Polyflare:
Sparse Polynomial Modeling for Efficient Approximate Lens Flare Rendering

Institute of Computer Science II

Rheinische Friedrich-Wilhelms-Universität Bonn

Author
Lukas Sabatschus

Supervisor
Prof. Dr. Matthias B. Hullin

April 30, 2022

Abstract

Lens flares are an artifact of light passing through a photographic lens in unintended ways. Rendering these is invaluable for many applications, but no public implementation of physically-based real-time rendering exists to our knowledge. In this thesis, we present a technique for using sparse polynomials to more efficiently render lens flares, which were used previously to model lens distortion and aberrations. Additionally, we provide a flexible implementation of select previous methods for interactive lens flare rendering and our polynomial approach.

Contents

1 Introduction

Lens flares are made up of different artifacts caused by light paths other than the imaging path of a lens system, the most obvious ones of which are ghosts and diffraction at the aperture. Ghosts are different shapes on the resulting image caused by internal reflections of bright light sources off different lens surfaces. In lens design, they are considered to be a degrading artifact, because they reduce the contrast and distract from the primary subject. Undesired light being rendered onto the sensor is commonly reduced either by shielding the lens using lens hoods or by altering the lens design to incorporate anti-reflective coatings.

In computer graphics, cinematography, and artistic photography, lens flares are often desired as they increase the perceived brightness of bright light sources and add visual interest. For this reason there already exist many approaches to rendering lens flares [14]. These approaches are typically based on ray tracing, which is a technique for rendering a scene by tracing rays from a light source to the sensor. Ray tracing does not simulate any wave optical effects, but these can be added using a technique by Hullin et al. [5], which we will call sparse ray tracing.

We present a novel technique for improved lens flares using sparse polynomials and several methods for fitting polynomials for this application.

We also provide a library for our novel technique including fitting methods, dense and sparse ray tracing, and two applications using this library.

Outline The remainder of this thesis is organized as follows. Chapter 2 describes our model of the lens system and how we render ghosts using traditional ray tracing and the sparse technique. Then in chapter 3 we present our novel technique for improved lens flare rendering using polynomials and how to fit dense and sparse polynomials to points generated by ray tracing. Afterwards we present our implementation in chapter 4. Our results are discussed in Chapter 5. Finally, Chapter 6 gives the conclusion of this thesis.

2 Light transport

This chapter begins by defining our model of lens systems including the sensor. Then we discuss our use of spectral rendering, and finally, the dense and sparse ray tracing techniques are discussed.

2.0.1 Lens Model

We model a lens system as a list of elements followed by a sensor, as shown in figure 2.5. The elements are either a lens surface or the aperture. Every element has a radius and a position.

Apertures additionally include their number of blades.

Lens surfaces also consist of coatings, and one set of Sellmeier coefficients for outside the lens and one set for inside.

The sensor replaces the radius, with a function from wavelength to an RGB value, which is specified in section 2.2.4 and needed for spectral rendering, as discussed in section 2.1.

2.1 Spectral Rendering

In more traditional ray tracing applications, each ray carries information about the RGB color of the light from the source to the sensor. To model spectral phenomena, such as those discussed later in this section, each ray is assigned a wavelength and propagated through the lens system loosing energy as it passes through each element. Once a ray hits the sensor, the wavelength and energy is used to calculate the RGB color and intensity of the light.

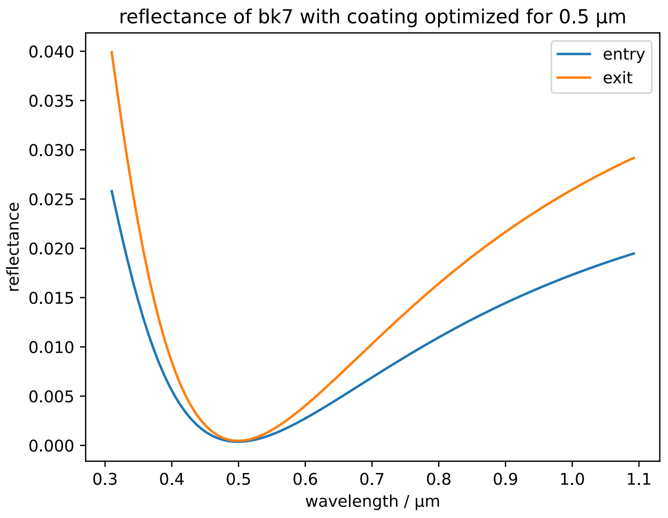
Rendering using spectral rendering is more computationally expensive than traditional ray tracing, because many rays are needed to fill out the spectrum and form white light. This performance cost is necessary however, because many ghosts exhibit strong dispersion, and the anti-reflective coatings color the ghosts in different colors. Both of these factors are important for the realism of the lens flare and can only be modeled directly using spectral rendering.

2.1.1 Coatings

Coatings on a lens surface are used to reduce the amount of light reflected by the surface.

Here we only consider "quarter-wave" coatings, which are described by a thickness and an index of refraction. "Quarter-wave" coatings are made up of a thin layer of low refractive index material. These minimize reflections at a single wavelength and a single angle of incidence, so each set of parameters has a certain color of light that passes through preferably, an example of which can be seen in figure 2.1. This means to get a lens system that renders colors faithfully, one needs to use multiple kinds of "quarter-wave" coatings. This faithful rendering for the non-reflected path through the lens results in colorful ghosts, because each ghost takes a different path through the lens and therefore different wavelengths are accentuated or reduced by the coatings.

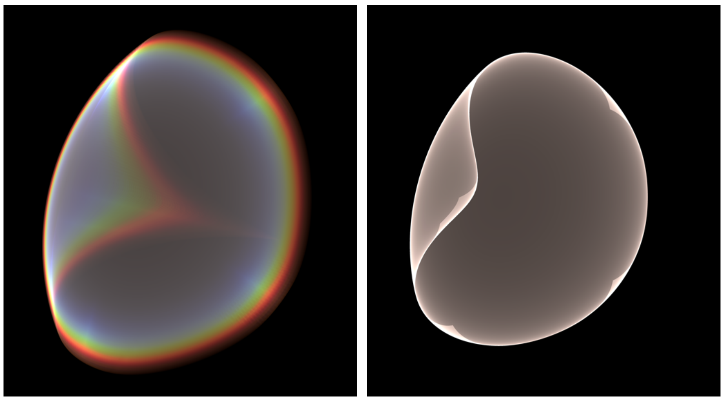
Figure 2.1: Coating



2.1.2 Dispersion

Dispersion is the phenomenon that the index of refraction of a material changes with wavelength, resulting in light of different wavelengths being imaged to a different position on the sensor by a lens. In ghosts, the light is scattered by the lens, multiple times, and in ways where dispersion has not been minimized by the lens design. This results in much stronger dispersion than in the direct image. Figure 2.2 shows an extreme case of this.

Figure 2.2: A particular ghost with and without dispersion. Rendered with Polyflare.



To model dispersion, we use the Sellmeier equation [18], which approximates the refractive index of a material as a function of wavelength. Equation 2.1 is the most common Sellmeier equation and has three terms.

$$\text{The Sellmeier equation: } n^2(\lambda) = 1 + \frac{B_1\lambda^2}{\lambda^2 - C_1} + \frac{B_2\lambda^2}{\lambda^2 - C_2} + \frac{B_3\lambda^2}{\lambda^2 - C_3} \tag{2.1}$$

The coefficients B_1, B_2, B_3 and C_1, C_2, C_3 are readily available for common materials used in lens design.

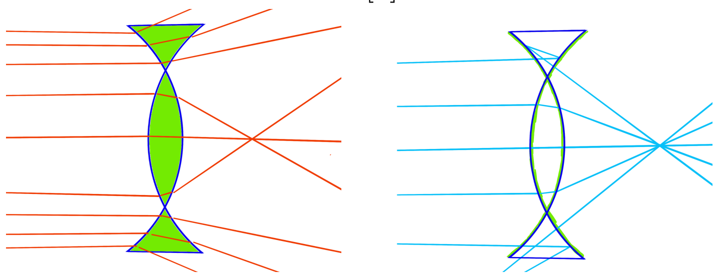
2.2 Ray tracing

The most naive technique for rendering ghosts is recursive ray tracing. This works by generating rays at the light source and tracing those through the lens system. Because of the number of rays needed for an acceptable noise level this is usually reserved for offline rendering, but it results in the most accurate images. These properties make it viable to use in high-budget movies like “The Lego Movie 2: The Second Part” [13].

We don’t use the usual approach of recursive or randomized reflection or refraction at each surface, but rather enumerate all possibilities and generate each ray once for each. This is more efficient, because the decision of whether to reflect or refract is done only once and does not have to be made each time the ray is intersected with a lens element.

Additionally, the order of intersections is given by the order of the elements, not by the order of intersections, as described by Hullin et al. [5].

Figure 2.3: "Left: traditional intersection with the nearest surface along the way.
Right: our intersection in fixed sequence. Note how the outermost rays still converge to the focus."
Modified from Hullin et al. [5]



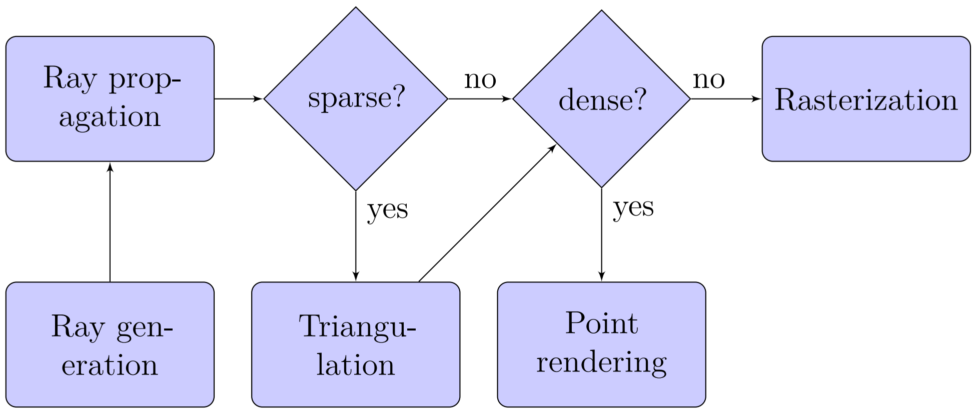
The ordering of intersections brings the benefit of more continuous results around the edges of the image, which is especially important for the sparse ray tracing technique, as discussed in section 2.3.

2.2.1 Pipeline Overview

Depending on the technique used, different parts of the pipeline are executed.

Polynomial rendering uses polynomials for the ray propagation, but is otherwise sparse.

Figure 2.4: The pipeline for rendering a ghost.



2.2.2 Ray initialization

For all rendering techniques, the rays origins are initialized at the light source. Their directions are uniformly distributed.

For rendering, the directions form a grid of equally spaced points. To make sure any errors do not correlate, for function fitting, we sample the directions and positions of rays in a uniformly random fashion. To generate data for fitting, sampling is done randomly for both the position of the light source and the direction of the ray.

