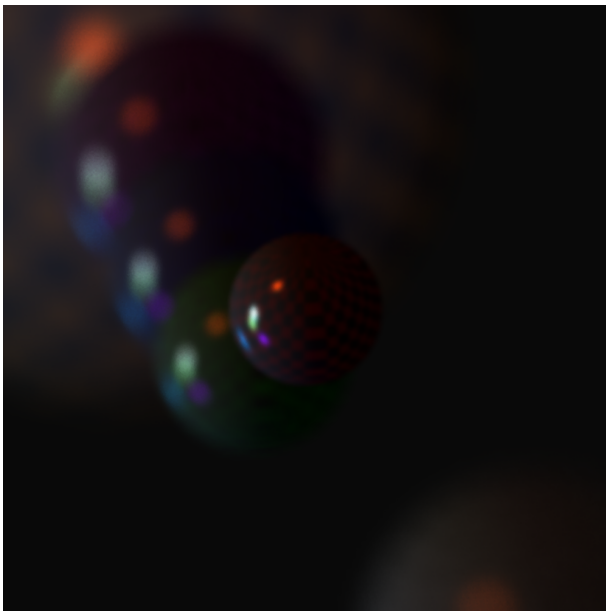


Figure 2.6: A simple synthetic scene consisting of spheres, rendered with distributed raytracing. The red sphere was focused, everything in front and beyond is blurred.

2.4 Filtering

Filtering can be used to remove aliasing from a function when sampling it (for example, by some low pass filter). Consider two functions, $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. Following

notation in [SM09], *convolution* corresponds to integrating over the product of both functions

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt \tag{2.12}$$

where g is the original function and f the applied filter. Note that during integration, g is evaluated at $x - t$ to center the filter f on x. The definition described above is easily adapted to work on discrete domains, i.e.

$$(a \star b)[i] = \sum_j a[j]b[i-j] \tag{2.13}$$

where a and b are discrete functions (interpreted as sequences of values), a corresponds to the continuous filter f. This is just the sum of samples taken around i, weighted with b. 2D function, such as images, can be filtered this way, too. In this case, both a and b have to be two-dimensional functions:

$$(a \star b)[i,j] = \sum_{i'} \sum_{j'} a[i',j']b[i-i'][j-j']. \tag{2.14}$$

Note that in practice, filter function are $\neq 0$ only on some finite range around 0. Therefore, the corresponding sums (and the integral in the continues version) of the convolution definition can be bounded, since all other values of b (g) are weighted with zero anyway. See [SM09] for more details on the theoretical properties of filters. Images are typically filtered by centering a *kernel* on every pixel in the input image, and weighting the neighboring pixels with the coefficients saved in it [Sze10].

2.4.1 Gaussian Filter

The 1D *Gaussian function* is defined as [YGV98]

$$g_{1D}(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}}. \tag{2.15}$$

The 2D version of this filter is obtained by multiplying two 1D Gaussian functions, each evaluated in their respective dimension, yielding [Sze10; YGV98]

$$g_{2D}(x,y) = g_{1D}(x) \cdot g_{1D}(y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{2.16}$$

The Gaussian filter is commonly used to produce smoothly blurred images. It can be used to remove aliasing such as moire patterns [SM09].

2.4.2 Laplace Filter

The Laplacian operator [Sze10; YGV98] is defined as

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{2.17}$$

where I is some 2D image. Since the second derivative reacts strongly to high frequencies, this filter can be interpreted as a high pass filter. Usually the *Laplacian of Gaussian* is applied, i.e. the image is first smoothed with a Gaussian filter, and subsequently the Laplace operator is applied, leading to a bandpass filter. Taking the Laplacian of an image can also be used in edge detection problems.

2.5 Wavelets

Wavelets can be used to decompose a function f into several basis functions which describe various levels of detail via their corresponding coefficients. f can be reconstructed by combining the basis functions multiplied with their respective coefficient [SDS95]. By neglecting coefficients (i.e. setting them to zero) which describe high-frequency components of f, the resulting reconstruction loses some information. However, this can be used when lossy compression is acceptable [TH10]. The wavelet transformation scheme is used in `JPEG2000` [CSE00].

2.5.1 Haar-Wavelets

A 1-dimensional image can be interpreted as a function defined on $[0, 1)$. According to [TH10], a vector space notation can be used to describe those functions: Let V^0 describes all images with only one pixel, therefore V^0 contains only constant functions, V^1 describes all images consisting of two pixels (i.e. two piecewise constant functions), and so on. Consequently, V^j describes all images constructed of 2^j pixels. Note that V^j divides $[0, 1)$ in equally sized intervals, each containing a piecewise function describing the corresponding pixel. V^j can be though of as a vector space, since every image of size 2^j can be expressed as a vector $v \in V^j$, with 2^j components. It holds that $V^0 \subset V^1 \subset ... \subset V^j$, since a piecewise function defined on some interval can be described by two piecewise function defined on the intervals obtained when splitting the larger interval in the middle.

The basis for V^j is constructed as follows [SDS95; TH10]:

$$\begin{aligned} \phi_i^j(x) &:= \phi(2^j \cdot x - i) \qquad \text{for } i = 0, ..., 2^j - 1 \text{ with} \\ \phi(x) &:= \begin{cases} 1 & \text{for } 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2.18}$$

The Haar-Wavelet basis is defined as [SDS95; TH10]:

$$\begin{aligned} \Psi_i^j(x) &:= \Psi(2^j \cdot x - i) \qquad \text{for } i = 0, ..., 2^j - 1 \text{ with} \\ \Psi(x) &:= \begin{cases} 1 & \text{for } 0 \leq x < 0.5 \\ -1 & \text{for } 0.5 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2.19}$$

These basis functions are a basis for the vector space W^j. Interestingly, the basis of V^j and the Haar-Wavelet basis for W^j can be used to describe all vectors in V^{j+1} [SDS95]. It is now possible to describe some image from V^j with basis functions from V^{j-1} and W^{j-1} with corresponding coefficients. To obtain the original image, the basis functions are multiplied by their coefficients, and linearly combined. Note that coefficients for V^{j-1} can be interpreted as an image with 2^{j-1} pixels, so this image can be expressed similarly via the basis's of V^{j-2} and W^{j-2}, indicating the recursive nature of the wavelet transformation.

The coefficients of V^j can be interpreted as describing the average of two pixels (*approximation*), while the coefficients for W^{j-1} represents the finer details in both pixels (also called *detail coefficients*) [SDS95]. In fact, images are wavelet-transformed in 1D by averaging neighboring pixels, and constructing the detail coefficients as the difference of the pixel from the average i.e. given two pixels $[\ x\ y\ ]$, the pixels are averaged to $a = (x + y)/2$, the detail coefficient is obtained as $d = (x - y)/2$. From $[\ a\ d\ ]$, x and y can be obtained via

$$a + d = (x + y)/2 + (x - y)/2 = x \tag{2.20}$$

and

$$a - d = (x + y)/2 - (x - y)/2 = y. \tag{2.21}$$

To wavelet-transform a 2D-image, two approaches exist: *standard decomposition* and *nonstandard decomposition* [SDS95]. In the first approach, one transforms all rows of the image until the desired depth of recursion is reached. Afterwards the same procedure is applied to the columns of the row-transformed image. The second approach consists of alternating between row and column transformation: First one transforms the rows of the input image. Then one uses this one-time row-transformed resulting image for column-wise transformation. These steps are repeated to the desired depth.

2.5.2 Compression

Applying wavelet-transformation to a 2D image yields an resulting image of the same size as the input [Sze10]. Also, this transformation is lossless [TH10]- applying the inverse transformations in an recursive manner will transform the image back to its original state. However in the case of coherence in the image data, detail coefficients could be zero or very small, since averaged pixels are very similar. This provides an opportunity for compression by introducing a cutoff value $\epsilon \geq 0$ [PS01; TH10] The simplest method consists of setting every (absolute) detail coefficient $< \epsilon$ to zero (*hard thresholding*). Clearly, some details are lost in this step- the inverse transform will only yield the averaged pixels if detail coefficients are zero. Depending on the selected ϵ, this could be almost visually indistinguishable from the original image. Then *quantization* can be performed, mapping values to values which require less bits, but are also less precise. In a subsequent step, a lossless compression method like *Huffman-encoding* or *run-length encoding* could be applied to the transformed and thresholded image to further reduce the image's size [TH10]. The last two steps are part of many lossy image compression algorithms.

2.6 Spatial Data Structures

Spatial data structures are a class of preprocessing schemes which allow fast access to spatially distributed data such as points or geometry [SM09].

2.6.1 kd-Trees

kd-Trees are a variant of *Binary Space Partitioning* Trees [PH10], suited for low dimensionality nearest neighbor search [ML09]. The data structure's expected memory requirements is in $\mathcal{O}(n)$, queries can be answered expectedly in $\mathcal{O}(log\ n)$, with n being the total number of points. The requirement for low dimensionally d of the points arises from the hidden constant factors which increases runtime of queries by at least 2^d [AMN+98]. Arya et al. propose to reduce this problem by approximating the optimal result, i.e. the returned nearest neighbor point of a query point is at most $1 + \epsilon$ times further away than the optimal neighbor.

The tree is constructed by splitting the input points in some dimension, and then recursively dividing the resulting halved point sets [ML09; PH10]. Note that this will lead to the splitting planes dividing the points being perpendicular to the split

dimension's axis. Figure 2.7 shows an example for a 2D point set. Several methods for determination of the split dimension exist, including randomized selection [ML09].

After construction of the data structure, it can be queried by traversing the tree.

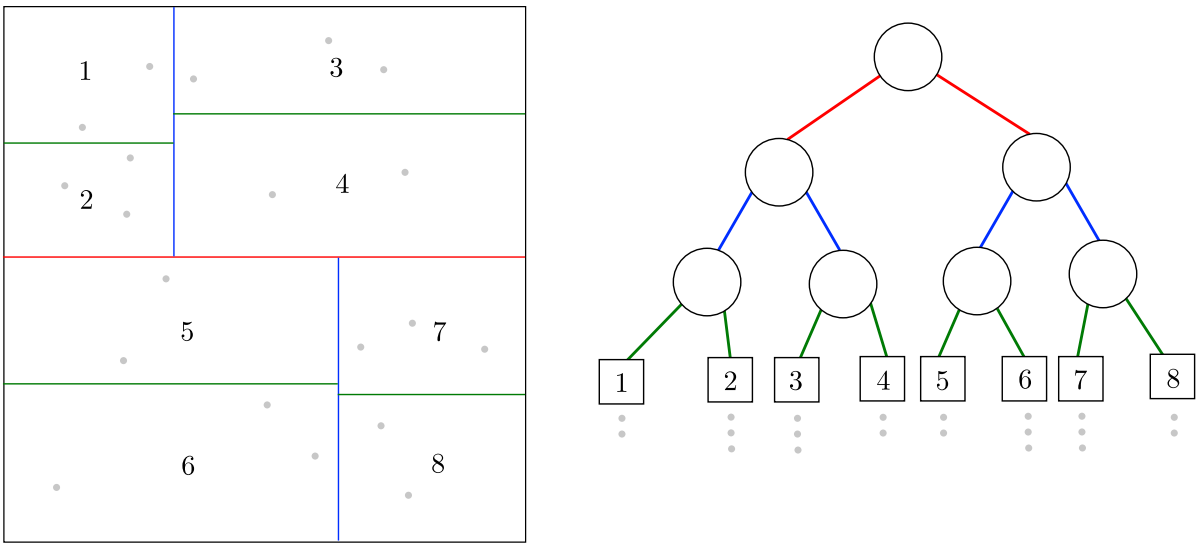


Figure 2.7: Depiction of a 2D point set on the left, and a possible space partitioning by the colored planes. On the right, the resulting kd-tree. The cell numbers correspond to the child nodes.

2.6.2 Octrees

Octrees [SSF05] are very similar to kd-trees, but they have two distinct differences: They only operate on 3-dimensional spaces, and spatial subdivision is done by regularly subdividing space in eight smaller cubes . This means, that a parent node has either eight children, and is therefore *internal*, or none, and therefore a leaf. Leaf nodes contain the data, similar to kd-trees.

3 Related Work

Numerous light field rendering approaches have been proposed in the past. Depending on the light field's size and the requirements on image quality, rendering can even be achieved at interactive frame rates. The following section describe various aspects of light rendering, as they were explored in recent times. See Section 2.1.2 for an introduction to capture and compression of light fields.