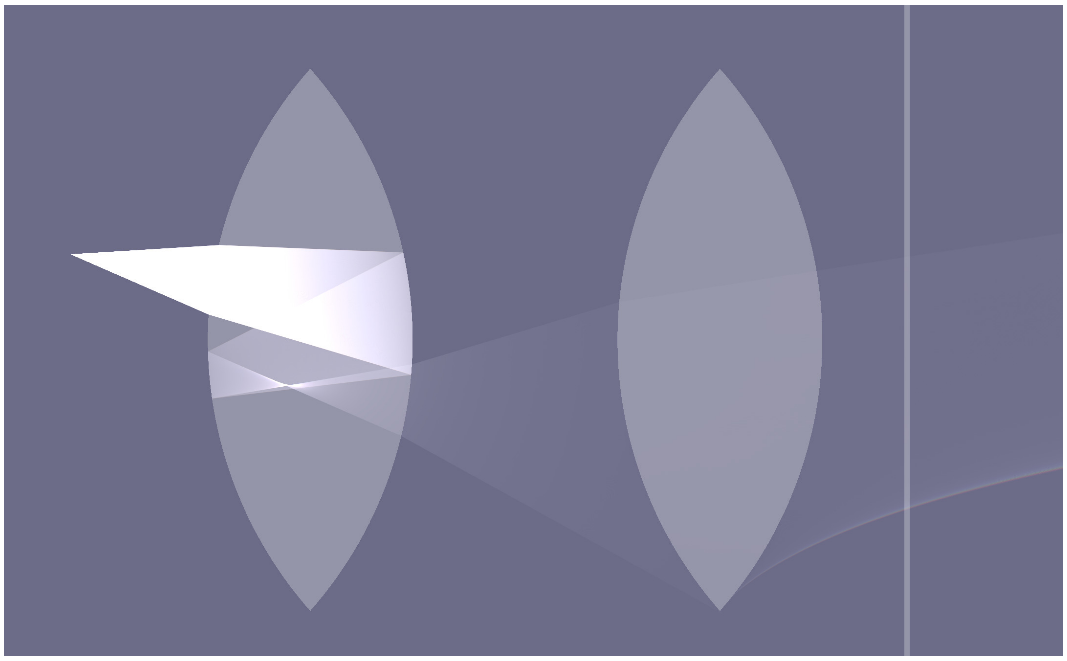
We use Planck's law to calculate the power of each ray based on the wavelength to create white light.

2.2.3 Ray propagation

Once the ray is initialized, it is traced through the lens system. This is done by intersecting the ray with every lens element in the predetermined order given by the ghost this ray belongs to.

Because all glass elements in our model are either spherical or cylindrical, the intersection with a lens element is a ray-sphere or ray-cylinder intersection, which can be computed analytically. Algorithms also exist for aspherical lenses, such as described by Joo et al. [8], but they are not implemented here.

Figure 2.5: 2d section of ray tracing for the first ghost in a two-lens system. Rendered with Polyflare.



As discussed in the introduction of this section 2.2, the order of intersections is given by the order of the elements, as this avoids discontinuities outside the physical lens element. This is only important for the sparse ray tracing technique, where rays outside the optical system are used for interpolation.

2.2.4 Wavelength to RGB color

Commonly, the wavelength of light is converted to an RGB color by first converting it to the CIE XYZ color space. Then, the XYZ color is converted to RGB by the CIE XYZ to RGB matrix.

To achieve a more realistic result, we use a lookup table from Mauer [11] generated by passing a narrow band of wavelengths to a camera sensor and observing the resulting RGB values.

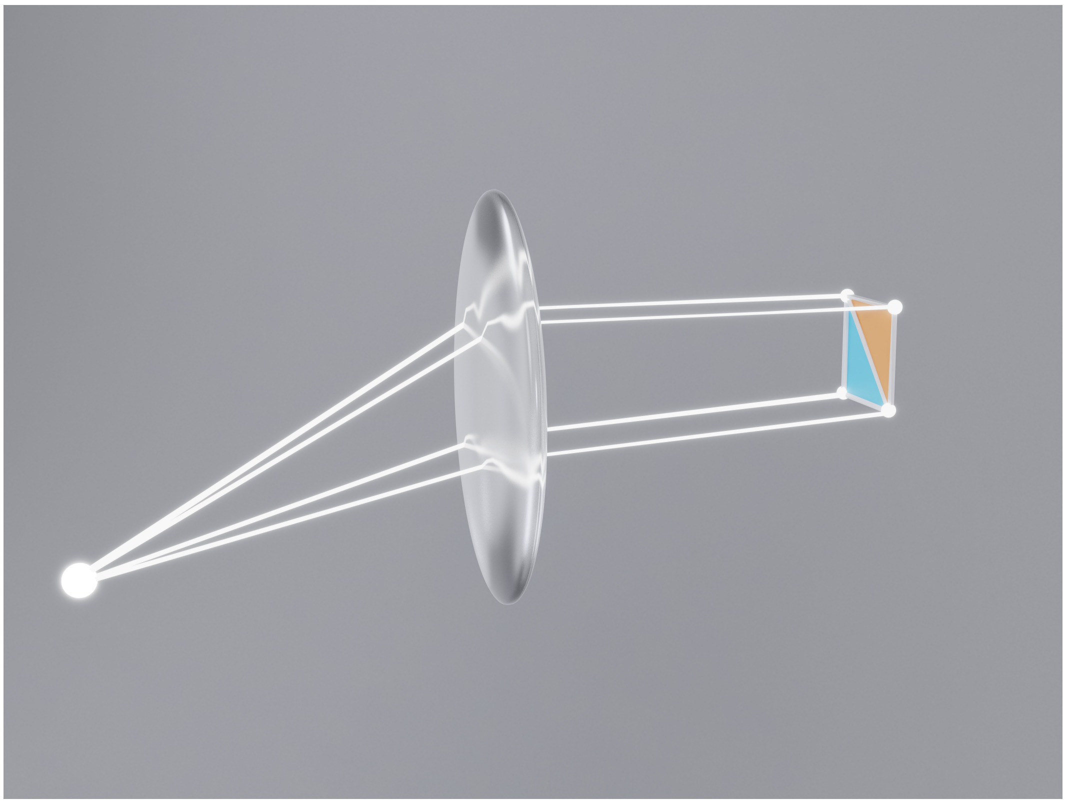
2.3 Sparse Ray Tracing

2.3.1 Previous Work

The most impactful optimization Hullin et al. introduced in “Physically-Based Real-Time Lens Flare Rendering” [5] was to use a sparse ray tracing technique. This works by using the GPU rasterization pipeline to interpolate the traced rays and gave about a 5300 times speedup over dense ray tracing. The rasterization pipeline is highly optimized and offers a huge performance boost as we will see in chapter 5.

In particular, a grid of rays is generated that is then traced through the lens system and the resulting positions and other parameters are then used to form a triangle mesh which is then rendered using a fragment shader.

Figure 2.6: Visualization of how rays are part of a triangle. With help from Olivia Mélancolie



3 Polynomials

Polynomials have many useful properties, such as being able to be fit to data easily, being smooth, being easily derived, and being cheap to evaluate. Using polynomials in computer graphics is not a new concept, as they have been used successfully previously in rendering images through lens systems by Hullin et al. [6]. However, instead of using them for lens distortion and aberrations, we apply them to rendering lens flares.

Specifically, polynomials are a set of functions constructed by adding together a set of monomials. Monomials are made up of a coefficient and a power for each variable. We use polynomials with dimension 4 of degree d, Where the first two dimensions are the position of the light source, and the other two give the direction of the light going from the light source. The wavelength is ignored here for simplicity's sake, but this would have to be taken into account for realistic results including dispersion.

$$\text{Exponents: } d_x + d_y + d_z + d_w \leq d \tag{3.1}$$

$$\text{Monomial } i\text{: } m_i = c_i \cdot x_x^{d_x^i} \cdot x_y^{d_y^i} \cdot x_z^{d_z^i} \cdot x_w^{d_w^i} \tag{3.2}$$

$$\text{Polynomial: } p(x) = \sum_{k=0}^{n} m_k \tag{3.3}$$

We call a polynomial dense, when all monomials of the given degree are present. If this is not the case, we call it sparse.

3.1 Previous Work

Hullin et al. used a "construction kit" approach for their polynomials, meaning they constructed a polynomial for each lens element and concatenated them together afterwards [6]. This method requires little preprocessing, but the error from each lens element is added to the final polynomial. These polynomials were only passingly applied to the problem of ghost rendering.

Schrade et al. improved on this by computing a polynomial of the entire lens system [17]. Additionally, they selected the most important terms of the polynomial for faster evaluation. Schrade et al. only used the polynomial for the lens distortion and aberrations, but not for the ghost rendering.

3.2 Irradiance calculation using Polynomials

We use polynomials to calculate the irradiance of the ghosts directly instead of using an additional compute step after ray tracing using the area of triangles around each vertex.

The brightness of a given pixel on the sensor is determined by the radiant flux of the light hitting the sensor. When starting with an isotropic point light source, the radiant intensity is constant, so the irradiance of a ray bundle at the sensor is the quotient of the area of the solid angle of the ray bundle at the light source and the area of the ray bundle on the sensor.

Using an anisotropic point light source, the starting intensity of each ray bundle is not constant and has to be multiplied with the quotient, that is calculated as above.

Differentially this is given by the determinant of the Jacobian of the ray tracing function. a and b are the input angles.

$$|Df| = \begin{vmatrix} \frac{\partial f_x}{\partial a} & \frac{\partial f_x}{\partial b} \\ \frac{\partial f_y}{\partial a} & \frac{\partial f_y}{\partial b} \end{vmatrix} = \frac{\partial f_x}{\partial a}\frac{\partial f_y}{\partial b} - \frac{\partial f_x}{\partial b}\frac{\partial f_y}{\partial a}$$

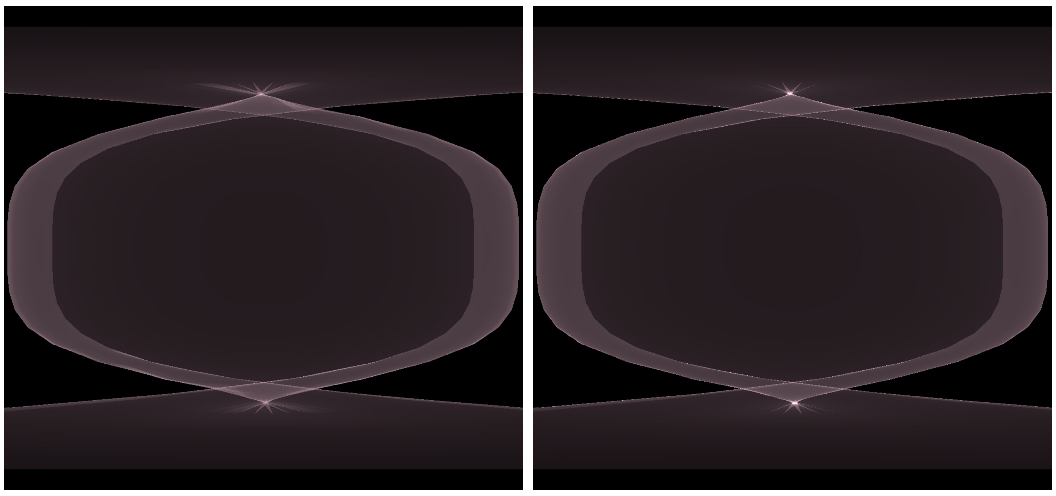
For polynomials, this can be explicitly calculated, because of the ease of derivation.