3.1 Light Field Datasets

Current light fields are usually not very large in comparison to the light fields used in this work. For example, the light fields found in the Standford Light Field Archive usually consist of about 300 images [Com]. For example, the *treasure chest* dataset consists of 289 images with an resolution of 1536×1280. Another publicly available light field database is [Wet]. Again, those light fields are small in size and consist of 25 to 49 synthetic images. The low number of views limits movements and aperture settings. Birklbauer et al. utilized light field data which is already multiple GB large, however the data still fits in random access memory of typical machines [BOB13]. The digital scan of Michelangelo's David statue done by Levoy et al. [LPC+00] consists of 7000 color images, which were compressed in a lossless fashion. This amount of data is already enough to not fit in main memory, however this dataset is still way smaller than the light fields used in this work, which may exceed TBs in size.

3.2 Rendering of Light Field Data

Levoy and Hanrahan implemented an interactive renderer for their light fields [LH96]. Since they used light slabs for parameterization, both quadrilateral defining the slab can be constructed from polygonal primitives such as triangles. Similar to texture mapping, the s, t coordinates as well as the u, v coordinates can be obtained by interpolation from the quadrilateral's corner points. Afterwards, the 4D light field may be queried with s, t, u, v to obtain the correct radiance of the desired rays. Levoy and Hanrahan just

used the nearest available sample points to interpolate the radiance via quadralinear interpolation.

Gortler et al. [GGSC96] proposed the use of texture mapping in a very similar fashion as Levoy and Hanrahan. In addition, they used a rough geometric estimate of the light field's geometry. For real world scenes, they estimated the geometry by a set of voxels engulfing objects in the scene. This geometric approximation can be used to select ray samples which are more likely to represent the object's reflected radiance for a desired ray.

Both rendering approaches described above assume the light field to be available in the form of light slabs. Buehler et al. introduced a rendering technique which works on light fields available as a stream of images [BBM+01].

These approaches let the user view the light field with a pinhole camera- depth of field effects were not considered.

Isaksen et al. [IMG00] proposed, in a addition to a dynamic reparameterization method depending on a user-selected focus plane, a synthetic aperture rendering technique. The st-plane contains regularly distributed *data cameras* indexed with (s, t). Each camera's pixel (and therefore the corresponding ray) can be address via (u, v). For some arbitrarily defined focus plane F, a ray parameterized by (s, t, u, v) can be parameterized with (s, t, f, g), with (f, g) being the ray intersection point on F, and (s, t) the index for the camera on th st-plane which recorded the ray. Rendering is done by intersecting a desired ray with the st-plane, say at (s_0, t_0), and determining the cameras on the st-plane located around the intersection point (s_0, t_0). Additionally, the focus plane F is intersected by the ray at (f_0, g_0). Then ray samples are taken from each camera (indexed via (s', t')) by sampling the image at the pixel corresponding to (s', t', f_0, g_0) and combined with an additional filter to the desired ray. Note that the range at which data cameras are regarded can be interpreted as the aperture size which determines the resulting depth of field effect. Isaksen et al. indicated that this method will keep the aperture parallel to the st-plane for the sake of interactive performance.

Birklbauer et al. [BOB13] presented an approach which enables interactive viewing of *gigaray* light fields, utilizing the rendering approach described by Isaksen et al. Light field data is cached in GPU-memory and replaced by new data according to the probability of future usage of already cached data.

The renderer already available for the dense data sets was implemented by Siedelmann [Sie15]. It simulates a synthetic aperture by reprojecting light field images to the user-selected focus plane, and averaging the resulting color values. With sufficiently many images, the synthesized image appears highly realistic, especially when the depth of field is small. The renderer could be interpreted as being semi-interactive, since users

are able to change camera parameters on the fly. The focus plane can be rotated freely, however the camera position is restricted to the light field's capture hemisphere.

4 System

This chapter contains a description of all applied methods which contribute to the final rendering system. Note that "light field camera" refers to a light fields' single captured image together with their extrinsic parameters. Strictly speaking, a single camera recorded the light field, but this is equivalent to multiple cameras recording the images at the same time. In principle, an entire light field of n images could have been recorded in a single go with n cameras.

4.1 Overview

Rendering of camera movements is accelerated if multiple nodes work on processing the data simultaneously. It is however necessary to extract information about the required data first- not all light field images have to be considered. This information can than be assigned to render nodes which independently process their relevant data.

The proposed system consists of two major phases: In the preprocessing phase, all relevant camera positions which will contribute with recorded radiance to the rendered sequence of images are collected and (optionally) additionally compressed in a lossy fashion. Collecting those cameras, as well as bounding the contributing region in image space will yield an implicitly given approximation of the envelope of contributing rays.

In the second phase, the images gathered in the first step are sequentially processed to obtain the synthetic images. Nodes always process a temporally connected section of the camera movement.

Nodes running the second phase will independently return their current state periodically, leading to a progressively improved result.

Users may stop the processing of individual nodes at any time, if their corresponding section was rendered in the sufficient quality. Figure 4.1 shows the system's concept.

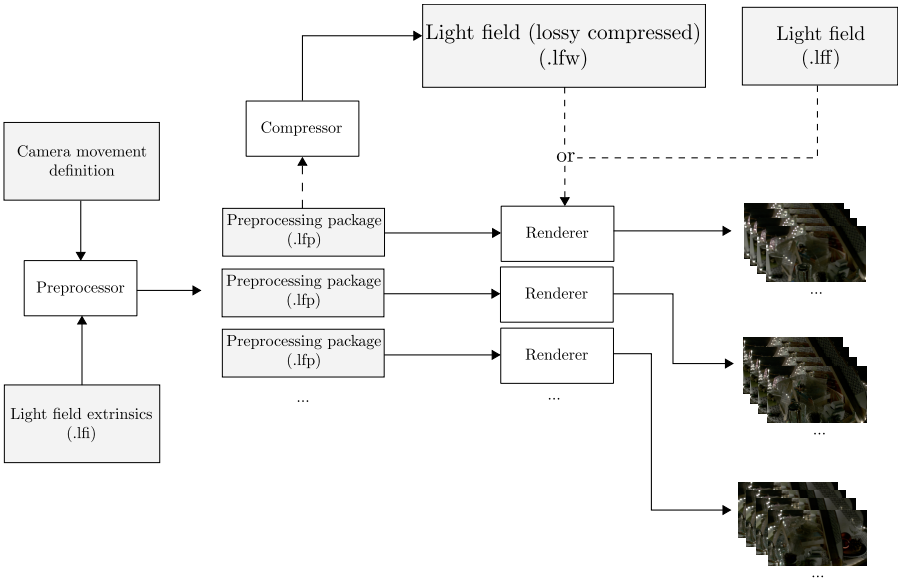


Figure 4.1: Overview of the proposed pipeline. The preprocessor selects the relevant light field images. Multiple packages are generated which serve as input for the renderer, together with a light field. This light field can be available in lossless or lossy format. Since the preprocessing package contains the relevant frames, they can be additionally compressed. This makes distribution of a partial light field possible. The renderer is then executed on multiple machines and will process the selected light field images. Periodically, the current state is returned.

4.2 Preprocessing

Recall that dense light field recordings require multiple TB of space and consist of potentially millions of images- integrating over all those images is infeasible. Most camera movements require only a small subset of light field images to render new views. These images have to be determined first. This step is intended to be fast, such that it can easily be done on a single computer. From this preprocessing step it is possible to assign rendered cameras and the necessary light field images that have to be processes to individual nodes.

4.2.1 Relevant Camera Extraction

The following subsection describe how light field cameras and therefore relevant images in the dataset can be obtained. Additionally, the approach described attempts to bound the rays that have to be checked for contribution for every synthetic camera in the

sequence. This is done by constructing an axis-aligned bounding rectangle in image space which bounds the pixels that contribute with their recorded radiance.

Approximative Finite Camera Frustum

Consider rendering pinhole cameras. When rendering such a pinhole camera C in traditional raytracing, for every pixel in the image plane, a ray r can be constructed. If r passes through the projection center of a second pinhole camera C', this camera might have recorded a ray r' very close to the requested ray, since r passes through its center point, (e.g. in Figure 4.2). If such a ray was indeed recorded, by looking up the radiance of ray r' in the image of C', one finds the valid radiance value for a pixel in the image of C. By constructing a frustum consisting of planes for C, relevant cameras can be gathered by checking if their projection center resides inside the frustum.

The approach described could be used to select relevant cameras of a light field dataset.

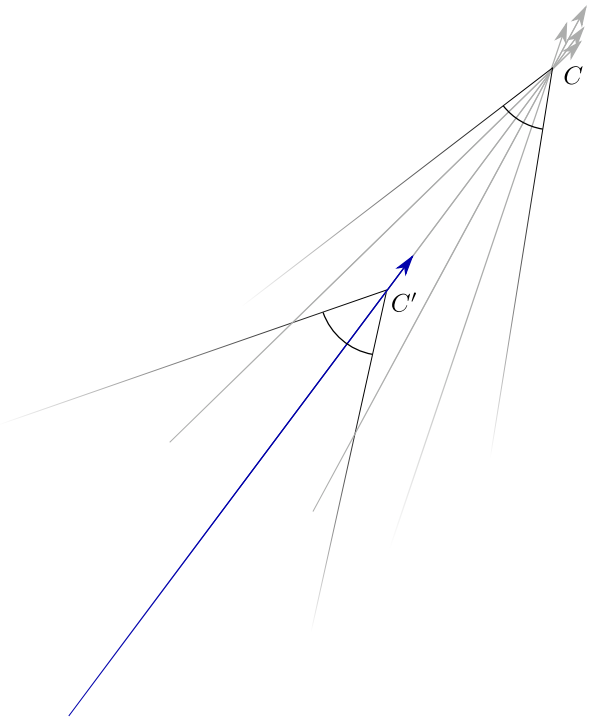


Figure 4.2: Reconstruction of a desired ray from another camera. The gray rays depict rays which are needed to construct the view defined by the upper camera C. The blue ray depicts some correct ray from the lower camera C' which should be used by C.

The rendering system described in this work should be capable of supporting cameras with finite aperture sizes. Therefore, a modified approach is necessary to cover relevant

cameras. For this, a modified frustum is constructed, which will contain all points that lie on rays which are potentially contributing to the rendered image. If a required ray was recorded by a light field camera, its position must lie on the ray itself. Therefore, if all points on the ray are covered by the frustum, all contributing light field cameras are covered, too.

Consider the lens l of the rendered camera C, modeled as a circle, and the focus plane f_c, as well as the image plane i_c. The sensor plate s_c is modeled as a rectangle, and placed at i_c. Now the four rays which originate at the corners of s_c and pass through the center of the lens are constructed. Let $t_k,\ k \in \{1, 2, 3, 4\}$ be the intersection points of the four rays with f_c. Rays that are contributing to the rendered image must intersect the rectangle defined by corner points t_1, t_2, t_3, t_4. A ray might hit such a point p, but will only be refracted by the lens l and hit the sensor plate s_c if this ray also passes through the lens l. Therefore rays must intersect both the lens and the focus plane rectangle if they contribute to the image. A frustum can now be constructed such that it covers every point on every possible contributing ray. To achieve this, a widened frustum is constructed from four planes. Figure 4.3 shows the construction in 2D. The axis-aligned bounding rectangle r_l of the lens l is constructed. A frustum plane must be constructed such that it contains the ray with largest angle to the optical axis (dotted blue rays in Figure 4.3). Then this plane is shifted by the diameter of the lens l (blue rays, shift indicated by gray arrows). Note that this frustum will cover all points lying on the most "extreme" rays relative to the optical axis of the rendered camera, i.e. the rays grazing one side of the focus plane, and similarly striking through the edge of the bounding rectangle r_l on the opposite side. However, the constructed frustum overestimates the set of contributing rays slightly. Figure 4.4 shows how cameras are collected and discarded. Note that the frustum should also be constructed for the backside of the rendered camera, as light field cameras might be positioned behind the rendered camera.

Frustum-based Octree Traversal

With the frustum described above, relevant cameras of light field datasets can be collected. To accelerate the collection process, light field camera positions are inserted into an octree. It is now possible to intersect the frustum with the octree's voxels, beginning with the tree's root. If the corresponding voxel of an inner node is intersected or completely contained in the query frustum, this node has to be further traversed. Otherwise traversal can be stopped for this node. If a leaf is intersected or inside the frustum, camera positions being stored at this leaf must be individually checked to determine if they are inside the frustum. Note that this will not work very well if the

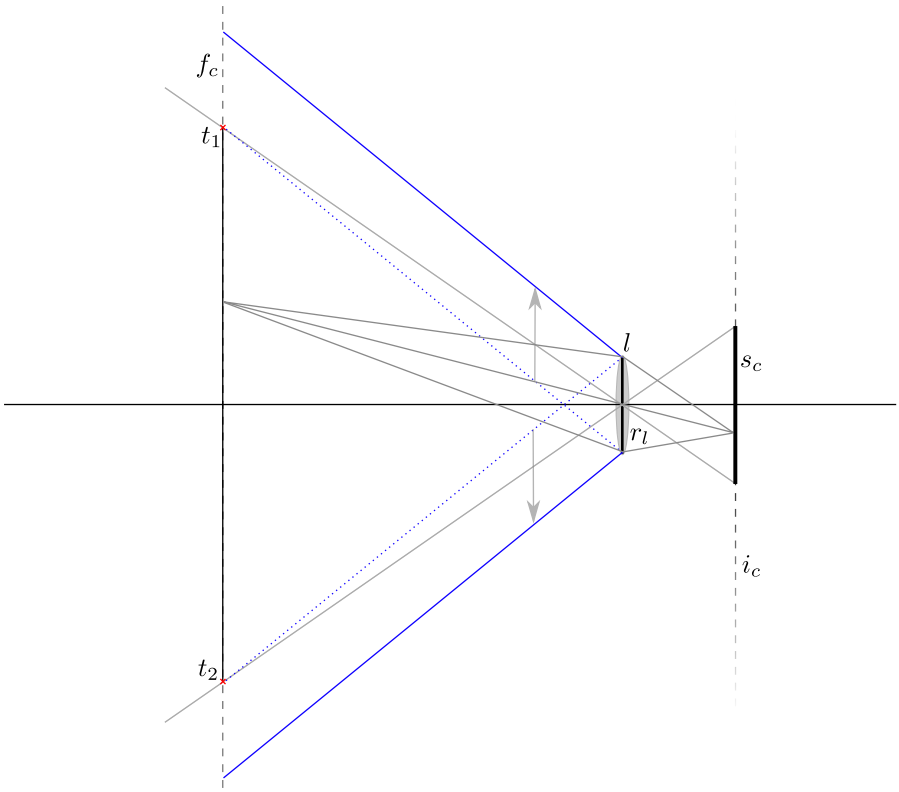


Figure 4.3: Depiction of the frustum construction in 2D. The gray lines correspond to the pinhole camera's equivalent frustum. The dotted blue lines show the frustum planes before shifting. The blue lines depict the final frustum's planes.

frustum contains large parts of the root voxel, or engulfs it completely. In this case, all nodes might have to be traversed.

Another issue is, that light field cameras looking towards the desired camera are collected too, if their position lies inside the frustum. However those cameras can be easily discarded by only considering light field cameras where the difference between the light field and desired camera's forward vector does not exceed a certain angle.

Extraction of Relevant Area in Image Space

Light field cameras which likely recorded contributing rays can now be retrieved. However, it is possible that those cameras are not actually contributing with any rays. In addition to the steps described above, the image space area of light field cameras which implicitly estimates the set of rays can be computed.

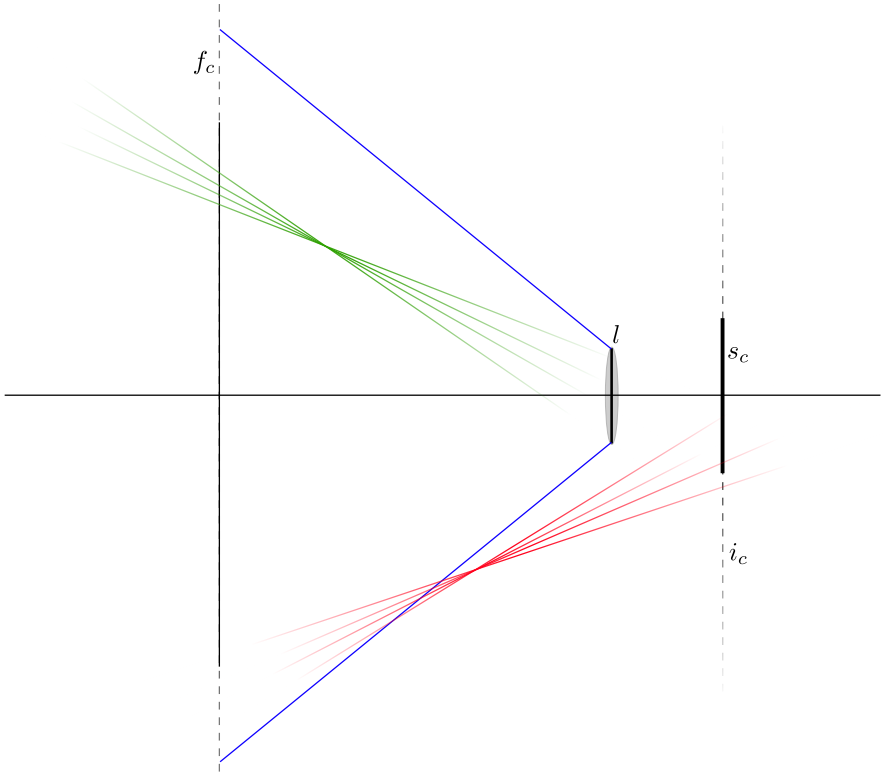


Figure 4.4: Depiction of the frustum. The green rays belong to a relevant camera. The red rays belong to a camera which will not contribute with radiance information. Some of those rays intersect the focus plane rectangle, but not the lens.

Consider a light field camera C_{lf} and a camera C_{fa} which has a finitely sized aperture. Again, let r_l be the bounding rectangle of C_{fa}'s lens, and let r_f be the bounding rectangle on the focus plane, obtained as described in Section 4.2.1. Both r_l and r_f are projected into the image plane of C_{lf}. Rays striking through one of the rectangles will have their intersected pixel covered by the corresponding projected rectangle. Relevant rays must intersect both rectangles if they contribute to the rendered image. Therefore, by computing the overlapping area R of both projected rectangles, one obtains the area of pixels describing contributing rays.

If there is no overlap, C_{lf} can be discarded, since there is no relevant ray hitting both rectangles.

Note that it is possible to approximate the lens geometry by not just an bounding rectangle, but other (convex) polygons. Different aperture shapes can be modeled and considered this way, see Section 4.3.3.

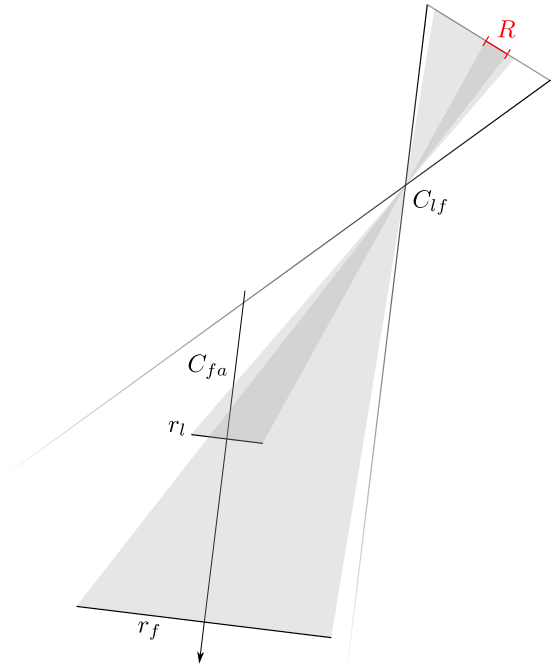


Figure 4.5: Projection of the lens and focus plane into the light field camera's image space. Note that relevant rays must pass through both rectangles, otherwise they won't contribute with radiance. The relevant region R in image space is marked in red.

The complete preprocessing process determines relevant image space areas for a sequence of cameras to be rendered. Since it is now known which light field camera is contributing to which rendered image, the sequence in which frames are processed by the renderer can be defined beforehand. A simple strategy is just shuffling the collected frames in an uniform fashion, to obtain an more even result among all rendered frames, even before all selected frames have been processed. Alternatively, one might reorder the sequence such that sensor plates of rendered cameras are hit with rays equally densely, by defining the ordering in a round-robin fashion. Preferring light field cameras with larger covered areas is possible, but might lead to biased results until all frames are processed by the renderer. This happens because in dense light fields, many cameras vary only slightly in their extrinsic parameters as the recording positions are packed very densely. Many very similar images are processed first, leading to less uniform depth of field effects until all frames are processed.

4.2.2 2D-Wavelet Compression

To reduce the amount of needed data, one may just discard a subset of required frames. Often this is feasible, decent rendering results are still possible. It is also possible to compress the individual frames in as lossy fashion, to further reduce the amount of data shipped to nodes. The light field's images are available as raw Bayer Pattern images. Cameras typically record with color filters placed above the individual sensor elements, resulting in the pattern in the raw image [Sze10]. Since the human eye is most sensitive to green color, green filters cover twice as many sensor elements as the other two channels.

A 2D Wavelet compression scheme as described in Section 2.5 was implemented. The raw light field image's raw channels are reordered to different areas in the image, to obtain better correspondence between neighboring pixels, see Figure 4.6. Afterwards, they are Wavelet-transformed in both image dimensions (Figure 4.7) and then additionally compressed with a standard lossless compression scheme. Rendering nodes then uncompress the compressed frames and reconstruct the image by applying the inverse wavelet transformation. The original raw image is obtained by reordering the pixels to the original pattern. Peter and Straßer proposed mapping the range determined by the minimum and maximum coefficient to a single byte [PS01], essentially saving coefficients as fixed floating point numbers limited to some range. A similar approach was used for every compressed image.

After the relevant light field frames have been determined, only those images are required for rendering. Those frames can be compressed and shipped to nodes. This makes distribution of partial light fields possible.

While compressing the light field only in two dimensions is not ideal, a compression ratio of 15 or more is achievable without too much loss in image quality.

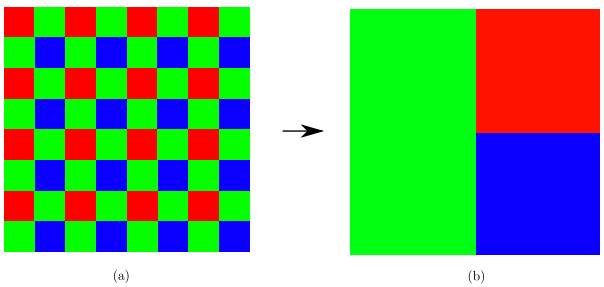


Figure 4.6: Reordering of the Bayer Pattern. (a) shows the initial pattern, (b) the reordered pattern.

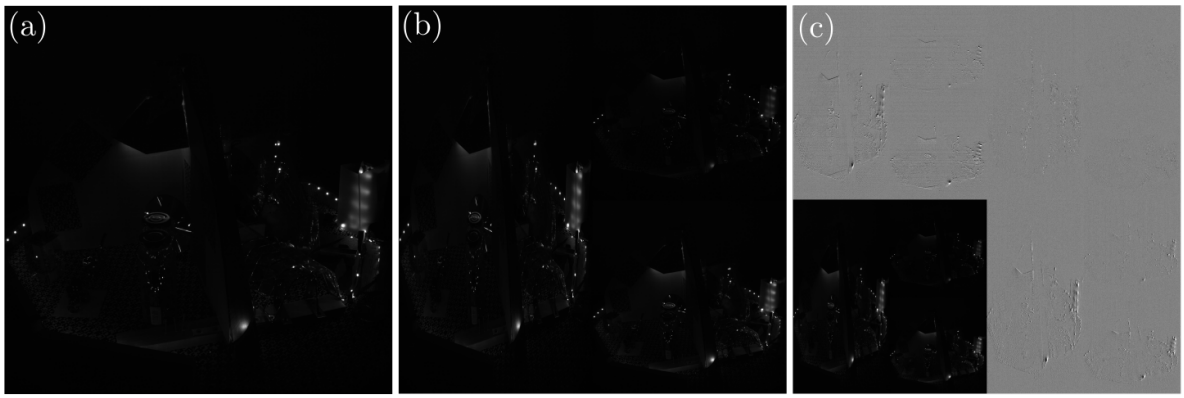


Figure 4.7: A raw light field frame is shown in (a). The reordered image is shown in (b). The wavelet transformed image is shown in (c). One transformation step was applied in each image dimension. No thresholding was applied. The averaged image is located in the lower left, the detail coefficients occupy the rest of the image. To increase perception of the detail coefficients, contrast and brightness were slightly modified. Large areas contain the same value, indicating coherence in the data.