Because this equation only depends on the initialization parameters, it can be calculated before ray tracing and without taking any other vertices into account.

3.2.1 Fragment Intensity

One additional benefit of calculating the irradiance differentially is its ability to be evaluated in the fragment shader instead of a compute shader operating on vertices, and therefore per pixel instead of per vertex. This results in a slight increase in visual fidelity at a slight performance cost.

Figure 3.1: Comparison of the intensity being calculated per vertex vs per fragment



(a) Vertex (b) Fragment

3.3 Dense Polynomial Fitting

We fit a dense polynomial to the generated points using linear least squares as described by Schrade et al. [17].

Linear least squares is a method solving linear equations to data. It works as follows:

Using the multi-index $\alpha := \alpha_i | i \leq A, \sum \alpha_i < d$. The total number of terms is A. To fit Polynomials with the total number of terms A, create a matrix and the vector of data points like

$$\mathbf{X} = \begin{bmatrix} x_1^{\alpha_1} & x_1^{\alpha_2} & x_1^{\alpha_3} & \cdots & x_1^{\alpha_A} \\ x_2^{\alpha_1} & x_2^{\alpha_2} & x_2^{\alpha_3} & \cdots & x_2^{\alpha_A} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_N^{\alpha_1} & x_N^{\alpha_2} & x_N^{\alpha_3} & \cdots & x_N^{\alpha_A} \end{bmatrix} \text{ and } \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

Now, the optimal coefficients $\vec{\beta}$ are: $\vec{\beta} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\vec{y}$ [10]. We can rearrange the matrix and vector to get:

$$\begin{aligned} \vec{\beta} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\vec{y} \\ \Rightarrow (\mathbf{X}^{\mathrm{T}}\mathbf{X})\vec{\beta} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X})(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\vec{y} \\ \Rightarrow (\mathbf{X}^{\mathrm{T}}\mathbf{X})\vec{\beta} &= \mathbf{X}^{\mathrm{T}}\vec{y} \end{aligned}$$

Then solve for $\vec{\beta}$ using linear algebra.
We can then use our model of the form $\widehat{y} = \mathbf{X}\vec{\beta}$.

3.4 Sparse Polynomial Fitting

Schrade et at. [17] used sparse polynomials to evaluate polynomials of high degree quickly; they reduced the number of terms to a constant s, resulting in a sparse polynomial. This process is not as straight forward as selecting the terms with the highest coefficients, because the coefficients may correlate strongly and therefore removing some terms may cause others to grow. A specific combination of monomials could be responsible for a pattern that is otherwise missing in the resulting polynomial.

Schrade et al. [17] used a variant of orthogonal matching pursuit 3.8 for this purpose, but also saw room for improvement, which is why we propose three alternative solutions in the following sections.

3.5 Orthogonal Polynomials

Orthogonality is the property of two elements of a vector space that the dot product of the two vectors is zero. For polynomials, this dot product is commonly defined as an integral over the entire real numbers with a weight function [1].

$$\langle P_i, P_j \rangle := \int_{-\infty}^{\infty} \mathrm{P}_i(x)\mathrm{P}_j(x)\mathrm{w}(x)dx$$

Both the position and the direction of each ray are finite and easily normalized to $[-1, 1]$, so we restrict the integral to that interval and set $\mathrm{w}(x) = 1$, as is done in Legendre polynomials [1]:

$$\langle P_i, P_j \rangle := \int_{-1}^{1} P_i(x)P_j(x)dx$$

The polynomials are orthogonal if and only if:

$$\langle P_1, P_2 \rangle = 0$$

Using the above definition, an orthogonal basis can be constructed. This defines the Legendre polynomials up to a constant called the standardization. To avoid unnecessary constants however, we will use the following, more restrictive definition of orthonormal polynomials:

$$\langle P_i, P_j \rangle = \delta_{ij}$$

where δ_{ij} is the Kronecker delta:

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

This results in the standardization $P_0 = \frac{1}{\sqrt{2}}$ for the Legendre polynomials used in this thesis.

3.5.1 Benefits of Orthogonal Polynomials

Because the contribution of each polynomial in the basis is independent, orthogonal Polynomials can be fit to data independently of each other. This presents two major benefits for our application:

Fitting Fitting time scales linearly with the number of terms.

Sparsity The terms with the highest coefficient are the most important.

These benefits still exist in higher dimensions, as each base polynomial can be multiplied together to form a higher dimensional polynomial basis, which is still orthogonal.

3.5.2 Higher Dimensions

An orthogonal basis in one dimension can be used to construct an orthogonal basis in a higher dimension by multiplying the base polynomials of different dimensions together [4]. Therefore, they can be evaluated quickly using one-dimensional lookup tables.

For four dimensions, one would multiply four basis elements P_i, P_j, P_k, P_l, one for each dimension. This increases the degree d of the polynomial to the sum of the degrees of the basis polynomials. So it needs to be limited to $i + j + k + l \leq d$.

$$\text{4 dimensions: } P_{ijkl}(x) = P_i(x_1) \cdot P_j(x_2) \cdot P_k(x_3) \cdot P_l(x_4)$$

In order for these polynomials to exhibit the same benefits in higher dimensions, they must be orthogonal with respect to the dot product:

$$\begin{aligned}
&\langle P_{ijkl}, P_{mnop} \rangle \\
&:= \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} P_{ijkl}(x, y, z, w) \cdot P_{mnop}(x, y, z, w) dxdydzdw \\
&= \langle P_i, P_m \rangle \cdot \langle P_j, P_n \rangle \cdot \langle P_k, P_o \rangle \cdot \langle P_l, P_p \rangle
\end{aligned}$$

In the case where $ijkl = mnop$, the dot product is one, while if any one of the indices is different, the corresponding term is zero, and the dot product is zero.

To more efficiently evaluate each basis polynomial, we propose the use of lookup tables for each basis polynomial that are then multiplied together to form the higher dimensional polynomial. This is more efficient than evaluating each term individually, because evaluating each basis polynomial means computing a power, a multiplication, and an addition for up to d terms.

Because of the problems discussed in section 3.5.5, we did not implement this approach in our application.

3.5.3 Fitting to Data

Once all basis polynomials have been calculated, the coefficients for each can be calculated directly by interpreting the data points as a pointwise defined function and calculating the dot product of each basis polynomial with the data-function d:

$$\beta_{ijkl} = \langle P_{ijkl}, d\rangle = \sum_{j=1}^{N} P_{ijkl}(x_j) \cdot y_j$$

Because the scalar product of the basis elements are independent, the coefficients can be fit independently and therefore the fitting time scales linearly with the number of terms unlike with normal polynomials, where the fitting time scales quadratically.

3.5.4 Sparsity

Selecting the best combination of terms is a difficult problem with normal polynomials, but with orthogonal polynomials, the best combination is always the one with the highest coefficients. This is because the contribution of each basis polynomial to the error is independent of all others.

3.5.5 Problems with Orthogonal Polynomials

Unfortunately, because not all rays entering the lens make it through the lens, and which rays are discarded depends on the ghost, we were unable to construct an orthogonal basis of polynomials for efficiently rendering ghosts. Nevertheless, we still discuss different orthonormal bases of polynomials, as these may prove useful for applications, where rays are not discarded.

3.6 Legendre Polynomials

The typical standardization for Legendre polynomials is $P_n(1) = 1$, such as used for the Rodrigues formula [15].

$$\int_{-1}^{1} L_i(x)L_j(x)dx = \frac{2}{2j+1}\delta_{ij}$$

For this, there exists an explicit formula: From the Rodrigues formula, the following formula can be derived:

$$L_i(x) = 2^i \sum_{k=0}^{i} x^k \cdot \binom{i}{k}\binom{\frac{i+k-1}{2}}{i}$$

Or for each monomial:

$$m_{ik}(x) = x^k \cdot 2^i \binom{i}{k}\binom{\frac{i+k-1}{2}}{i}$$

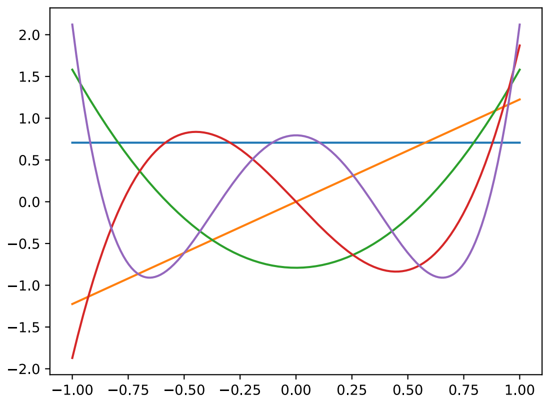
Because of our alternative standardization of Legendre polynomials:

$$\int_{-1}^{1} L_i(x)L_j(x)dx = \delta_{ij}$$

Our monomials add one additional term highlighted in yellow:

$$m_{ik}(x) = x^k \frac{2j+1}{2}\binom{i}{k}\binom{\frac{i+k-1}{2}}{i}$$

Figure 3.2: Legendre polynomials



3.7 Gram Polynomials

Because Legendre Polynomials are defined over the continuous interval $[-1, 1]$, they are not orthogonal on a grid with a finite number of points.

For this reason, gram polynomials are more useful, which are orthonormal on a grid with a given finite number of points, where the inner product is defined as the sum over the points, instead of the integral [2].

$$\langle G_i, G_j \rangle = \sum_{k=0}^{N-1} G_i\left(x_k\right) G_j\left(x_k\right), \quad x_k = \frac{2k-1}{N}$$

So the definition of the Gram polynomials is:

$$\langle G_i, G_j \rangle = \delta_{ij}$$

Which can be constructed using the Gram-Schmidt process beginning with the polynomials

$$P_i(x) = x^{i-1}$$

When the number of samples on the grid is large, the error of Legendre polynomials approaches zero and the Gram polynomials approach the Legendre polynomials.