4.3 Rendering of Light Field Data

Preprocessing returns a set of cameras, as well as the image space area of relevant pixels for each rendered camera in the sequence. Depending on the camera movement and parameters, this approach has one significant advantage: A single frame often contains rays contributing to multiple rendered cameras at once. In the ideal case, a camera with large aperture size and slow movement speed will lead to all rays recorded by a light field camera contributing in some sequence frame. The two implemented approaches for rendering will be described in the following sections. The *ray collection* approach considers individual light field rays and places their radiance information in the rendered cameras sensor plates. *Sparse on the fly rebinning* serves as a backup mechanism to the first approach. A sparse representation of L is build from loaded light field data and can be queried for interpolated ray information. Note that both approaches use the camera selection process described in Section 4.2.1 to obtain a sequence of frames to process.

4.3.1 Ray Collection Approach

In this approach, light field frames are processed sequentially. The described method was chosen since it is not feasible to keep all required light field data in memory. Instead, light field images are fetched and discarded after their contributing radiance values have

been extracted. For each rendered camera, the rays of the relevant area in image space are constructed.

Let C be a pinhole camera, e.g. a camera from the light field dataset. Points on some ray r are all projected to the image space point $p = (x, y)$. To obtain r, $(x, y, 1)$ is multiplied by the inverse intrinsic matrix, i.e.:

$$p_{camera} = K^{-1} \cdot p \tag{4.1}$$

which will yield a point on r in camera coordinates. By multiplying the inverse extrinsic matrix, this point is transformed to world space:

$$\tilde{p}_{world} = \begin{pmatrix} R & t \\ 0^T & 1 \end{pmatrix}^{-1} \cdot \tilde{p}_{camera} \tag{4.2}$$

where $\tilde{p}_{camera}$ is the homogeneous version of p_{camera}, augmented with an additional component being 1. Now the world space point is obtained from the first three components of $\tilde{p}_{world}$. By casting a ray from the camera's position towards the obtained point (or the opposite direction), the corresponding ray is constructed.

Every ray captured by light field cameras and considered relevant is constructed as described above. Then those rays are refracted by the cameras lens of a rendered camera, and collected in a *synthetic sensor plate*, mimicking a real camera's sensor plate (however, without being multiplexed like the Bayer pattern). Be S a matrix of size $w \times h \times 6$ representing such a sensor plate, with w and h being the width and height in pixels. Since the light field dataset's frames are available in raw format as they were recorded by the camera, sensor plates have to be present for each color channel. Rays therefore only ever contribute with radiance in a single channel. Be $p = (r_{sum}, g_{sum}, b_{sum}, r_{cnt}, g_{cnt}, b_{cnt})^T$ the vector representing the data collected in a single sensor element of S. Whenever a ray hits p, the radiance carried by this ray is added to the corresponding sum and the counter incremented.

Refracting a ray is straightforward: One just computes the intersection with the rendered camera's focus plane. Then one casts a ray from this point towards the center of the lens. Finally the intersection of this ray and the image plane can be computed, resulting in the sensor element which is hit. This corresponds to the approach described in Section 2.3.2, however in the opposite direction. An alternative approach is treating the rendered camera as a pinhole camera (with the aperture at the center of the lens) and projecting the focus planes' intersection point into the cameras' image space, which will similarly determine the corresponding pixel. To discard rays which do not hit the lens of the rendered camera, the intersection between the lens' plane and the original rays are computed. If the distance intersection/lens-center is larger than the radius of the lens, the ray has to be discarded as it does not hit and pass through the lens. This corresponds

to the thin lens model's properties for circular lenses. The final image can be computed by averaging the collected data, i.e. $(r, g, b)^T = (r_{sum}/r_{cnt}, g_{sum}/g_{cnt}, b_{sum}/b_{cnt})^T$.

This however, might lead to noisy images, especially if not enough rays were collected over the course of the process. To obtain a smoother image, inverse distance weighting [She68] can be used to take radiance collected in surrounding pixels into account. This weighting was chosen because of the rapid falloff with increasing distance of the weight $1/d$, where d is the distance to another pixel, keeping the amount of introduced blur low. A positive side effect is that interpolation of collected radiance from the neighboring pixels becomes possible if a sensor element did not collect any radiance in its corresponding channel.

As more and more frames of the light field are processed, the data collected in the synthetic sensor plates converges towards the optimum image result achievable without interpolation between rays. Note that this approach is very accurate.

This approach works best when rendering a camera movement close to the original light field camera positions, and/or with large aperture sizes. Only the rays stored in the dataset are utilized, and used to compute the desired camera movement's images. More diverse camera movements require more light field images to produce acceptable results. Another limitation is that rendered cameras require a certain minimum aperture size for this approach to work well, otherwise only a small number of rays is contributing to the rendered images with every processed light field frame. Consider rendering a single camera with an aperture size smaller than the distance between the camera positions of the dataset. If such a camera is placed between the original recording positions, no ray can possibly hit the aperture, since no ray was recorded at that position. The approach described below attempts to circumvent this problem by augmenting the rendered images with interpolated rays. Tilted focus planes can be rendered too, one just modifies the focus plane's normal vector and intersects a light field ray with this plane.

4.3.2 Sparse On The Fly Rebinning

The second approach copes better with less restricted camera movements, and works with small aperture sizes. This method attempts to rebin high frequency regions of $L(s, t, u, v)$, while keeping low frequency areas more sparse. Levoy and Hanrahan stated that the light field's rays should be distributed uniformly, since every ray is equally likely to be used to construct a new view [LH96]. This assumption is dropped in favor of better memory efficiency. Hash maps are utilized to keep memory requirements acceptable-instead of initializing a potentially large $4D$-array, a subset of discretized rays is kept in the hash map. Again the preprocessing as done before is applied in the first step.

Afterwards a *primary slab* p_s is constructed, consisting of two planes. The slab's st-plane is positioned roughly among the camera positions of the movement path, facing roughly in the same direction as the rendered cameras. This slab will be used to parameterize $4D$-Cubes. Light field rays are parameterized against this slab as well. Consider the st-plane, and the point o_{st} which is used to define this plane, together with its normal vector. Two axes being perpendicular to each other can be placed in the plane. It is now easily possible to discretize the plane in equally long segments of length r in both dimension, resulting in a $2D$ grid structure. The same is applied to the uv-plane. Rays intersecting the two slabs at $(s,t,u,v)^T$ are then assigned the indices $(s_d,t_d,u_d,v_d)^T \in \mathbb{Z}^4$, by determining the two grids the ray traverses when it intersects the two planes, by computing:

$$\begin{pmatrix} s_d \\ t_d \\ u_d \\ v_d \end{pmatrix} = \begin{pmatrix} \lfloor s/r \rfloor \\ \lfloor t/r \rfloor \\ \lfloor u/r \rfloor \\ \lfloor v/r \rfloor \end{pmatrix}. \tag{4.3}$$

Rays whose intersection points fall into the same grid rectangles are discretized to the same index. r therefore controls how finely the resulting discretization is. This is similar to the discretization applied by Gortler et al. [GGSC96]. By parameterizing a subset of the rays required for rendering a camera, a 4-dimensional bounding box can be computed, which can serve as a rough estimate of its ray envelope parameterized by a single slab. However this approach could potentially cover a large amount of rays not required if a camera movement contains sharp changes in view direction, position or aperture size. As an example for such a case, just consider a camera path parallel to the slab, with changing aperture size, and resulting smaller frustums. A possible solution for a more fine-grained approach would be computing bounding boxes for subsequent frames of the camera movement, each with their own slab.

The method proposed here is utilizing a rough discretization to build a set of 4-dimensional hypercubes: If a ray is to be covered, it is parameterized against the slab and discretized. This will yield the indices of the intersected grids on their respective planes, and therefore a 4D cube. The cube's indices are then hashed and saved into a hash map. By constructing the corresponding key for a desired (discretized) ray, the hash map can be queried for it. It is now possible to associate information like radiance with a discrete ray, while keeping memory requirements low by constraining the map's number of elements.

Rebinning is done by parameterizing the rays of light field images. Since there is a high level of coherence in the light field L, not every ray in L is mandatory to construct a new view. Keeping every ray recorded by every light field loaded by the renderer would easily overflow available memory on nodes. However, high frequency sections of L should be represented by a denser collection of rays. These areas in the image are more

interesting for rebinning, low frequencies areas can be interpolated more easily. This requirement can be obtained by some common image processing techniques combined with probabilistically selection of rays.

First, the light field image I_{lf} is smoothed by a Gaussian filter, which will remove noise in the image. Then the Laplacian operator is applied to the smoothed image, yielding the *Laplacian of Gaussian* of the image. Let I_{LoG} be that image. High (absolute) values in I_{LoG} are an indicator for high frequencies in I_{lf}. Finally, the *probability map* I_p is computed from I_{LoG} by interpreting pixels as probability thresholds. Every pixel in I_{LoG} is mapped to I_p by

$$I_p(x,y) = \begin{cases} min(1, |I_{LoG}(x,y)| \cdot \alpha + \xi) & \text{if } |I_{LoG}(x,y)| \geq \beta \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where α scales the absolute values of I_{LoG} and β applies some threshold which has to be passed. $\xi \in [0,1)$ controls the minimum expected amount of rebinned rays per image. Ideally, ξ and β should be zero, but both parameters offer some flexibility since light field data might be compromised by noise. The probability map I_p now directly controls the likeliness of rebinning the recorded rays, i.e. for every pixel, a random number is drawn uniformly from $[0,1]$, the ray is rebinned if this number is smaller than the value in the probability map at this pixel. If the (discretized) ray is already present in the map, the radiance values are averaged. Note that applying another Gaussian filter on I_p will smooth out the probability map. Figure 4.8 shows a depiction of the process.

Note that high frequency regions in 2D slices of L usually correspond to high frequency regions in L itself. However, the opposite direction is not always true, e.g. consider a highly reflective material in the scene. Changing the viewing direction only slightly could lead to strong changes in observed reflected radiance, something that is not apparent in single light field images.

To further reduce the amount of rays in the hash map, a *coverage map* can be computed. A second hash map is initialized with the same primary slab, but with discretization interval $> r$. For all cameras, rays are constructed by spawning them originating on the lens, as it is done in distributed raytracing. These rays are parameterized against the primary slab, and inserted into the hash map. Assuming the amount of rays constructed this way is high enough, the hash map will contain the hypercubes which bound relevant rays for the rendered cameras. Parameterized rays not covered by the map are not relevant for the rendered cameras. It is now possible to restrict rebinned rays to those covered by the coverage map, further reducing memory requirements.

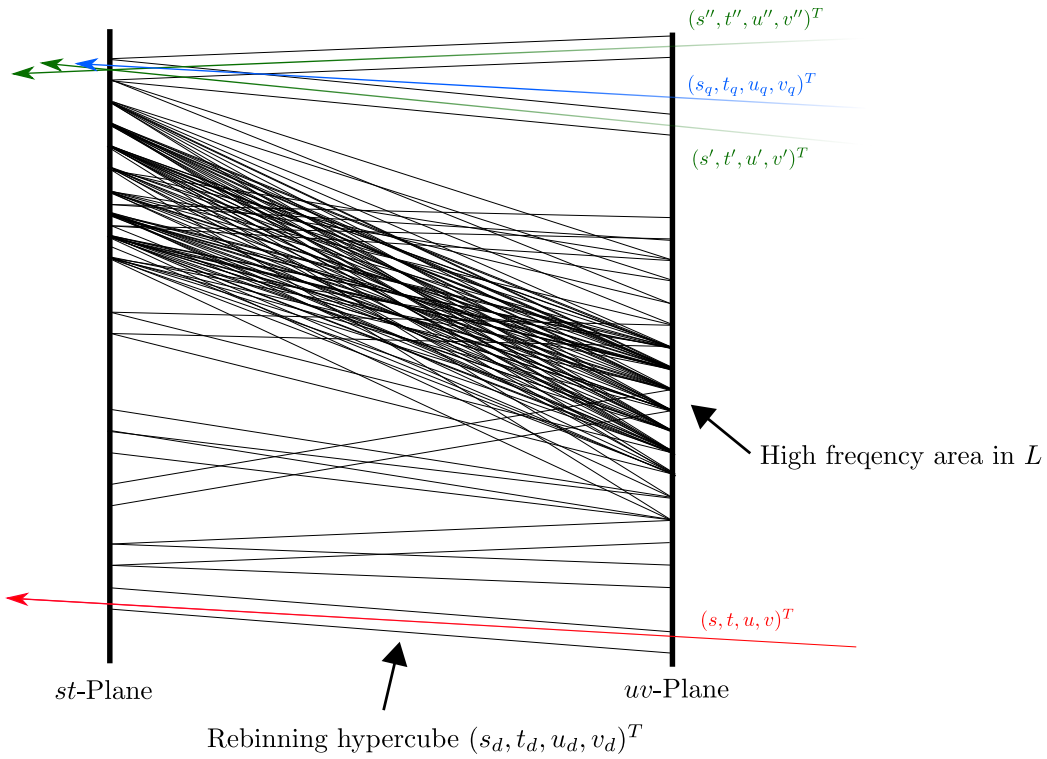


Figure 4.8: Side view of the sparse rebinning process. The st- and uv-plane define the light slab. The red ray $(s, t, u, v)^T$ was selected for rebinning and is placed in its appropriate hypercube $(s_d, t_d, u_d, v_d)^T$. As depicted, high frequency areas in L are rebinned more densely. Desired rays are interpolated from their nearest neighbors, e.g. the blue query ray $(s_q, t_q, u_q, v_q)^T$ is combined from the weighted green rays $(s', t', u', v')^T$ and $(s'', t'', u'', v'')^T$.

Rendering Phase

The hash map could be queried for discretized rays, however depending on r, this will most likely not lead to sufficient results: Lots of queries will fail since ray distribution is sparse. Instead, a number of images of the dataset are processed as described above. Periodically, or when the amount of hypercubes in the hash map reaches some threshold, a 4D kd-Tree is build, containing the center points (And therefore centered rays) of all hypercubes present in the map. An arbitrary ray can be parameterized and the kd-tree queried with a nearest neighbor or radius search. Similar to Wender et al. [WIG+15], who also used a (2D) kd-tree to represent their ray collection, 2D-Gaussian weighting is applied in each plane:

$$w_i = \frac{1}{4\pi^2\sigma_{st}^2\sigma_{uv}^2} e^{-\frac{(s-s')^2+(t-t')^2}{2\sigma_{st}^2}} \cdot e^{-\frac{(u-u')^2+(v-v')^2}{2\sigma_{uv}^2}} \tag{4.5}$$

where s, t, u, v denote the query ray's parameterization and s', t', u', v' the parameterization of a ray i which is used for interpolation. Note that in [WIG+15], the σ controlling the weighting of the front plane variables was directly used to control the size of the aperture. This is not the case in this work, the aperture is sampled by ray spawning. A 4D weighting of some kind (e.g. also Gaussian) could be used too, but the weighting described in Equation 4.5 gives greater control over the weighting. Usually, the sample spacing on the st-plane is more dense, if the camera movement is moving close to the light field capture positions, therefore σ_{st} could be chosen smaller than σ_{uv}.

The resulting interpolated radiance is determined by summing up all radiances $L_i = L(s', t', u', v')$ multiplied by their respective weights and normalizing by all n weights, with n being the amount of rays to interpolate over:

$$\tilde{L}(s, tu, v) = \sum_{i=1}^{n} \frac{w_i \cdot L_i}{\sum_{i=1}^{n} w_i}. \tag{4.6}$$

Images are then synthesized by applying distributed raytracing. Rays are spawned on the lens, parameterized against the slabs, and used as a 4D query point for the kd-tree. Since the rendered camera moves between subsequent frames, reparameterization similar to [IMG00] can be applied if the current camera plane (defined by position and forward vector) and focus plane differ too much from the current reparameterization slab. Isaken et al. directly used this as a means to render depth of field effects. It was found that applying reparameterization tends to help with ray selection, detailed objects are imaged more precisely and less smoothed. However this may not be desired in all cases, some artifacts in the form of "jitter" may come apparent during camera movement, which is harder to detect in still images.

Since four dimensions can still be qualified as low-dimensional, querying such a kd-tree is somewhat efficient, but performance degrades as more rays are present in the collection. On the other hand, interpolation of ray radiance is straightforward, which makes applying this approach possible directly from light field images.

Note that rendering images produced by pinhole cameras becomes possible by this approach, unlike the first described technique.

4.3.3 Combining both Approaches

The first approach can be interpreted as being *passive*: Synthetic camera sensors collect radiance information as more and more light field images are processed. The second approach attempts to build a sparse section of $L(s, t, u, v)$ by considering the desired rays of a camera path. The first approach works well with large apertures, since those

are able to collect more rays of the dataset, on the other hand, time until a meaningful result is obtained may be long if small apertures sizes are selected. It is possible to run both approaches in parallel: Images are fetched from the dataset, the rendered cameras collect the radiance of rays, and simultaneously build the sparsely rebinned and constraint version of $L(s, t, u, v)$. Periodically, a dump of the current state of the synthetic sensor plates can be done. Beforehand, the kd-tree containing rebinned rays is build and can be queried to augment the result with additional rays. If a sensor element has collected only a insufficiently amount of rays, additional rays can be constructed and interpolated by spawning them on the camera's lens. Then the kd-tree can be queried and the result combined with the collected rays. A possible optimization is the removal of rebinned rays due to them not being used in any query anymore, by keeping an array of flags indexed by the hash map's rays. Rays are flagged if they were used in some interpolation query. After all ray queries are done, those rays which were not flagged are removed. During the dumping stage, rays may be used for interpolation to augment the current result. Since it is likely that sensors collect a sufficient amount of ray samples, the need for such queries decreases. Eventually, a large subset of all rendered pixels could have collected enough rays from the dataset images, which means that the collection of rebinned rays can be reduced. Similarly, the coverage map can be shrunken.

This has positive two effects:

- Rays which are deemed suitable for interpolation are never removed until more desirable rays are present, since they were queried at some point, and will be in the future (with high probability).
- As the coverage map containing hypercubes is reduced, the remaining ones can be filled more densely, since memory gets freed, thus increasing the amount of available rays for interpolation.

Both approaches attempt to complement each other, by keeping memory requirements low while increasing image quality of results.

Simulating Aperture Shapes

By computing the distance between the intersection point of a ray hitting the lens plane and the lens' center point, rays which do not pass through a circular-shaped lens can be discarded easily if the distance is greater than the lens' radius. This will result in a circular bokeh effect. Aperture stops block light rays from hitting the lens, and will result in a bokeh effect shaped just as the aperture stop [WZHX13]. In the light field rendering system, aperture stops modeled as regular polygons can be used to create similar effects. Rays intersecting the lens plane are blocked if they do not lie inside the

polygon. OpenCV's *pointPolygonTest* function is used for this check. A small performance optimization is employed to cut done on the runtime of this check: Two bounding circles are created, one bounding the polygon's outside, and one bounding the polygon's inside area. if a ray intersection point lies outside the outer circle, it can be discarded instantly. Similarly, if it lies inside the inner circle, the ray will pass through. The polygon test has to be done only in the area between the outer and inner circle. By rotating the polygon depending on the aperture size, the effect of closing aperture blades can be modeled.

This setup can be used for both approaches described in this work, but the approach seen in Section 4.3.2 requires the selection of uniformly selected points on the lens, such that query rays can be spawned. To obtain such a point (x, y) on the unit disk, it can be chosen by selecting r uniformly at random from $[0, 1]$ and θ from $[0, 2\pi)$. Then, the coordinates are computed as [BEL+06; Wei]

$$x = \sqrt{r}\ cos\ \theta,\ y = \sqrt{r}\ sin\ \theta. \tag{4.7}$$

4.4 Implementation

The system was implemented in C++, utilizing some of the code developed by Siedelmann [Sie15]. Eigen3 [Eig16] is used for vector arithmetic. Google's sparsehash [Goo10] hash map implementation is used for the sparse rebinning approach described in Section 4.3.2, as well as the FLANN [ML14] kd-tree implementation which provides the means to construct and query such trees. The implementation makes heavy use of the OpenCV [CV15] library for image processing. The Squash [Squ15] library is used for compression of the wavelet-transformed images. Polygon clipping used in the preprocessing step to compute the relevant image space region as described in Section 4.2.1 is provided by the Clipper [Joh14] library.

Parallelization of the ray collection approach is straightforward by assigning sections of the rendered camera movement to individual threads, which all work on the same light field frame. Since threads do not work on the same camera movement frame simultaneously, data races are not possible, and no synchronization is necessary. A set of loader threads constantly buffers light field frames into a queue, which is then accessed by the renderer. This ensures that there is no bottleneck induced by IO-operations. In case of long camera sequences being worked on by the renderer, this works well since processing a light field frame for all rendered camera positions usually takes longer then loading a single light field frame, which will keep the buffer queue full at all times.

4.5 Software

This section contains information about the produced software. The naming scheme (**lf...**) from Siedelmann's work was kept to keep naming consistent. The main programs are **lfpreprocessor**, **lfcompressor** and **lfrenderer**. Auxiliary tools are **lfcammvminfo**, **lfvisualizer** and the **lfrendertools** library. **lfcammvminfo** can be used to display information about individual frames of the camera movement definition, e.g. current aperture size, position of the camera etc. **lfvisualizer** outputs light field camera positions, rendered camera positions and collected light field cameras into a plain-text format which can be piped into a python script providing some interactive visualizations. **lfexample** produces an exemplary camera movement definition with the camera being initially placed at a given light field camera's position. **lfrendertools** contains most utility classes which are used by the other programs. This structure was chosen such that development of new tools based on already present code can be done more easily.

4.5.1 Camera Movement Definition

The camera movement is defined inside a **.yaml** [Yaml02] file. It is possible to set initial position, lookat position, inital speed, as well as the camera and lens parameters. The movement is given by acceleration periods starting at given time steps. Aperture and focus distance changes are given in a similar fashion. Additionally, the focus plane can be tilted vertically and horizontally.

4.5.2 Preprocessor

lfpreprocessor extracts the needed frames from the light field as seen in Section 4.2.1. Necessary input are light field calibration data (camera positions and rotation, as well as the intrinsic matrix) and the camera movement definition. The given camera movement is split in equally sized packets, which will serve as the input of the renderer. The output will consist of a list of frames and their bounded region in image space for every rendered camera. Note that knowledge about the renderer's main memory is necessary beforehand, since every camera movement frame has to be initialized.

4.5.3 Compressor

lfcompressor also takes the preprocessor's output, as well as the light field as input, and compresses all light field frames contained in it via wavelet compression, see Section 2.5

and Section 4.2.2. Additionally, **lfcompressor** is also able to compress the entire light field.

4.5.4 Renderer

lfrenderer accepts the preprocessing packages, as well as the compressed frames or the original light field. All approaches described above, include the combined method, can be used as render modes. Periodically, the renderer will dump its current state.

5 Results and Discussion

This chapter contains the results obtained by the renderer. All rendering test were done on four machines with an i5-6500 with 3.20 GHz. Table 5.1 shows some of the parameters used in the evaluation. Table 5.2 shows the rendering times.

The used light field has a lossless compressed size of about 1.6 TB with 999127 calibrated light field images. The capture positions form more of a band (band height less than 1 cm) than a hemisphere, which restricts the possible largest aperture size, as well as feasible movement. Originally, it was planned to run this pipeline in a cluster. Due to technical difficulties with the provided cluster, this was not possible until now. However, running the produced code in the cluster should be straightforward, since there is no communication necessary between render nodes- one merely runs instances of **lfrenderer** as individual jobs.

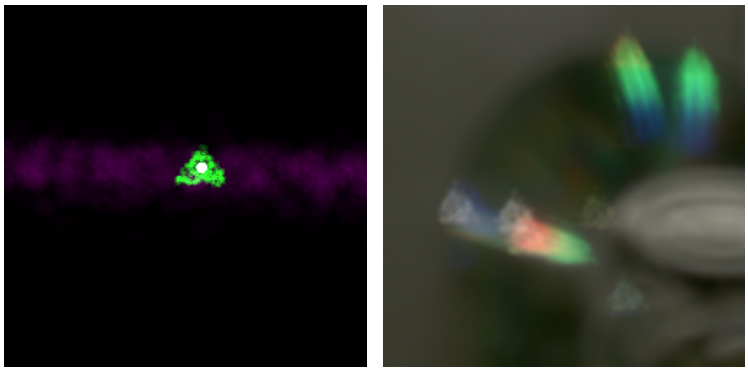
Camera movement	*stack*	*short*	*sharp*	*long*	*bokeh*
Time	1 s	2 s	1 s	16 s	30 s
Frames	30	60	30	480	900
Fps	30	30	30	30	30
Aperture diameter	4 mm	4 mm	2 mm	2 mm	0.1 - 2 mm
Focus distance	20 - 100 cm	10 - 50 cm	80 cm	15 - 100 cm	15 - 100 cm
Light field images	827	5621	1914	27173	67502

Table 5.1: Selection of the camera movements that where used for evaluation. Both *long* and *bokeh* indicate the difficulty of rendering camera movements, the amount of relevant data increases significantly with longer movements. Stationary renderings such as the focal stack *stack* require less light field data.