Figure 3.3: $\sum_{i=1}^{d} \sum_{j=1}^{d} \left| \left(\sum_{p}^{N} L_i \cdot L_j \frac{1}{N} \right) - \delta_{ij} \right|$

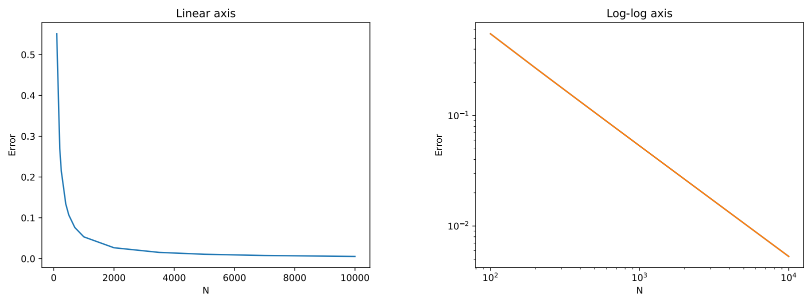
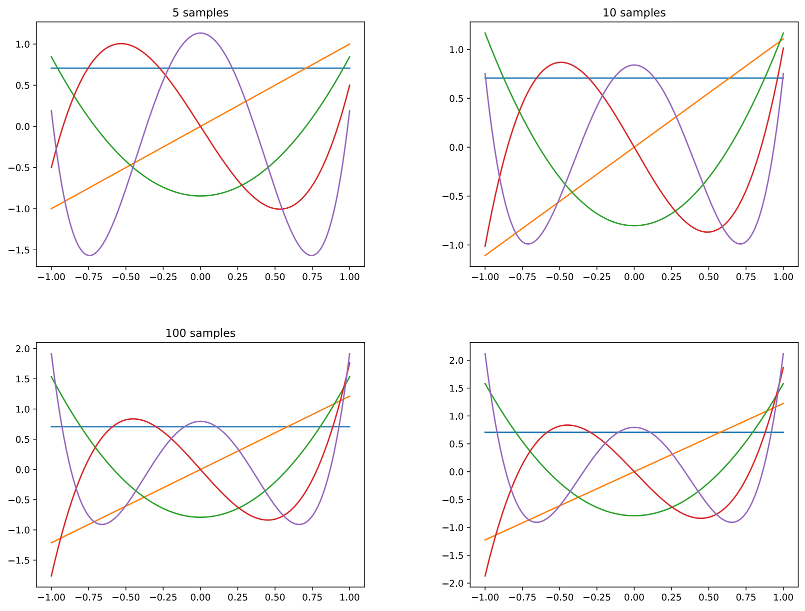


Figure 3.4: Gram polynomials on different number of samples compared to Legendre polynomials (bottom right)



3.8 Orthogonal Matching Pursuit

Previously, a variant of Orthogonal Matching Pursuit was used by Schrade et al. [17] to fit sparse polynomials to a set of data points, so this is the first algorithm we implemented.

Orthogonal Matching Pursuit is a greedy algorithm that starts with an empty Polynomial and then progressively adds the term with the minimum error until the number of requested terms is reached. Adding replacement to this algorithm means that instead of stopping when the number of terms is reached, the algorithm will continue to replace terms with the one minimizing the error until the number of terms is reached. The term that is replaced by the new one is determined by calculating the error of the polynomial with each of the existing terms replaced and choosing the one with the minimum error.

We also implemented two variations of Orthogonal Matching Pursuit:

No replacement This algorithm is the same as the original, but it stops when the number of terms is reached instead of replacing terms.

No fitting This algorithm is the same as the original, but it retains the same coefficients from the dense polynomial.

As discussed in the conclusion of their paper, they see room for future improvement over orthogonal matching pursuit. We will see in chapter 5, that especially for modeling ghosts, OMP is not a good choice.

3.9 Simulated Annealing

Annealing is the process of a physical system cooling down and minimizing its thermodynamic free energy. This process can be seen as an analogy for the simulated annealing algorithm, where a digital system goes through the same "cooling" process.

This is modeled by an objective function, an acceptance function, and a concept of neighborhood. The objective function is the equivalent of the thermodynamic free energy of the system, while the neighborhood is how steps can be taken from the current state to the next state. How likely a step is to be accepted is determined by the acceptance function.

Now the basic idea is that the system is simulated by taking steps in the neighborhood and evaluating the objective function on the new state. How large the step is, is determined by the temperature, where a larger temperature means a larger step size. Each step the temperature is decreased until it is zero and the algorithm stops.

3.9.1 Simulated Annealing for Sparse Polynomials

Applying simulated annealing to the sparse polynomial fitting problem, the space of possible solutions is every combination of the required number of terms from the original dense polynomial.

Taking a step means exchanging terms in the sparse polynomial with an equal number of terms that are not in the sparse polynomial.

Here, the objective is function is the error of the current sparse polynomial fit to the data points. It is very important to fit after each step, as this improves the quality of the fit significantly, as will be seen in section 5.3.2. We define our acceptance function as the following:

$$p_{\text{accept}}(\Delta E) = \begin{cases} e^{-\frac{\Delta E * 100}{t}} & \text{if} \Delta E > 0 \\ 1 & \text{if} \Delta E < 0 \end{cases}$$

where t is the temperature, which starts at one and linearly decreases to zero.

How many steps to take is calculated using the following formula:

$$\text{num\_steps} = \max(t^3 \cdot \text{num\_terms}, 1)$$

Both the acceptance function and the number of steps to take are subject to further tuning in the future.

Thanks to the low sensitivity to uncorrelated noise, we were able to speed up the execution by selecting a random subset of the points for each iteration without reducing the quality of the fit by much, as we will show in section 5.3.2.

4 Implementation

This chapter describes the implementation of Polyflare, consisting of the library and the applications.

We will start with an overview, go into the chosen languages and libraries, and finally discuss each component of Polyflare.

4.1 Overview

The Polyflare workspace comprises three Rust crates:

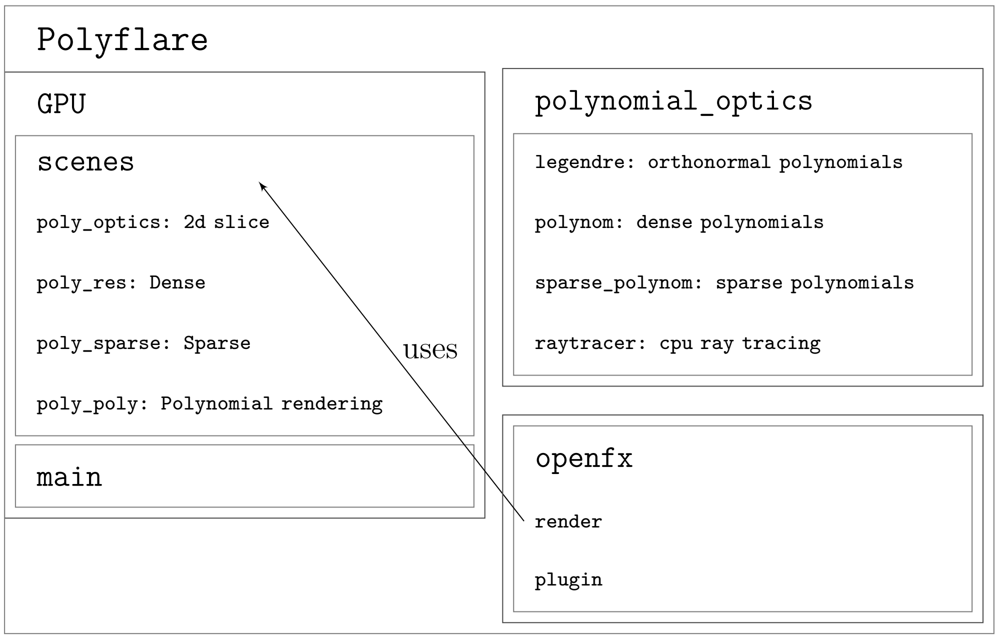
`GPU` is a GPU-based implementation of the ray tracing algorithms.

`polynomial_optics` is the CPU part of the ray tracing and polynomial algorithms.

`openfx` contains the plugin for OpenEffects.

Each of these crates is made up of several modules, each made up of several files. Files contain structs, which are like classes in C++, but can't inherit from other structs. They will be discussed in their relevant sections.

Figure 4.1: The Polyflare workspace


Polyflare
GPU
scenes
poly_optics: 2d slice
poly_res: Dense
poly_sparse: Sparse
poly_poly: Polynomial rendering
main
uses
polynomial_optics
legendre: orthonormal polynomials
polynom: dense polynomials
sparse_polynom: sparse polynomials
raytracer: cpu ray tracing
openfx
render
plugin

4.2 Rust

For the CPU implementation of Polyflare, we chose Rust [20] over the more traditional C++ for the following reasons:

Memory Safety Rust uses a borrow checker to ensure that all values are either shared and immutable, or non-shared and mutable. Therefore, a value can not be modified while it is in use somewhere else.

Concurrency Is easy to work with in Rust and through the borrow checker, concurrency is guaranteed to be safe.

Safety The two concepts above and other parts of the language work together to avoid undefined behavior and hard to explain crashes, which have been a source of trouble in previous C++ applications by the author.

Compiler Warnings and Errors, in the eyes of the author, are more helpful and easily understood than those provided by C++.

Iterators are an important part of the language, and are used in many places in Polyflare.

Builtin Package Management allows for many libraries to be used without having to install them manually and Polyflare makes liberal use of these.

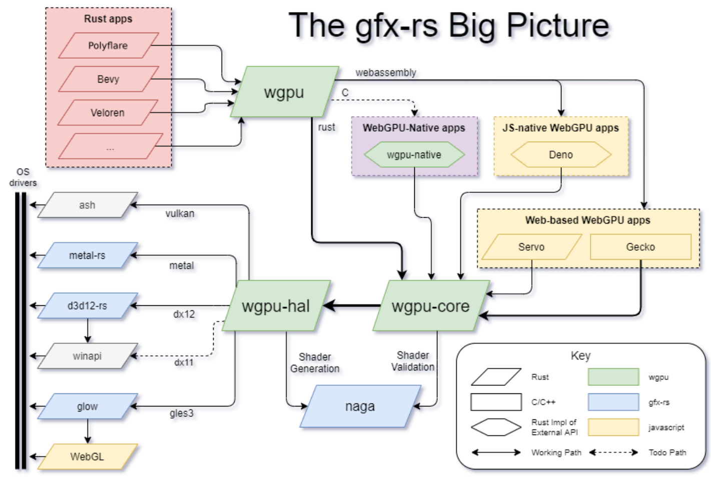
4.3 WGPU

For the GPU implementation `wgpu` was chosen.

"`wgpu` is a cross-platform, safe, pure-rust graphics API. It runs natively on Vulkan, Metal, D3D12, D3D11, and OpenGLES; and on top of WebGPU or WebGL2 on wasm." [16]

Using `wgpu` allows us to run Polyflare on all major platforms without much effort. In particular, Polyflare is tested to be running on Windows, macOS, and Linux using D3D12 or Vulkan, Metal, and Vulkan or OpenGL respectively.

Figure 4.2: Visualization of the components in the gfx-rs ecosystem. Edited from [3].



4.4 Open Effects

We chose to write a plugin for the "Open Effects API" or "OFX" [19].

"The Open Effects API allows plugins written for it to be used in multiple 'hosts', such as The Foundry Nuke, Natron, Assimilate Scratch, Sony Vegas, or FilmLight Baselight." [19]

To write the plugin for OFX in Rust one needs an intermediate layer. This is provided in our implementation by "OpenFX bindings" [12].

4.5 Ray tracing

We need to implement the ray tracing algorithm twice. Once for the CPU and once for the GPU. These are mostly the same, yet have some differences. They will be highlighted in their respective sections 4.6.1 and 4.7.2.

4.5.1 Ray initialization

On the CPU, we implemented both random and grid ray initialization, whereas on the GPU, we only implemented grid ray initialization. For the grid initialization, `ray_num_x`, `ray_num_y` determine which ray on the grid is being initialized, while `sqrt_num` is the number of rays in each direction and `width` is the width of the grid. For random sampling, only the width and number of rays is needed.

Various parameters, such as `width`, `sqrt_num` and the center direction of the grid are given to the GPU as uniforms, which are then used in the shader.

Source Code 4.1: Generating Rays

```
var dir = posParams.init.d;
dir.x = dir.x + (ray_num_x / f32(sqrt_num - u32(1))
                 * width - width / 2.);
dir.y = dir.y + (ray_num_y / f32(sqrt_num - u32(1))
                 * width - width / 2.);
dir = normalize(dir);
var ray = Ray(posParams.init.o, posParams.init.wavelength,
              dir, str_from_wavelen(posParams.init.wavelength),
              vec2<f32>(0., 0.), vec2<f32>(0., 0.));
```

`str_from_wavelen(wavelength: f32) -> f32` is responsible for converting the wavelength from micrometers to meters needed for Planck's law and bringing it to reasonable values for the rest of our algorithm.

The temperature of the incoming light is a user-controlled parameter.

Listing 4.1: Calculating the irradiance for each wavelength

```rust
fn planck(wavelen: f32, temp: f32) -> f32 {
    let b = 1.380649e-23; // Boltzmann constant
    let e = 2.718281828459045;
    let hc = 1.9864458571489286e-25; // h*c
    let hcc2 = 1.1910429723971884e-16; // 2*h*c^2
    return hcc2
        / (pow(wavelen, 5.))
        / (pow(e, (hc) / (wavelen * b * temp)) - 1.) / 1.e12;
}

fn str_from_wavelen(wavelen: f32) -> f32 {
    return planck(wavelen / 1000000., 3000.) * 10.;
}
```

Equation 4.1: Planck's Law

$$\frac{2h\nu^3}{c^2}\frac{1}{e^{\frac{h\nu}{k_BT}}-1} \tag{4.1}$$

Because the constants are quite extreme in value for f32 and using them directly would result in a loss of precision, we use pre-calculated values for $h * c$ and $2 * h * c^2$ to avoid the loss of precision. This has the added benefit of reducing the number of operations needed and therefore increasing the speed of ray generation.

4.5.2 Ray propagation

The order of intersections in the lens for a given ghost is fully described by two indices i and j. The ray is intersected with the lens elements until the j-th intersection, at which point the ray is reflected, then it is refracted backwards until the i-th intersection, at which point the ray is reflected again to be refracted through all the lens elements after that point to the sensor.

For the ghost (i, j), where $i < j$, we iterate through all elements other than the j^{th} element, at which we reflect for the first time, then trace backwards, reflect again at the i^{th} element, and finally trace forwards until the end.

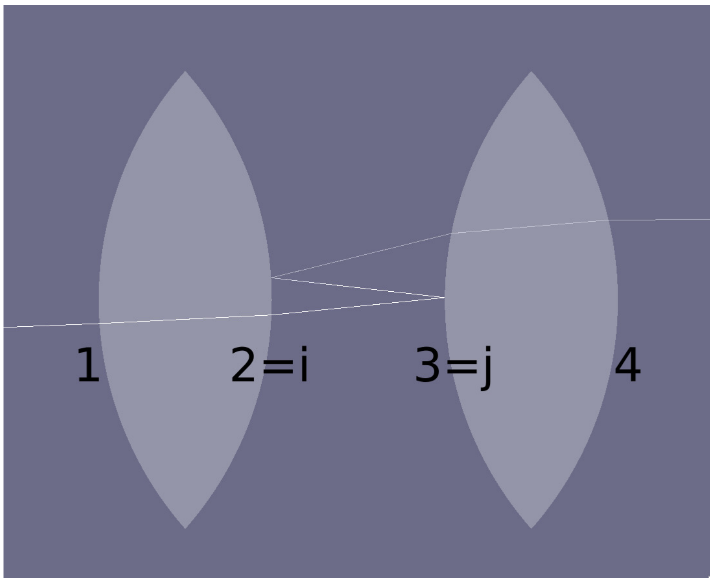
Algorithm 1 Ray propagation

```
for (num_element, element) in elements.iter().enumerate() {
    if num_element != j {
        ray.propagate(element);
    } else {
        // reflect at the first element,
        // which is further down the optical path
        ray.reflect(element);

        // propagate backwards through system
        // until the second reflection
        for k in (i + 1..j).rev() {
            ray.propagate(&elements[k]);
        }
        ray.reflect(&elements[i]);

        // then propagate forwards through
        // the rest of the system
        for k in i + 1..=j {
            ray.propagate(&elements[k]);
        }
    }
}
```

Figure 4.3: Visualization of the ray propagation.
Rendered with Polyflare



4.5.3 Wavelength to RGB conversion

The lookup tables need to be slightly post-processed to be used, because the white-balance is not taken into account yet. This white-balancing is left to the user as creative control.

Listing 4.2: Wavelength to RGB conversion

```
/// Wavelength is given in micrometers
fn lookup_rgb(wavelength: f32) -> vec3<f32> {
    let lower_index = u32((wavelength -
 ↪  sensor.measuremens[0].wavelength / 1000.0) * 100.0);
    let factor = (wavelength % 0.1) * 10.0;
    return sensor.measuremens[lower_index].rgb * (1.0 - factor)
     + sensor.measuremens[lower_index + u32(1)].rgb * (factor);
}
```

4.5.4 Drawing

For both sparse and dense ray tracing, all ghosts and wavelengths need to be added per pixel. To reduce rounding errors, we use an intermediate `f16` texture for the blending, which is then converted in another shader to the final RGB texture.

4.6 Polynomial Optics

The `polynomial_optics` library is used to implement the CPU part of the ray tracing and polynomial algorithms. It also contains a binary, which is used to test parts of the library.

4.6.1 Ray tracer

The ray tracer module contains the structs and functions needed for ray tracing. The two main structs are `Ray` and `Element`.

`Ray` represents a single ray, which can be propagated though or reflected by an `Element`.

As discussed in chapter 2, each ray contains an origin and a direction, which are both of type `cgmath::Vector3<f64>` from the `cgmath` library, which implements useful functions, like addition, multiplication, and dot product.

The `Element` struct represents an element in the lens. It contains a position, a radius, and the `Properties` enum. For the aperture, this enum contains a single `u32` integer representing the number of blades in the aperture. For an Element made of Glass, it contains a `Glass` struct. This struct contains a `Sellmeier` struct, which contains the coefficients of the Sellmeier equation and a `QuarterWaveCoating`.

We nest the structs for individual properties within each other to be able to implement methods on each. For example, getting the index of refraction (listing 4.3) of a glass is implemented by the `Sellmeier` struct.

Listing 4.3: Sellmeier

```rust
impl Sellmeier {
    pub fn ior(&self, wavelength: f64) -> f64 {
        let wavelength_sq = wavelength * wavelength;
        let mut n_sq = 1.;
        for i in 0..3 {
            n_sq += (self.b[i] * wavelength_sq) / (wavelength_sq -
↪   self.c[i]);
        }
        n_sq.sqrt()
    }
}
```

All different Sellmeier indices, obtained from LaCroix Precision Optics [9], are saved in a `csv` file and read by a function in `Sellmeier`.

Multiple `Elements` combine into a `Lens`, where the ray tracing methods are implemented.

```rust
pub struct Lens {
    pub elements: Vec<Element>,
    pub sensor_dist: f64,
}
```

These `Lenses` can also be saved and loaded to a file in a human-readable format using methods on `Lens`, which use the `serde` and `ron` libraries internally.

4.6.2 Polynom

Dense polynomials are represented using the `Polynom4d<N>` struct. It is generic over the type of the coefficients, which is `N`, meaning any numeric type may be used for the coefficients and input variables.

```rust
pub struct Polynom4d<N> {
    pub coefficients: Vec<N>,
    degree: usize,
}
```

Because the polynomial is dense with a known degree, the coefficients can be stored without the need to store the exponents for each, as these can be calculated from the index into the array.

This step is quite delicate, as we needed to increase bit-depth from 32 to 64 and switch from a naive Rust implementation to an external library for it to converge. Intel-MKL[7] was chosen for this step.

We fit the dense polynomial by creating a column vector of the desired output values, and a matrix of the input values evaluated for each term and then handing these to the `mathru` library to solve. Initially, we used 32 bit floating point numbers, but these did not converge until switching to 64 bit. `mathru` has multiple back-ends, including a pure Rust implementation, and the Math Kernel Library allowing for a trade-off between performance and fewer dependencies.

Sparse from Dense

The functions for creating a sparse polynomial from a dense polynomial also reside in `polynom`.

`get_sparse` implements OMP from section 3.8 and our variants of it.

`simulated_annealing` implements our simulated annealing algorithm for polynomials.

4.6.3 Sparse Polynom

Unlike for dense polynomials, for sparse polynomials, it is not sufficient to store the coefficients, because it is not clear which coefficients correlate to which exponents. Therefore, we introduce the `Monomial<N>` struct, which stores the coefficient and the exponent. A list of these monomials is then used to create a `Polynomial<N>`.

Thanks to the use of iterators, we were able to multithread the most expensive operations in the sparse polynomial creation by using the Rayon `.par_iter()` instead of the standard Rust `.iter()`.

Source Code 4.2: Calculating the approximate error of a sparse polynomial

```rust
pub fn approx_error(
    &self,
    points: &[(f64, f64, f64, f64, f64)],
    num_samples: usize,
    offset: usize,
) -> f64 {
    (points[offset..offset + num_samples]
        .par_iter()
        .map(|p| (p.4 - self.eval([p.0, p.1, p.2, p.3])).powi(2))
        .sum::<f64>()
        / num_samples as f64)
        .sqrt()
}
```

4.7 GPU

All code interacting with the graphics card is contained in the `gpu` crate.

As outlined in figure 4.1, this crate contains the main application the libraries necessary for writing it or any other application based on Polyflare.

4.7.1 Main

The application is responsible for the user interaction and the main loop. We read optional command-line arguments for selecting the graphics API, GPU, and whether to enable VSync and provide a GUI (figure 5.4) for the user to interact with. The GUI is implemented with the help of the `imgui` and `winit` libraries.