5.1 Relevant Camera Extraction

Extraction of the relevant cameras, together with their contributing image space region is quite fast- reading in all camera extrinsic parameters often takes longer than the

extraction itself. Interestingly, if rendered cameras are placed between the original light field capture positions with a shaped aperture (e.g. a triangle), the collected light field cameras positions tend to form the aperture's shape, see Figure 5.1. Recall that the extended frustum does not model the aperture shape in any way, so this effect is caused entirely by the projection and overlapping of the focus plane and lens approximation. This approach is also effective for discarding non-contributing light field images, see Figure 5.5.



(a) Camera selection **(b)** Rendered view

Figure 5.1: Selection of cameras from the light field (purple) and resulting rendering. The rendered camera (white) has a triangle-shaped aperture, which is reflected in the selection of cameras (green).

5.2 Ray Collection Approach

The ray collection technique performs quite well if the used light field is dense- enough ray samples hit the sensor and a very smooth and clear image is obtained. This approach is not applicable for general light fields since those might by sparsely sampled. In the case of a larger aperture, depth of field effects and bokeh in out-of-focus areas can be observed very clearly, see Figure 5.2. Light field density affects the obtainable image quality, see Figure 5.4. Noise has low impact due to many ray samples being averaged.

It is possible that bokeh effects are "cut off", an effect caused by missing data. For example, if the camera is placed among the light field cameras, and the aperture is larger than the light fields spatial resolution, radiance information will be missing for parts of the aperture, an effect also described by [IMG00].

While the preprocessing step helps immensely, see Figure 5.5, performance is still an issue with this approach. Consider rendering a focal stack consisting of n frames, i.e. the camera is standing still while modifying the focus distance in n frames. Also assume this

Figure 5.2: Frame from the *bokeh* movement. With the provided light field, bokeh effects are obtainable by selecting low focus distance and large aperture settings. Also note how the approach described in Section 4.3.3 enables shaped bokeh effects, in this case a pentagon.

camera is placed among the light field cameras and n' light field frames were selected. It is now highly likely that all light field images that are to be processed are contributing with radiance in all rendered images. The required effort is therefore $n' \times n$. Note that in this case, likely most light field images are completely relevant, i.e. all recorded rays are useful and contribute in the rendered frames. Profiling with `gprof` [GKM82] revealed that most CPU time is spend doing linear algebra operations, e.g. matrix-vector multiplication, as well as intersection tests. About 90% of computational effort was spend on such operations. A GPU implementation might help with performance.

Processing takes long if a large number of rendered images use rays of the currently processed light field frame, e.g. for the focal stack rendering in Figure 5.3, almost every light field image contributed with radiance in every rendered image. This has to be accepted if dense light field data is to be exploited. The main advantage of this approach is the easily controllable memory requirements- the synthetic sensor plates have the largest memory footprint with $width \times height \times 6 \times 4\ B$ for a single rendered frame, but do not change in size over the rendering process.

Figure 5.3: Selection of frames from *stack*. From left to right, top to bottom: Decrease of the focus distance. A smooth depth of field effect becomes visible. 827 light field frames contributed with their captured radiance. The aperture's diameter was chosen as 4 mm.

5.3 Sparse On the Fly Rebinning

The rebinning technique is not able to construct images as clean as the ray collected ones. Ideally rebinning is turned off if the ray collection approach gathered enough data, this approach should only serve as a backup solution. Naturally, discretization levels have high impact on the image sharpness, however lower discretization resolution leads to less noise. Another effect is that missing data is easily interpolated from present rays, e.g. if the camera's aperture is larger than the light field's captured positions, round bokeh effects are still obtainable. Figure 5.6 shows how interpolated ray information is used to fill gaps. This also serves as a preview of the attainable image to the user. Sparse rebinning also leads to noise appearing in light field images having stronger impact on the rendered result, because sparse ray samples from noisy regions in L will have stronger impact on rendered images- less samples are available to suppress resulting noise effects. If image values are linearized via the inverse response curve first, this problem amplifies. This is especially apparent in darker regions of rendered images. Low pass filtering the image before utilizing any rays reduces this problem, e.g. by a simple box filter. Kernel size should be chosen small to preserve details. This could also be solved by a slight modification of the randomized ray selection: Every ray sample present in the image is considered for rebinning, but only those that are randomly selected are

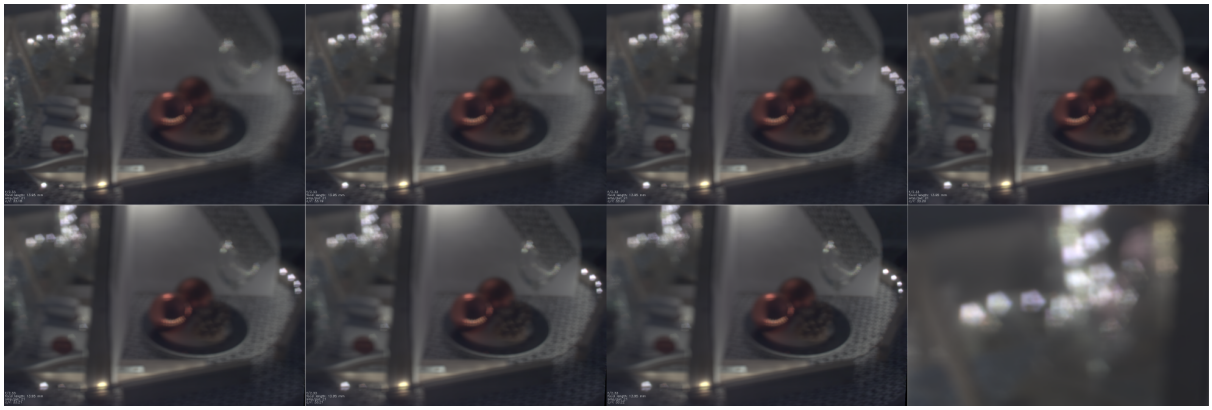
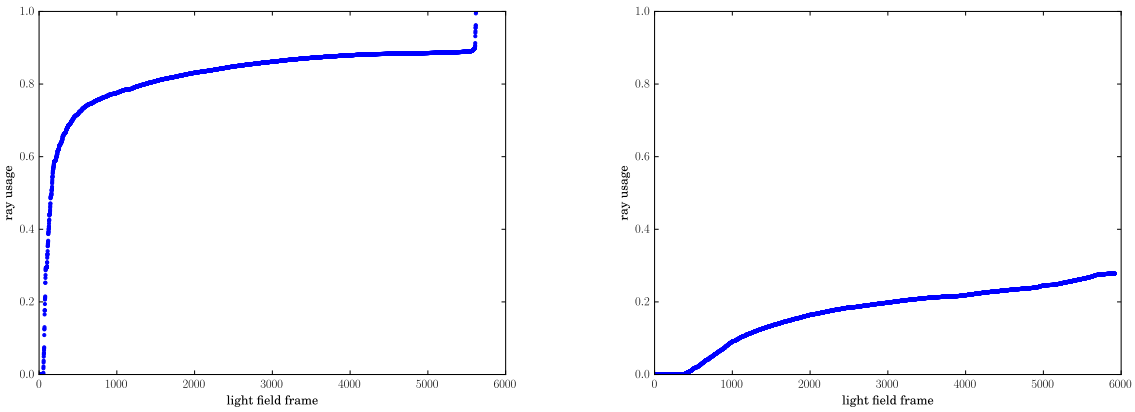


Figure 5.4: Rendering from a different light field that is about twice as dense. The focus plane was tilted along the vertical axis. The bottom right image shows a magnified view of one of the frames. The bokeh effects are smoother than in Figure 5.2.

able to create a new discretized ray. Those rejected are still used for radiance averaging if their corresponding rebinning hypercube is already present. However, this requires parameterizing every single ray in the image and testing for possible rebinning- an operation deemed to expensive for the designated augmenting nature of the technique. See Table 5.2 how sparse rebinning affects performance. Another problem are mis-calibrated light field images, i.e. camera positions that the calibration placed in the wrong positions. Rebinned rays from such cameras could potentially have strong impact in low-frequency regions, since those wrongly placed rays are more likely used for interpolation and assigned bigger importance.

Since only a fraction of rays is actually considered for rebinning, processing of light field images is way faster than in the first approach. Noise as it is typically associated with distributed raytracing occurs if the sample count per pixel is not high enough. Another issue is the constantly increasing size of the ray collection. Capping the total amount of rebinned rays, as well as removal of unused rays, is absolutely necessary. The main advantage of this technique is its flexibility. It is possible to acquire decent depth of field effect from a low number of light field frames, which is generally not possible with the ray collection approach, see Figure 5.8. Therefore sparse rebinning could also be applicable in less dense light fields.

Another problem is the monotonically increasing amount of collected rays. If the discretization interval is chosen too small, the user-given ray limit is reached fast despite all described countermeasures.



(a) Ray usage with preprocessing

(b) Ray usage without preprocessing

Figure 5.5: Depicted here is the effectiveness of the preprocessing step on rendering effort. The frames were ordered according to their ray usage $\frac{\text{used rays}}{\text{estimated rays}}$. On the left hand side, ray usage for the camera movement shown in Figure 5.9. 5621 light field frames where processed, but only the predetermined regions in image space. On the right, ray usage when the whole image is considered. 5920 Frames were selected, there was no filtering based on ray contribution happening. With preprocessing, rendering of the camera movement took 36 minutes (See Table 5.2), while considering the entire image took 133 minutes, an increase in render time of factor 3.7. Without preprocessing, lots of rays are rejected because they do not contribute to the rendered images. About 300 image did not contribute any radiance or close to nothing. Note that those images do not appear in Figure a, as those are discarded beforehand.

5.4 Effects of Wavelet Compression

Naturally, lossy compression downgrades image quality in exchange for smaller size of a light field image. Also note that using the `Squash` library might not be the best choice for compression of the quantized and thresholded wavelet-transformed images. Still, the light field's size can be reduced significantly, see Table 5.3. Transforming only once in each image dimension works best. Otherwise, quantization artifacts resulting from the mapping of the whole detail coefficients range to only one byte appear. Error is further reinforced by higher threshold values. Also, decompression is faster as only one inverse transformation step has to be done per image dimension.

Using 16-bit integers works better if more than one transformation step is done in each image dimension. By providing an additional parameter q, the mapping can be

(a) No rebinning

(b) With rebinning

Figure 5.6: Example frame from the *bokeh* camera movement. The aperture size was reduced so far that the ray collection approach was not able to achieve a sufficient result. Gaps appear and the aperture shape becomes apparent. The missing information is interpolated from the sparse data in Figure b.

forced to only use q many values. E.g. $q = 1024$ allows mapping to 1024 different values. This also gives the user more control over the achievable compression ratio. Less quantization artifacts will appear if q is chosen high enough. However, compression ratio will naturally suffer from this, since twice the amount of bytes are used for the (uncompressed) transformed image. Figure 5.10 shows the effect of Wavelet compression when rendering with such data.