4.7.2 Scenes

Most of the code for interacting with the GPU is in one of the four scenes. Each of these scenes is responsible for drawing one of the components of the application.

They each contain a struct containing the state necessary for drawing, a function for initialization, and a function for drawing. Included in the state are the `RenderPipeline` and metadata for each shader file used. Using the metadata, we can check each frame if the shader has changed, and if so, we can update the pipeline. This enables rapid development of the shader, without having to recompile and restart the entire application. Should the file not be present at startup, we fall back to the shader included at compile time.

poly_optics is responsible for the 2d slice through the entire scene including rays passing through the lens. We already saw this in figure 2.5.

poly_res contains the dense ray tracing code.

poly_tri is the sparse ray tracing scene. The irradiance calculation in this scene is done in a separate compute shader, where it is calculated per vertex from the average area of each neighboring triangle.

poly_poly implements the polynomial drawing.

4.8 OpenFX

The plugin for OpenEffects is in the `openfx` crate. We only tested the plugin on Natron on Linux, but it should work on all platforms supported by `ofx-rs`.

Although the `ofx-rs` is “Not ready for production” [12], and a plugin written in it will crash when run on multiple threads, it was possible to work around the problem by disabling multithreading in the host application.

Writing the plugin in Rust allowed us to re-use the code from section 4.5 for rendering, and we just need to write the interface to the host application. The rendered image is then copied over to the CPU using the same code as is used for writing it to a PNG image described in section 4. All updates to the configuration by the user are immediately reflected in the plugin, and the image is updated accordingly.

Because “OpenFX bindings” only provides a small subset of the input options of OFX and for example drop down menus are not available. So instead, we use sliders for nearly everything, like lens selection which we would have preferred to be a drop-down menu. This results in the user interface not being very intuitive, but it is still possible to use the plugin and render animations.

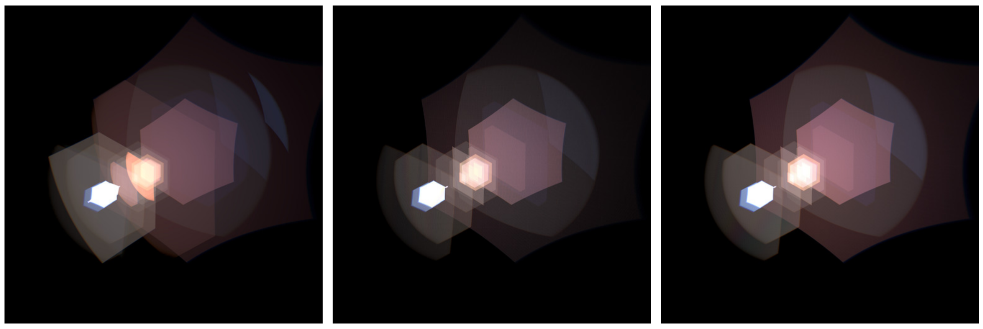
5 Results

All performance results were obtained on an Intel Core i7-8700K at 4.30GHz, with an NVIDIA GTX 1070 Ti graphics card.

5.1 Sparse Ray tracing

Through sparsely ray tracing, we can retain almost all visual qualities of dense ray tracing, but gain a considerable speedup. Figure 5.1 shows comparisons of the two methods and different number samples for dense ray tracing.

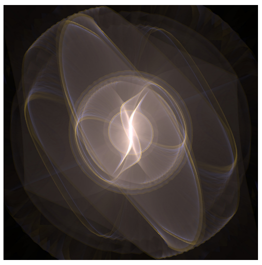
Figure 5.1: Comparison of sparse and dense ray tracing on a tessar lens rendering a 8192x8192 image

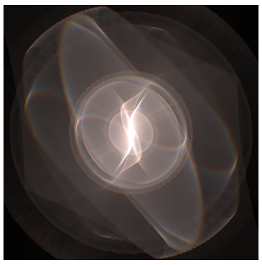


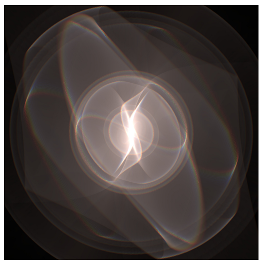
(a) Sparse (4.5s) (b) Dense (50s) (c) Dense (500s)

Both of our ray tracing implementations also support anamorphic lenses, an example of which is shown in Figure 5.2.

Figure 5.2: Comparison of sparse quality settings and dense ray tracing on a fictional anamorphic lens rendering a 768x768 image.


(a) Low Quality Sparse (16ms)


(b) Medium Quality Sparse (77ms)


(c) High Quality Sparse (200ms)

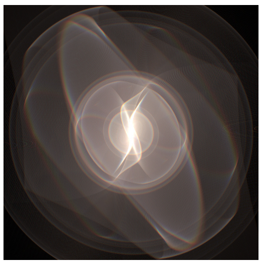
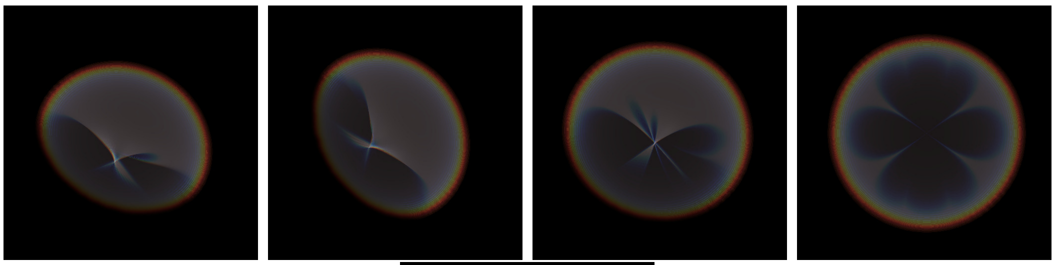
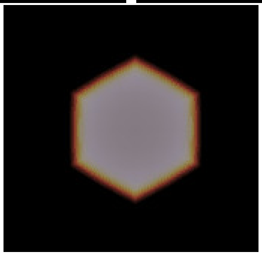

(d) Medium Quality Dense (9s)

Figure 5.3: Different Views of Ghost 15 from the Tessar Lens




(a) reduced aperture size

5.2 Applications

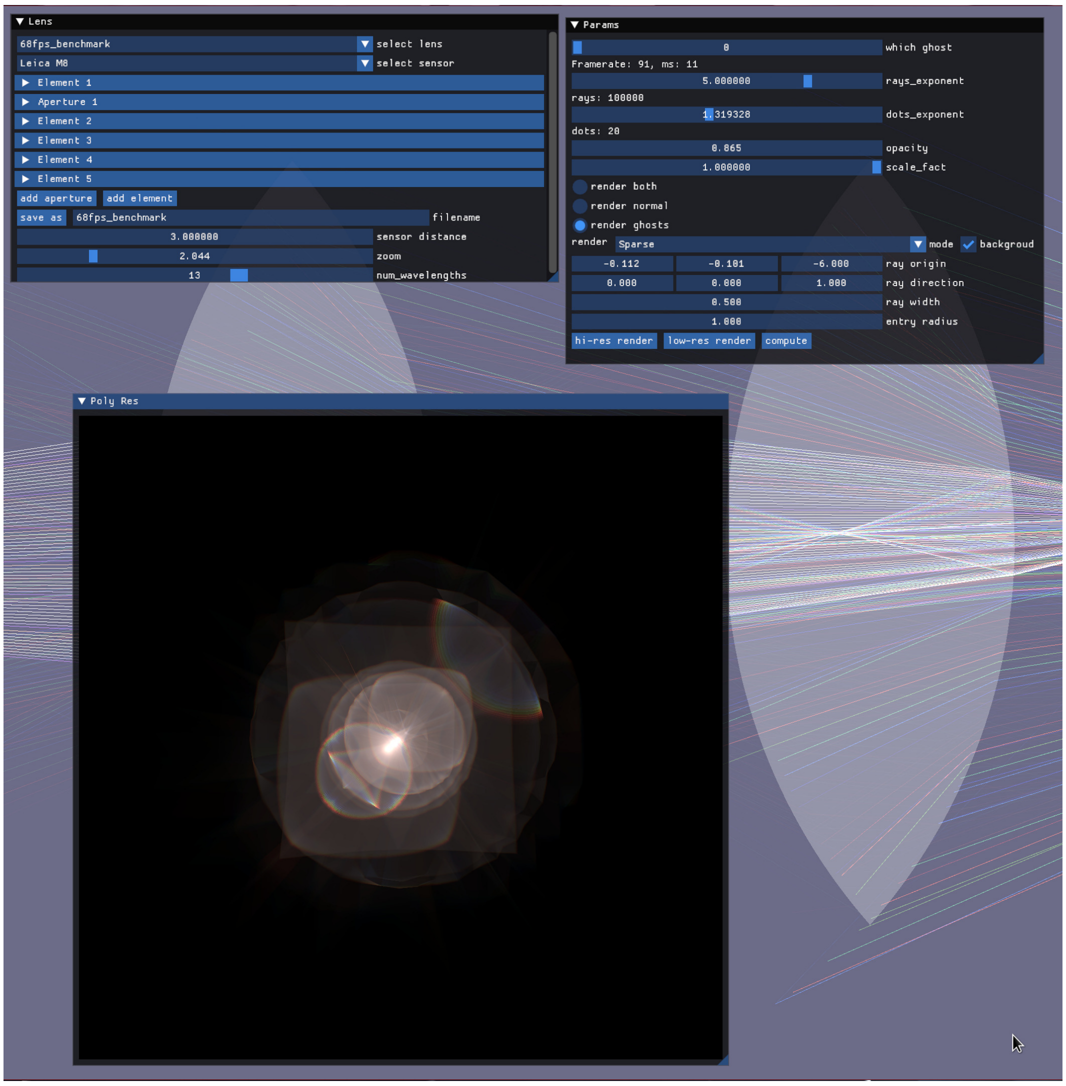
Polyflare comes with two applications, the first is the main application, while the second is an OpenEffects plugin.

Our main application (figure 5.4) has all features of the Polyflare library, including interactive dense, sparse and polynomial ray tracing, lens creation, lens preview, selecting individual ghosts and saving renders. It allows for the creation of lenses with interactive preview of the lens and resulting ghosts.

The OpenEffects plugin (figure 5.5) shares all persistent data with the main application, meaning lenses that are created in the main application are also available in the OpenEffects plugin.

Because of the modular design of the Polyflare library, the rendering component of the OpenEffects plugin was implemented within in a few hours, with the entire plugin being done in a single week.

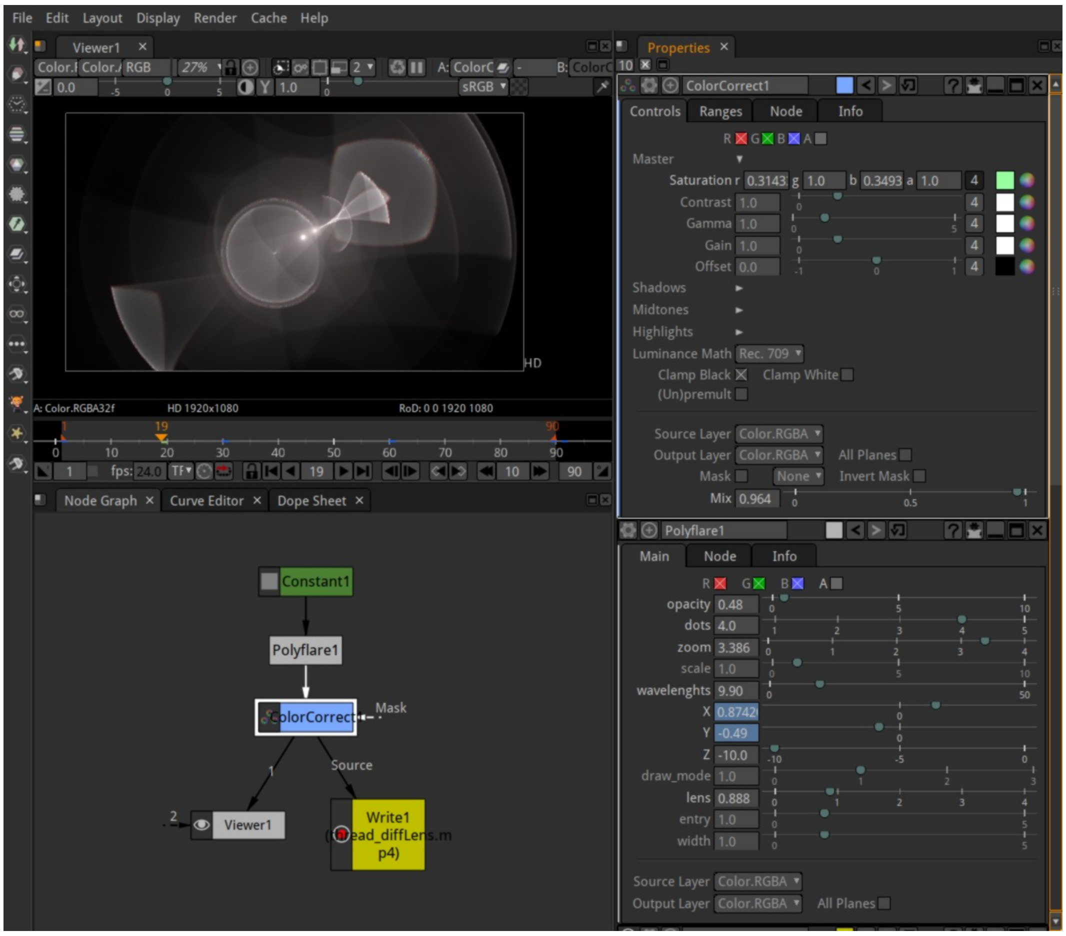
Figure 5.4: Using the Development Application



```
68fps_benchmark      select lens
Leica M8             select sensor
68fps_benchmark      filename
3.000000             sensor distance
2.044                zoom
13                   num_wavelengths
```

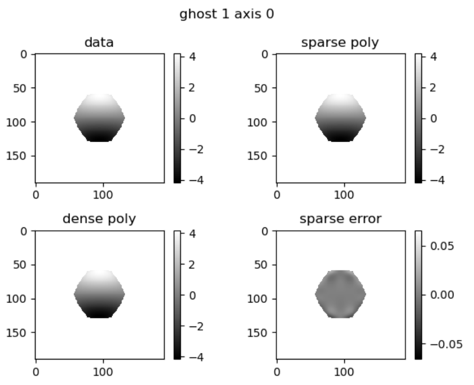
```
0                    which ghost
Framerate: 91, ms: 11
5.000000             rays_exponent
rays: 100000
1.319328             dots_exponent
dots: 20
0.865                opacity
1.000000             scale_fact
render both
render normal
● render ghosts
render Sparse        mode   ✓ backgroud
-0.112   -0.101   -6.000   ray origin
0.000    0.000    1.000    ray direction
0.500                ray width
1.000                entry radius
```

Figure 5.5: Using the Open Effects Plugin in Natron

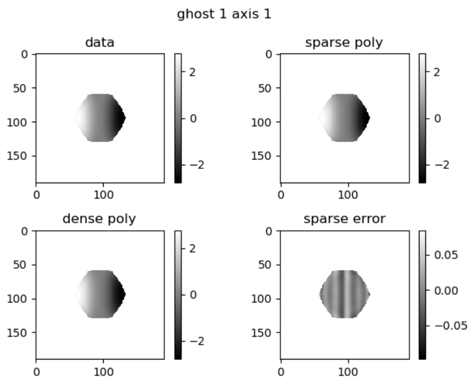


5.3 Polynomials

Figure 5.6: Overview of polynomial error



(a) Ghost 1, x-axis



(b) Ghost 1, y-axis

5.3.1 Dense Polynomials

Using a degree of 10 with 11673 samples, our dense polynomial fitting achieves an error of about 0.002368 using grid sampling and 0.001752 using random sampling on the x-axis of the first ghost of our example lens. This is equivalent to about two pixels on a 1080p image.

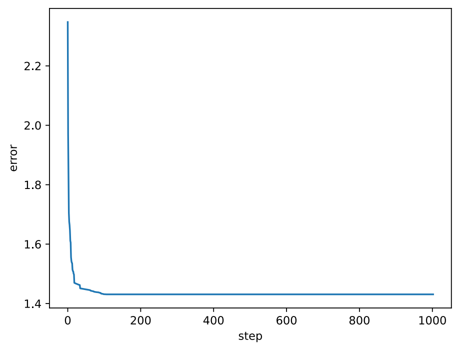
5.3.2 Sparsifying Polynomials

Different Algorithms

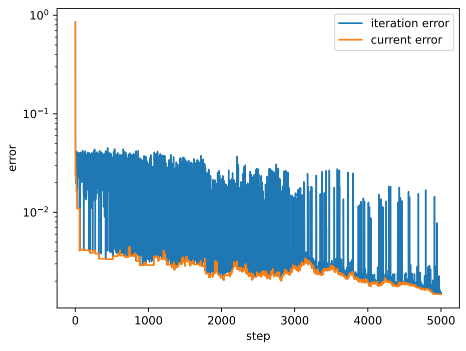
We compare orthogonal matching pursuit and simulated annealing starting with the same dense polynomial of degree 10 and a total of 1001 terms. It has an average error of 0.001614. The sparse polynomials retain 100 of these terms.

Algorithm	OMP with replacement	OMP	Simulated Annealing
∅ Time	32 min 45 s	4 min 44 s	4 min 40 s
∅ Error	1.315	1.078	0.06954
Std	0.326	0.348	0.0120

Figure 5.7: Convergence Behavior of Sparsifying Algorithms

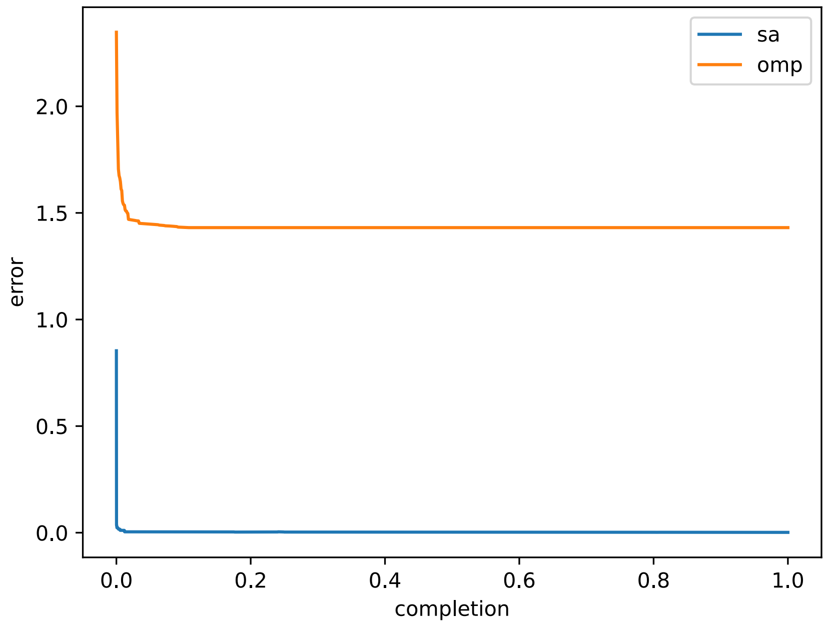


(a) OMP



(b) SA

Figure 5.8: Orthogonal Matching Pursuit vs. Simulated Annealing



(a) linear axis

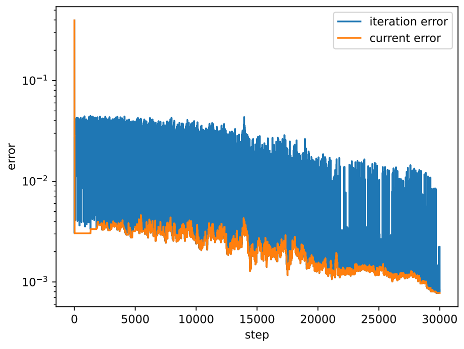
Simulated Annealing

Simulated annealing has two main tunable parameters: the number of samples that get taken into account and the number of iterations. The runtime of the algorithm scales roughly linearly with both of these parameters.

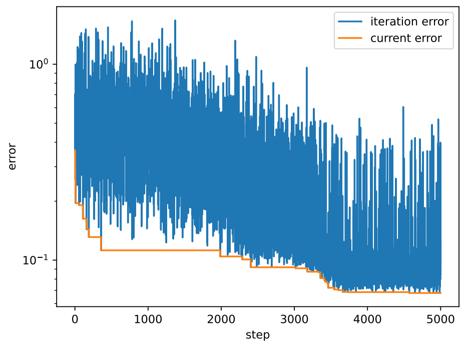
Samples	Iterations	Runtime	Error
1000	5000	139 s	0.0667
1000	10000	275 s	0.0538
2000	5000	275 s	0.0511
10000	5000	1450 s	0.0511
2000	30000	1656 s	0.0377

Taking advantage of multithreading for calculating the error and fitting, the runtime of the algorithm scales roughly linearly with the number of threads. For one thread, the runtime is 117s seconds, while with six threads, the runtime is 24.7 seconds, a speedup of 4.7.

Figure 5.9: Simulated Annealing Convergence Behavior



(a) 30000 iterations, 2000 samples

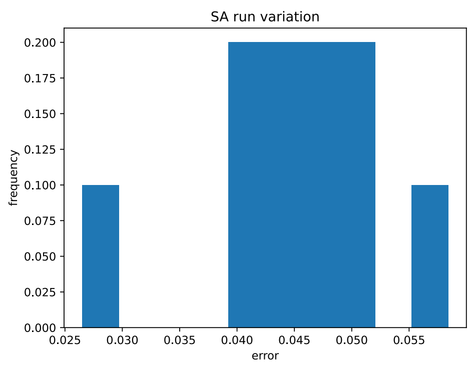


(b) No Fit

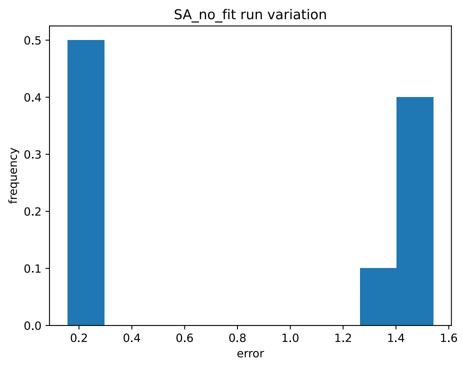
With the number of samples set to 30000 the error from inside the algorithm doesn't correspond to the actual error. Although the actual error is still smaller than with fewer samples.

As Simulated Annealing is a stochastic algorithm, the error is not always the same. Over ten runs with 2000 samples and 30000 iterations each, the average error is 0.04557 with a standard deviation of 0.007845. A histogram for this test is shown in figure 5.10.

Figure 5.10: Simulated Annealing Run Variation



(a) 30000 iterations, 2000 samples



(b) Not fitting
20000 iterations, 10000 samples

Simulated Annealing without Fitting

For 2000 samples and 5000 iterations, skipping the fitting step in simulated annealing reduces the runtime from 280s to 12s, but also increases the error to roughly 1.3. When the number of iterations and samples is set to match the runtime of the algorithm with fitting, the error is 0.82 with a standard deviation of 0.63.

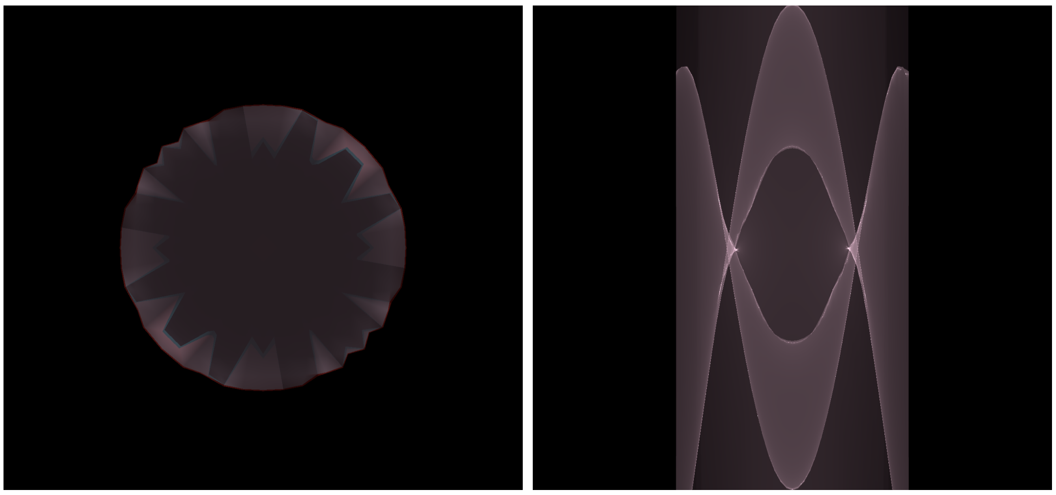
5.3.3 Rendering using Polynomials

With the average error of the sparse polynomials at about six pixels on a 1k image, one would expect the resulting image to be visually very similar to the original. However, the rendering is not anywhere close to that. Instead, we see a completely different shape in figure 5.11.

This is likely due to an implementation error in the polynomial evaluation on the GPU, that we were unable to find in the limited timeframe of the thesis.

Nonetheless, the irradiance is still calculated correctly for any given polynomial.

Figure 5.11: Sparse and Polynomial rendering of the same ghost



(a) Sparse rendering (b) Polynomial rendering

6 Discussion and Future Work

6.1 Polynomials

We presented simulated annealing for polynomial sparsification, a novel technique for isolating the most important terms in a dense polynomial, which provides a significant improvement in error for modeling ghosts, while being as fast or faster than the previously used orthogonal matching pursuit technique with and without replacement.

We were able to reduce the error from 1.5 using orthogonal matching pursuit with replacement to 0.0377 using simulated annealing, or about 1% positional deviation from the original image, which should result in a visually similar image. Unfortunately, we were unable to test this, as our implementation of the rendering algorithm does not result in a correct image. The irradiance calculation on the other hand does work correctly, although it suffers from severe aliasing. This aliasing could eventually be addressed by using filtering methods, that take advantage of the fact that the derivative is easily calculated analytically. Further improvement for simulated annealing could be achieved by tuning the parameters of the algorithm.

A basis of orthonormal polynomials proved unsuitable for modeling ghosts, as no appropriate such basis could be found when discarding rays, which is necessary for modeling ghosts. Nonetheless, orthonormal polynomials seem to be a good choice for sparse polynomials, as the most difficult and computationally expensive step, selecting the best combination of terms, becomes trivial using orthonormal polynomials.

In the future, polynomials could also be used for adaptive ray distribution.

6.2 Rendering

To our knowledge, we provide the first public implementation of real-time physically based rendering of lens flares.

Our implementation has many parameters, which allow for a trade-off between quality and performance. It is however somewhat basic and does not provide more advanced features, such as diffraction, filtering or different parameters per ghost. Many of these features have however been considered while designing the architecture, so that they could easily be implemented in the future.

Our novel technique for using polynomials for computing the irradiance at each point has several advantages over computing the irradiance from the area formed by each ray bundle. First, it is much simpler to implement, as each ray can be considered independently. Second, it improves the quality of the image, as the irradiance is computed at each ray directly, instead of as the area of the ray bundle. Third, the visual fidelity of the image can be improved further by calculating the irradiance of each fragment, similar to the visual improvement Phong shading provides over Gouraud shading. And finally, the derivative of the irradiance can be computed analytically, which could improve filtering in the future.

The last point is especially important, as the major downside of calculating the irradiance at each point is significant aliasing.

Our library proved flexible enough to quickly develop a new application using it. It already provides both options for high quality and high performance rendering, although many optimizations and features are still missing. Possible improvements include

Optimized Wavelength Sampling Our current implementation samples the wavelengths uniformly, resulting in color shifts when changing the number of wavelengths. This could for example be fixed by using a lookup table of manually selected wavelengths.

Wave-Optics We focused on rendering ghosts, but diffraction at the aperture results in two noticeable artifacts. The first one is a star burst that appears over the light source and is easily added as another additive element over the ghosts. The second artifact is the ringing pattern around each ghost and has to be more deeply integrated into the pipeline, but this can be done by replacing a single function in our drawing code.

Filtering or Super Sampling Both the sparse and dense ray tracing techniques suffer from a high amount of aliasing. This could be addressed by using a filter or super sampling.

Adaptive Quality Many ghosts can be rendered using very few samples, while others need many more for a good quality. By choosing a different quality level for each ghost, Hullin et al. [5] achieved significant improvement.

Culling the Least Visible Ghosts is another technique for improving performance without sacrificing quality too much.

Bibliography

[1] G. E. Andrews and R. Askey. Classical orthogonal polynomials. In C. Brezinski, A. Draux, A. P. Magnus, P. Maroni, and A. Ronveaux, editors, *Polynômes Orthogonaux et Applications*, pages 36–62, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[2] R. Barnard, G. Dahlquist, K. Pearce, L. Reichel, and K. Richards. Gram polynomials and the kummer function. *J. Approx. Theory*, 94(1):128–143, jul 1998.

[3] C. Fitzgerald. Wgpu. `https://github.com/gfx-rs/wgpu/commits/master/etc/big-picture.png`, 2021 (accessed Januarary 2022).

[4] V. X. Genest and L. Vinet. The multivariate hahn polynomials and the singular oscillator. *Journal of Physics A: Mathematical and Theoretical*, 47(45):455201, oct 2014.

[5] M. B. Hullin, E. Eisemann, H.-P. Seidel, and S. Lee. Physically-based real-time lens flare rendering. *ACM Trans. Graph. (Proc. SIGGRAPH 2011)*, 30(4):108:1–108:9, 2011.

[6] M. B. Hullin, J. Hanika, and W. Heidrich. Polynomial Optics: A construction kit for efficient ray-tracing of lens systems. *Computer Graphics Forum (Proceedings of EGSR 2012)*, 31(4), July 2012.

[7] Intel. Intel math kernel library. `https://software.intel.com/en-us/mkl`, 2003 (accessed Februaruary 2022).

[8] H. Joo, S. Kwon, S. Lee, E. Eisemann, and S. Lee. Efficient ray tracing through aspheric lenses and imperfect bokeh synthesis. *Comput. Graph. Forum*, 35(4):99–105, jul 2016.

[9] LaCroix Precision Optics. Dynamic Material Selection. `https://www.lacroixoptical.com/sites/default/files/content/LaCroix%20Dynamic%20Material%20Selection%20Data%20Tool%20vJanuary%202015.xlsm`, 2015 (accessed October 2015).

[10] T. L. Lai, H. Robbins, and C. Z. Wei. Strong consistency of least squares estimates in multiple regression. *Proceedings of the National Academy of Sciences*, 75(7):3034–3036, 1978.

[11] C. Mauer. Measurement of the spectral response of digital cameras with a set of interference filters. Master's thesis, University of Applied Sciences Cologne, 2009.

[12] N. Orrù. Ofx-rs. `https://github.com/itadinanta/ofx-rs`, 2018 (accessed Februaruary 2022).

[13] E. Pekkarinen and M. Balzer. Physically based lens flare rendering in "the lego movie 2". In *Proceedings of the 2019 Digital Production Symposium*, pages 1–3, 07 2019.

[14] Pixar. The imperfect lens: Creating the look of Wall-E. Wall-E Three-DVD Box., 2008.

[15] O. Rodrigues. De l'attraction des sphéroïdes. *University of Paris*, pages 361–385, 1816.

[16] Rust Graphics Mages. Wgpu. `https://github.com/gfx-rs/wgpu`, 2018 (accessed Januarary 2022).

[17] E. Schrade, J. Hanika, and C. Dachsbacher. Sparse high-degree polynomials for wide-angle lenses. *Computer Graphics Forum*, 35(4):89–97, 2016.

[18] W. Sellmeier. Über die durch die Aetherschwingungen erregten Mitschwingungen der Körpertheilchen und deren Rückwirkung auf die ersteren, besonders zur Erklärung der Dispersion und ihrer Anomalien, Jan. 1872.

[19] The Open Effects Association. Open effects. `http://openeffects.org/`, 2006 (accessed Februaruary 2022).

[20] The Rust Foundation. Rust Programming Language. `https://www.rust-lang.org/`, 2010 (accessed Januarary 2022).