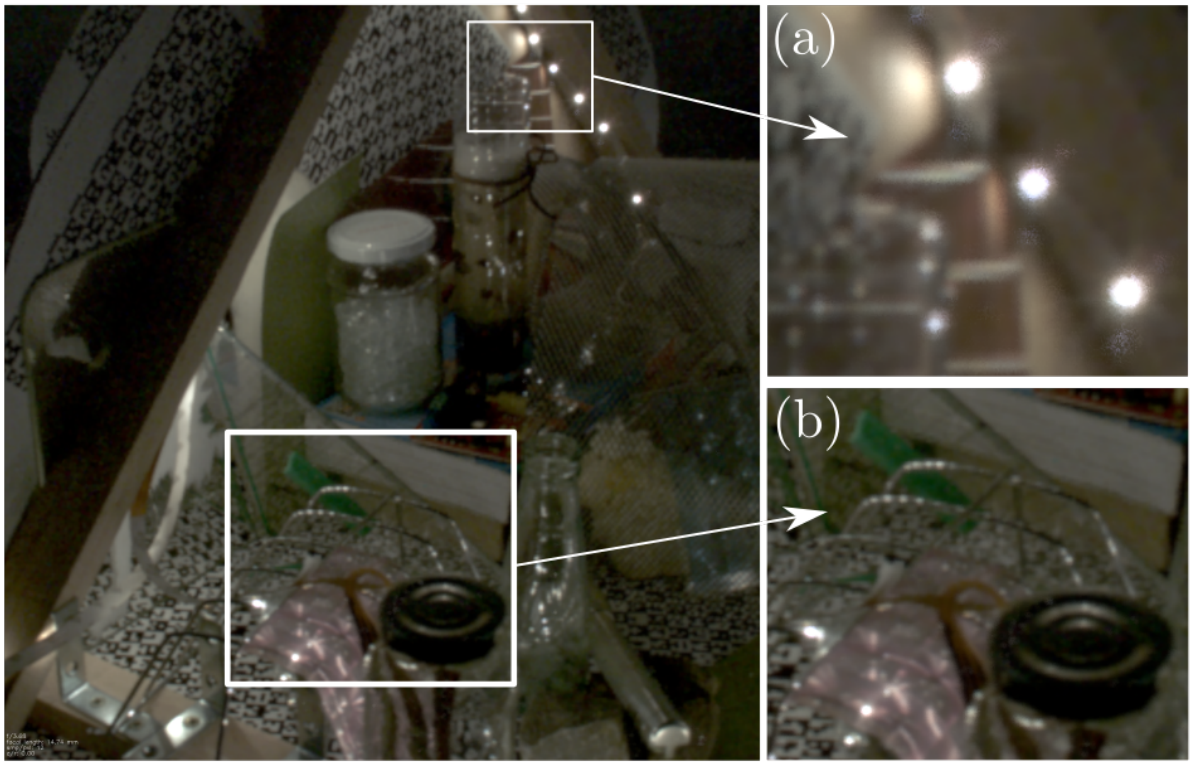


Figure 5.7: Sparse rebinning leads to lower image quality. As seen in (a), the background is slightly out of focus. Distributed raytracing leads to the appearance of noise. Additionally, some artifacts might appear as seen on the bottles' metal cap in (b).

Summarizing, the current implementation reduces light field size, but could be improved further. Decent results are obtainable if the amount of transformation steps is set to one, and thresholding is applied at about 8. Simple hard thresholding seems to not work well if more transformation steps are applied. A more sophisticated adaptive thresholding strategy could be used.

Figure 5.8: In this frame, raytracing was used to sample the sparsely rebinned light field. To acquire strong depth of field, the focus distance was chosen as 20 cm. 21 ray samples were taken per pixel. Distributed raytracing noise becomes visible. Note however, that bokeh effects are not cut off as it can be observed in Figure 5.4, missing data is interpolated.

	stack	*short*	*sharp*	*long*		*bokeh*	
	total	**total**	**total**	**total**	**longest section**	**total**	**longest section**
Ray Collect:	22 min 33 s	36 min 30 s	5 min 25 s	244 min 42 s	114 min 10 s	310 min 21 s	102 min 42 s
Ray Collect & Rebinning:	23 min 38 s	41 min 17 s	6 min 53 s	271 min 35 s	128 min 6 s	438 min 18 s	129 min 20 s

Table 5.2: Performance results for some camera movements in the light field. Given are the total times for rendering. The *long* and the *bokeh* movement could not be rendered on a single node in one go. For those movements, also the longest section is given. Ray collection was relatively fast for *bokeh*, since the aperture was reduced close to zero in parts of the movement, yielding fewer rays to process in large parts of the movements. Render time is reduced by a factor of about two to three if up to four machines are processing in parallel. The section that takes the longest time zu process determines the bottleneck.

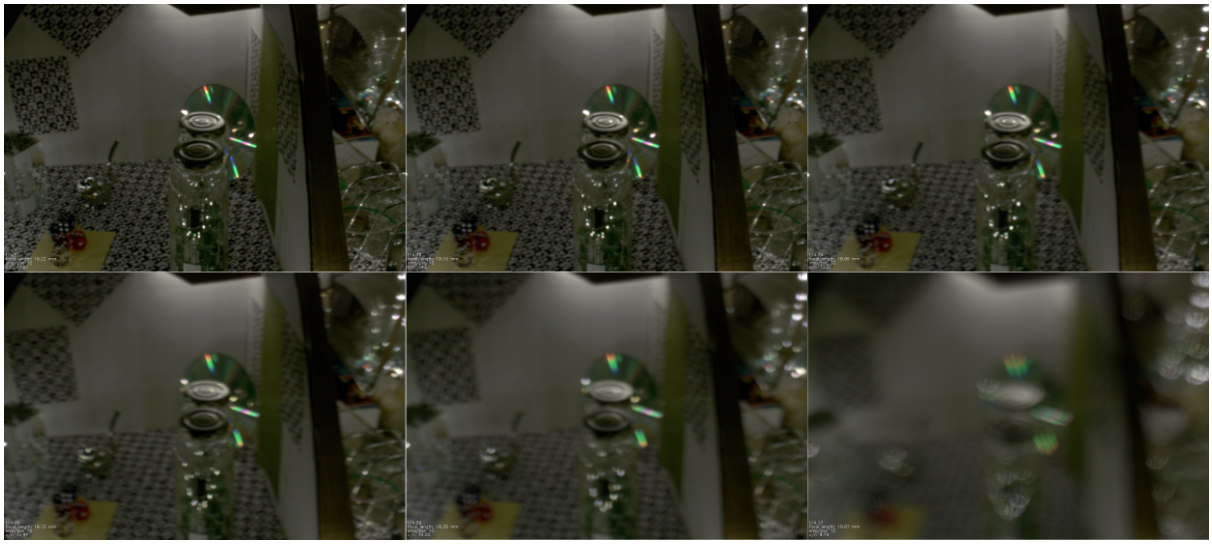


Figure 5.9: Selection of frames from the *short* movement. The camera was moved by about 2 cm, the focus distance was decreased over time. The render times can be found in Table 5.2.

transf. steps	threshold	compr. time	avg compr. ratio	size
1	2.0	1 min 45 s	11.7	694 MB
1	4.0	1 min 44 s	13.7	596 MB
1	8.0	1 min 41 s	15.1	542 MB
2	0.5	2 min 5 s	7.5	1.1 GB
2	2.0	1 min 56 s	21.1	391 MB
2	4.0	1 min 54 s	30.6	275 MB
3	4.0	1 min 56 s	44.6	192 MB

Table 5.3: Compression performance on the raw light field images. The 1914 required light field images of the *sharp* camera movement were compressed. The **threshold** column refers to the coefficient dropping threshold. For subsequent compression of the transformed image, the *compress* algorithm provided by `Squash` was used. Compression takes longer if more transformation steps are done.

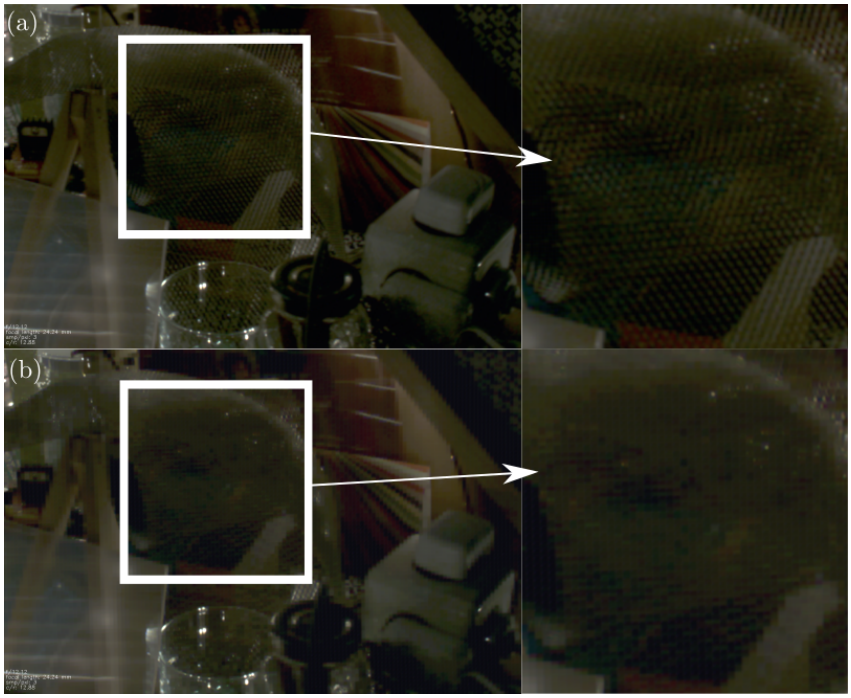


Figure 5.10: Compression artifacts occurring from Wavelet compression also appear in renderings. (a) shows the image transformed one time in each dimension, and thresholded with 8.0. (b) shows the same frame but light field images were transformed two times and thresholded with 4.0.

6 Future Work

The proposed system enables rendering of camera movements in densely recorded light fields on multiple nodes. Additionally, it also offers capabilities for lossy compression of the light field data. However, the system could be extended to support additional features which might be advantageous when processing dense light fields.

6.1 Progressive Rendering with Wavelet Coefficients

The compression scheme using Haar-Wavelets (See Section 2.5) could be used for the ray collection approach described in Section 4.3.1 to achieve progressive rendering similar to [PS01]. Instead of radiance, rays could carry wavelet coefficients. Recall that reconstruction of a wavelet-transformed image is achieved by adding detail coefficients to approximation coefficients. Reordering coefficients according to their importance (e.g. approximation coefficients before detail coefficients) enables render nodes to process those first. In a first steps, all averaged coefficients could be loaded and processed. This would yield the rendering achievable by downsampling by simple averaging of pixels. In subsequent steps, the detail coefficients for each light field image could be loaded and added to the results. To achieve this, one could compute which light field rays are influenced by the coefficients. Those rays are then just recollected in the synthetic sensor plates by addition to the total collected ray information, but without incrementation of the ray counter.

The wavelet compression scheme implemented in this work could be extended to support arbitrary ordering of coefficients, e.g. similar to the SPIHT method [SP96].

Note that as long as individual rays are considered, required computational effort will increase, since ray collection must occur in multiple rounds as more and more coefficients are loaded. However, coefficients that are zero can be discarded right away, since those have no influence on any rendered pixels anyway. To compress the data, it could be organized in coefficient packages (e.g. as parts of the wavelet transformed image) which are compressed in a lossless fashion. This would increase required runtime further, since decompression needs additional time.

It remains a question how beneficial progressive loading of wavelet coefficient is in general. If camera movements are to be rendered, processing time might be slightly longer than loading a single light field image in a background thread, since computational effort is increased in comparison to single image rendering.

6.2 Resampling the Dataset

Resampling a single image usually consist of filtering the input image with some filter function to obtain a differently sampled image- e.g. one with higher or lower resolution [SM09]. This concept could be applied on entire light fields, too. Currently, the light field is represented as a stream of images. This is not ideal if one wants to directly evaluate $L(s, t, u, v)$, since the current representation does not enable such operations on the light field. If the light field was represented as a 4D-Matrix, it could be evaluated directly. Resampling L from the light field images into a 4D matrix would yield such a representation. However, several problems have to be additionally considered:

1. Some suitable parameterization has to be found. In case of light slabs, multiple ones are required since the light field was captured from a hemisphere.
2. To apply resampling operations, one needs to know which light field ray samples from all light field images are closest to a ray that is to be resampled- this is computationally not trivial.
3. A renderer should be able to access the resampled data efficiently. This includes that resampled data should be loaded in packages, i.e. relevant data that is accessed should be loaded at once to increase network throughput. In the current representation, unnecessary data is being loaded if the rendered camera's aperture is small, or the position is too far away from the original capture positions, as only few rays of an image contributed with radiance. This problem does not necessarily vanish if the light field is resampled. Data must be organized in such a way that random access is possible *and* multiple samples are loaded at once. For example, if one always loads the surrounding rays around a requested ray, and the rendered camera is far away from the resampled light field slab, those rays might differ too much from required rays to be of any use in the novel view.
4. In case of dense light field, this resampling process could take a long time. Consider a light field consisting of 2048×2048 sized images. If the light field consists of 10^6 images, at least $4 \cdot 10^{12}$ rays have to be considered.
5. Additionally, the resampled data should be compressed. 4D-Wavelet compression could be used in this case.

Note that the approach described in Section 4.3.2 tries to accomplish something similar to this, but without a sophisticated resampling filter.

7 Conclusion

Rendering in dense light fields is a difficult task as the high amount of available data has to be processed. The requirement for camera movement rendering makes this even more difficult, as required data increases quite drastically. This work presents a technique allowing the extraction of relevant light field images, given a predefined camera movement of some thin lens camera. Additionally, it is estimated which rays are likely refracted onto the synthetic sensor, reducing required computational effort. Rendering simulates a thin lens camera by considering images as sets of rays, and simulating their refraction by the rendered camera's lens. Additionally, a sparse rebinning technique based on hash maps and kd-trees was developed. While the first approach works quite good and produces decent results, the second approach should be considered to be in a supporting role since this technique is able to interpolate missing data, but is less accurate. On the other hand, the approach is computationally relatively inexpensive and requires low amounts of memory. It also tries to adapt by only keeping data that is likely needed for interpolation. Still, both approaches work best if the rendered camera movement stays close to the original light field capture positions. Utilizing multiple nodes to process the data is necessary to obtain results in a reasonable amount of time, see Table 5.2. To augment the lossless compression scheme introduced by Siedelmann, simple 2D-Wavelet compression is employed.

Bibliography

[AB91] E. H. Adelson, J. R. Bergen. “The plenoptic function and the elements of early vision.” In: *Computational Models of Visual Processing*. MIT Press, 1991, pp. 3–20 (cit. on p. 15).

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu. “An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions.” In: *J. ACM* 45.6 (Nov. 1998), pp. 891–923 (cit. on p. 30).

[BBM+01] C. Buehler, M. Bosse, L. McMillan, S. Gortler, M. Cohen. “Unstructured Lumigraph Rendering.” In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: ACM, 2001, pp. 425–432 (cit. on p. 34).

[BEL+06] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, I. Wald. *Interactive distribution ray tracing*. Tech. rep. Technical Report, SCI Institute, University of Utah, No UUSCI-2006-022, 2006 (cit. on p. 53).

[BOB13] C. Birklbauer, S. Opelt, O. Bimber. “Rendering Gigaray Light Fields.” In: *Computer Graphics Forum* (2013) (cit. on pp. 33, 34).

[CLF98] E. Camahort, A. Lerios, D. Fussell. “Rendering Techniques ’98: Proceedings of the Eurographics Workshop in Vienna, Austria, June 29—July 1, 1998.” In: ed. by G. Drettakis, N. Max. Vienna: Springer Vienna, 1998. Chap. Uniformly Sampled Light Fields, pp. 117–130 (cit. on pp. 17, 18).

[Com] S. U. Computer Graphics Laboratory. *The (New) Stanford Light Field Archive*. http://lightfield.stanford.edu/lfs.html. Accessed 2016-05-09 (cit. on p. 33).

[CPC84] R. L. Cook, T. Porter, L. Carpenter. “Distributed Ray Tracing.” In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 137–145 (cit. on pp. 25, 26).

[CSE00] C. Christopoulos, A. Skodras, T. Ebrahimi. “The JPEG2000 still image coding system: an overview.” In: *IEEE Transactions on Consumer Electronics* 46.4 (Nov. 2000), pp. 1103–1127 (cit. on p. 28).

[CV15] *OpenCV (Open Source Computer Vision)*. http://opencv.org/. Accessed 2016-05-01. 2015 (cit. on pp. 21, 53).

[CW93] S. E. Chen, L. Williams. “View Interpolation for Image Synthesis.” In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’93. Anaheim, CA: ACM, 1993, pp. 279–288 (cit. on p. 25).

[EF03] M. T. El-Melegy, A. A. Farag. “Nonmetric Lens Distortion Calibration: Closed-form Solutions, Robust Estimation and Model Selection.” In: *9th IEEE International Conference on Computer Vision (ICCV 2003), 14-17 October 2003, Nice, France*. 2003, pp. 554–559 (cit. on p. 22).

[Eig16] *Eigen3*. http://eigen.tuxfamily.org/index.php?title=Main_Page. Accessed 2016-05-01. 2016 (cit. on p. 53).

[GGSC96] S. J. Gortler, R. Grzeszczuk, R. Szeliski, M. F. Cohen. “The Lumigraph.” In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: ACM, 1996, pp. 43–54 (cit. on pp. 11, 13, 15–18, 34, 48).

[GKM82] S. L. Graham, P. B. Kessler, M. K. Mckusick. “Gprof: A Call Graph Execution Profiler.” In: *SIGPLAN Not.* 17.6 (June 1982), pp. 120–126 (cit. on p. 59).

[Gla89] A. S. Glassner, ed. *An Introduction to Ray Tracing*. London, UK, UK: Academic Press Ltd., 1989 (cit. on pp. 11, 20).

[Goo10] Google. *sparsehash*. https://github.com/sparsehash/sparsehash. Accessed 2016-05-01. 2010 (cit. on p. 53).

[HSS97] W. Heidrich, P. Slusallek, H.-P. Seidel. “An Image-based Model for Realistic Lens Systems in Interactive Computer Graphics.” In: *Proceedings of the Conference on Graphics Interface ’97*. Kelowna, British Columbia, Canada: Canadian Information Processing Society, 1997, pp. 68–75 (cit. on pp. 20, 26).

[Ihm97] L. R. K. Ihm I. Park S. “Rendering of Spherical Light Fields.” In: 1997 (cit. on p. 16).

[IMG00] A. Isaksen, L. McMillan, S. J. Gortler. “Dynamically Reparameterized Light Fields.” In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 297–306 (cit. on pp. 34, 51, 58).

[Joh14] A. Johnson. *Clipper - an open source freeware library for clipping and offsetting lines and polygons.* http://www.angusj.com/delphi/clipper.php. Accessed 2016-05-01. 2014 (cit. on p. 53).

[Kaj86] J. T. Kajiya. “The Rendering Equation.” In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’86. New York, NY, USA: ACM, 1986, pp. 143–150 (cit. on pp. 11, 12).

[LH96] M. Levoy, P. Hanrahan. “Light Field Rendering.” In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: ACM, 1996, pp. 31–42 (cit. on pp. 11, 15–18, 33, 47).

[LPC+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, D. Fulk. “The Digital Michelangelo Project: 3D Scanning of Large Statues.” In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 131–144 (cit. on p. 33).

[MEG00] M. Marcus, A. Endmann, B. Girod. *Progressive Compression and Rendering of Light Fields*. VMV 2000. Nov. 2000 (cit. on p. 18).

[ML09] M. Muja, D. G. Lowe. “Fast approximate nearest neighbors with automatic algorithm configuration.” In: *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009, pp. 331–340 (cit. on pp. 30, 31).

[ML14] M. Muja, D. G. Lowe. “Scalable Nearest Neighbor Algorithms for High Dimensional Data.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (Nov. 2014), pp. 2227–2240 (cit. on p. 53).

[NLB+05] R. Ng, M. Levoy, M. Brédif, G. Duval, M. Horowitz, P. Hanrahan. *Light Field Photography with a Hand-Held Plenoptic Camera*. Tech. rep. Apr. 2005. URL: http://graphics.stanford.edu/papers/lfcamera/ (cit. on p. 17).

[PH10] M. Pharr, G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010 (cit. on pp. 26, 30).

[PS01] I. Peter, W. Straßer. “Rendering Techniques 2001: Proceedings of the Eurographics Workshop in London, United Kingdom, June 25–27, 2001.” In: ed. by S. J. Gortler, K. Myszkowski. Vienna: Springer Vienna, 2001. Chap. The Wavelet Stream: Interactive Multi Resolution Light Field Rendering, pp. 127–138 (cit. on pp. 18, 30, 44, 69).

[SD96] S. M. Seitz, C. R. Dyer. “View Morphing.” In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: ACM, 1996, pp. 21–30 (cit. on p. 25).

[SDS95] E. J. Stollnitz, T. D. Derose, D. H. Salesin. “Wavelets for Computer Graphics: A Primer - Part 1.” In: *IEEE Computer Graphics and Applications* 15 (1995), pp. 76–84 (cit. on pp. 28, 29).

[She68] D. Shepard. “A Two-dimensional Interpolation Function for Irregularly-spaced Data.” In: *Proceedings of the 1968 23rd ACM National Conference*. ACM ’68. New York, NY, USA: ACM, 1968, pp. 517–524 (cit. on p. 47).

[Sie15] H. Siedelmann. “Recording, compression and representation of dense light fields.” eng. MA thesis. Holzgartenstr. 16, 70174 Stuttgart: Universität Stuttgart, 2015. URL: http://elib.uni-stuttgart.de/opus/volltexte/2015/9926 (cit. on pp. 12–14, 17, 18, 34, 53).

[SM09] P. Shirley, S. Marschner. *Fundamentals of Computer Graphics*. 3rd. Natick, MA, USA: A. K. Peters, Ltd., 2009 (cit. on pp. 19, 27, 30, 70).

[SP96] A. Said, W. A. Pearlman. “A new, fast, and efficient image codec based on set partitioning in hierarchical trees.” In: *IEEE Transactions on Circuits and Systems for Video Technology* 6.3 (June 1996), pp. 243–250 (cit. on p. 69).

[Squ15] *Squash*. https://quixdb.github.io/squash/. Accessed 2016-05-01. 2015 (cit. on p. 53).

[SS] D. Scharstein, R. Szeliski. “A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms.” In: *International Journal of Computer Vision* 47.1 (), pp. 7–42 (cit. on p. 25).

[SSF05] L. Sylvain, H. Samuel, B. Fabrice. *Chapter 37. Octree Textures on the GPU*. 2005. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter37.html (visited on 04/12/2016) (cit. on p. 31).

[Sze10] R. Szeliski. *Computer Vision: Algorithms and Applications*. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010. URL: http://szeliski.org/Book (cit. on pp. 19–23, 25, 27, 28, 30, 44).

[TH10] K. H. Talukder, K. Harada. “Haar Wavelet Based Approach for Image Compression and Quality Assessment of Compressed Image.” In: *CoRR* abs/1010.4084 (2010). URL: http://arxiv.org/abs/1010.4084 (cit. on pp. 28–30).

[TRK07] S. Todt, C. Rezk-Salama, A. Kolb. *Fast (Spherical) Light Field Rendering with Per-Pixel Depth*. Tech. rep. Siegen, 2007 (cit. on pp. 16, 17).

[Vai07] V. Vaish. “Synthetic Aperture Imaging Using Dense Camera Arrays.” AAI3253550. PhD thesis. Stanford, CA, USA, 2007 (cit. on p. 17).

[Wei] E. W. Weisstein. *Disk Point Picking. From MathWorld—A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/DiskPointPicking.html (visited on 04/21/2016) (cit. on p. 53).

[Wet] G. Wetzstein. *Synthetic Light Field Archive*. http://web.media.mit.edu/~gordonw/SyntheticLightFields/. Accessed 2016-05-09 (cit. on p. 33).

[WIG+15] A. Wender, J. Iseringhausen, B. Goldlücke, M. Fuchs, M. B. Hullin. “Light Field Imaging through Household Optics.” In: *Vision, Modeling & Visualization*. Ed. by T. S. David Bommes Tobias Ritschel. The Eurographics Association, 2015 (cit. on pp. 50, 51).

[WJV+05] B. Wilburn, N. Joshi, V. Vaish, E.-V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz, M. Levoy. “High Performance Imaging Using Large Camera Arrays.” In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 765–776 (cit. on p. 17).

[WZH+10] J. Wu, C. Zheng, X. Hu, Y. Wang, L. Zhang. “Realistic rendering of bokeh effect based on optical aberrations.” In: *The Visual Computer* 26.6 (2010), pp. 555–563 (cit. on p. 23).

[WZHX13] J. Wu, C. Zheng, X. Hu, F. Xu. “Rendering realistic spectral bokeh due to lens stops and aberrations.” In: *The Visual Computer* 29.1 (2013), pp. 41–52 (cit. on p. 52).

[Yaml02] *YAML: YAML Ain’t Markup Language*. http://www.yaml.org/. Accessed 2016-05-08. 2002 (cit. on p. 54).

[YEBM02] J. C. Yang, M. Everett, C. Buehler, L. McMillan. “A Real-time Distributed Light Field Camera.” In: *Proceedings of the 13th Eurographics Workshop on Rendering*. EGRW ’02. Pisa, Italy: Eurographics Association, 2002, pp. 77–86 (cit. on p. 17).

[YGV98] I. T. Young, J. J. Gerbrands, L. J. Van Vliet. *Fundamentals Of Image Processing*. Available at https://www.researchgate.net/publication/2890160_Fundamentals_Of_Image_Processing, accessed April 24, 2016. 1998 (cit. on pp. 27, 28).

[Zha00] Z. Zhang. “A Flexible New Technique for Camera Calibration.” In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22.11 (Nov. 2000), pp. 1330–1334 (cit. on p. 22).

[ZZ09] W. Zinth, U. Zinth. *Optik: Lichtstrahlen - Wellen - Photonen*. Oldenbourg, 2009 (cit. on pp. 22–24).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature