Optimizations for Rendering Realistic Lens Flares in Polynomial Optics

By

Stephen DiIorio

* * * * * * * * *

Submitted in partial fulfillment

of the requirements for

Honors in the Department of Computer Science

UNION COLLEGE

June, 2015

Abstract

DIIORIO, STEPHEN Optimizations for rendering realistic lens flares in polynomial optics. Department of Computer Science, June 2015.

ADVISOR: Anderson, Matthew

Lens flare is a common optical phenomenon exhibited by lens systems, like those used for professional photography or film. As light travels from the front of a lens system towards the sensor at the back, it can either refract through or reflect off of the surfaces of the lenses. A lens flare is the result of (typically) unwanted reflections and scattering caused by imperfections in the lenses. This reflected or scattered light then travels to the sensor following unexpected paths. While considered by some to be degrading artifacts, lens flares have grown to become an essential ingredient for realistic imagery and are often used as artistic effects. As such, adding lens flares to realistic image rendering has become a desired feature. However, while it is possible to render realistic lens flares using ray tracing, current techniques are often too slow to use in real-time. On the other hand, approximations can be made to vastly improve the frame rate of rendering, but this comes at the cost of visual fidelity. We present several methods of approximation in an effort to render a more realistic lens flare in real-time.

Contents

1 Introduction **1**

1.1 Rendering and Ray Tracing 1

1.2 Monte Carlo Methods 5

1.3 Lens Flares 7

1.4 Polynomial Optics 10

1.5 Non-Linear Least Squares Adjustments 13

2 Related Work **14**

2.1 Rendering Techniques and Monte Carlo Methods 14

2.2 Lens-System Approximations 15

2.3 Lens Flares 16

3 Methods **17**

3.1 Algorithms for Producing Lens Flares 17

3.2 Algorithms for Evaluating and Manipulating Polynomials 18

3.2.1 Methods for Non-Linear Least Squares Adjustments 20

4 Experiments **22**

4.1 Specifics for Removing Terms 23

4.2 Specifics for Memoizing 24

4.3 Specifics for Non-Linear Least Squares Adjustments 24

5 Results **25**

5.1 Improving Performance: Removing Terms and Memoizing 25

5.1.1 Results from Removing Terms 25

5.1.2 Results from Memoizing 26

5.2 Improving Fidelity: Non-Linear Least Squares Adjustments 28

5.3 Final Images 28

5.3.1 Image Evaluation 29

6 Conclusion **34**

7 Future Work **35**

8 Acknowledgments **36**

Appendices **40**

A Rendered Lens Flare Images **40**

B Lens flare Comparisons **47**

C Difference Images of Rendered Lens Flares **51**

D C++ Code For Generating Lens Flares **56**

E C++ Code for Manipulating Polynomials **71**

List of Figures

1 Ray Tracing to Produce Images 8
2 A Real Lens Flare 9
3 Monte Carlo Sampling and Color Sampling 10
4 Interfaces in Polynomial Optics 11
5 Mapping Function 12
6 Polynomial Equations 19
7 The Scene 22
8 Achromatic Lens 23
9 Removing Terms and an Attempt at Memoizing Data 26
10 Removing Terms and a Better Attempt at Memoizing Data 27
11 Rendered Lens Flares 29
12 Difference and Scaled Difference Images 31
13 Different Image Evaluation Techniques 33
14 Largest Pixel Difference versus Polynomial Evaluation Time 34
15 Complete Set of Rendered Lens Flares 40
16 Comparisons Between the True Image and Other, Rendered Images 47
17 Complete Set of Difference and Scaled Difference Images 51

1 Introduction

In computer graphics we are interested in rendering a particular scene to a screen, whether it be done in real-time, like in a video game, or done over some longer period of time, like post-production effects in movies and photography. To render a scene realistically, we generally model how light would travel from a light source, through and off objects in the scene, and through a series of lenses, like those found within a professional camera, before reaching an optical sensor. A physical side-effect of using lenses is lens flare. While some consider it a degrading artifact, lens flare is considered by others to add to the perceived realism of an image. As such, inventing a method that accurately replicates a camera, its series of lenses, and all of its tiny flaws has become a group effort [1, 2, 3, 4, 5].

The idea of rendering lens flares is not novel; its foundation relies on techniques using Monte Carlo approximations and ranges to different ways of simulating lens physics, all of which are still undergoing constant improvements. In the following sections we first describe the process of rendering an image via ray tracing and Monte Carlo approximations. We then discuss what lens flares are and why using Monte Carlo methods to render them is a difficult problem. This leads into a discussion of other methods we can use to produce lens flares and other optimizations we can make.

1.1 Rendering and Ray Tracing

Rendering is the process of generating an image from a scene. Rendering is usually accomplished with either of two algorithms: object-order rendering or image-order rendering [6]. In object-order rendering, we consider every object in the scene, determine how they affect the color of the pixels in our image, and update the appropriate pixels. In image-order rendering, more commonly referred to as *ray tracing*, we consider each pixel individually, determine which objects affect that pixel and then calculate that pixel's value based on the objects present.

With the object-order algorithm, we iterate through every object, determine its contribution to the pixels in our image that will display it, and adjust those values accordingly. While its implementation is straightforward, this method of rendering is typically wasteful in the sense that it goes through every object in a scene, even if it is obstructed by another and will not be visible in our image. This can become computa-

tionally impractical when dealing with very detailed scenes.

Ray tracing offers several benefits compared to other rendering methods. With this type of rendering, we are concerned with how rays intersect within a scene. A *ray* is defined as a 3D parametric line given by

$$\vec{p}(t) = \vec{e} + t(\vec{s} - \vec{e}) \tag{1.1}$$

which can be interpreted as starting at the position of the eye, $\vec{e}$, and moving a fractional distance t in the direction of $(\vec{s} - \vec{e})$ towards a point, p [6]. *Tracing* a ray is sending one into a scene and observing which objects it intersects with [7]. When rendering a scene, we are interested in understanding how light would behave and move from object to object in the scene. Because of this, rays act as an excellent substitute for modeling the behavior of light.

Because rays exhibit the same behavior as light, capturing the finer details of a scene, like the particular colors and properties of different materials, becomes easier. Since rays are typically sent out from the sensor to sample a scene, we gain an understanding of the positioning of objects. This allows us to accurately know all of the objects present at a particular location and the compound effect they have on that pixel's color. Knowing this order of intersection also allows us to determine if, from our current point of view, the object a ray intersects with obscures the rest of the scene. In this case, we simply stop following that ray and associate the current pixel color with its first intersection, saving computation time [7]. One of the major problems with ray tracing, however, is the large number of samples needed to accurately model the behavior of light and generate a detailed rendering. A single rendering can take several hours to accomplish depending on the desired level of detail and complexity of the scene [8].

We are most interested in the measurement of radiance, or the amount of light, reaching any given point. This becomes particularly useful when we try to determine shadows within a scene. A *shadow* is formed on a surface when any object obscures a direct path from the light source to that surface. For any given point in the scene, we have to consider the direction light is coming in at, $\vec{l}$, the view direction, $\vec{v}$, and the vector normal to the surface, $\vec{n}$. If, for a particular point, a ray traveling in the same direction as $\vec{l}$ intersects with an opaque surface, then we know that light from the source does not reach that point, and it is in the shadow of an object [6]. This approach does not consider stray or secondary light that has reflected off some surface

that might illuminate that particular point, however.

Reflection also becomes easy to evaluate. *Perfect reflection* is when light hits a surface at some angle and bounces off at the same angle with respect to the normal. As such, a person staring at the surface from above with some angle with respect to the normal would see the same image that a person below the surface would see by staring off at the same angle. With rays, this means that an incident ray, $\vec{d}$, hits a surface at some angle, and a reflectance ray, $\vec{r}$, bounces off at the same angle. A mirror reflection can be calculated using

$$\vec{r} = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n}. \tag{1.2}$$

Thus, a particular color for a reflective surface can be determined by following the reflecting ray to the next object in the scene [6]. Not all surfaces reflect light perfectly and sometimes energy in the light is lost after it reflects off a surface, so different checks can be added to shift the color of the ray to the appropriate value.

Another phenomena to consider is refraction. *Refraction* occurs when light bends as it travels through dielectric surfaces with different indices of refraction. As light travels from a medium with a refractive index of n_1 to one with a refractive index of n_2, we find that the magnitude of the bend can be expressed using Snell's law: $n_1 \sin\theta = n_2 \sin\phi$, where θ is the angle of incidence and ϕ is the angle of refraction [6]. In three dimensions, we can express the transformed ray, $\vec{t}$ that has undergone refraction as

$$\vec{t} = \frac{n_1(\vec{d} - \vec{n}(\vec{d} \cdot \vec{n}))}{n_2} - \vec{n}\sqrt{1 - \frac{n_1^2(1 - (\vec{d} \cdot \vec{n})^2)}{n_2^2}}. \tag{1.3}$$

Here, when the value within the square root is negative, total internal reflection occurs (meaning the rays cannot pass through the surface no matter the angle of incidence) [6].

With realistic rendering, we try to model how light behaves in an environment, and it is not the case that light either reflects off a surface or refracts through a surface. Whenever light moves from one medium to another some light reflects and some light refracts. The reflectivity and transmissivity, essentially the percentage of light that reflects and the percentage of light that refracts respectively, are expressed by Fresnel's equations, and the directions of the rays determined by the law of reflection and Snell's law [2].

We also need a way to not only consider the transport of light but also the physical properties of the

surfaces. Not all surfaces transport light in the same manner, giving each of them their particular and individual appearances. Thus, for a more complete definition of how light behaves, we consider the ratio between incoming and outgoing radiance from a surface. This ratio, parameterized by both the angle of incidence and the angle of emittance, is also known as the bidirectional reflectance distribution function (BRDF). A BRDF allows for us to express the reflectance of a surface, ρ, as a function of incoming and outgoing radiance, L_i and L_o respectively, the angle of incoming light with respect to the normal, θ_i, and the range of angles incoming light can hit the surface with, $\Delta\sigma_i$ [6]. Formally, a BRDF can be expressed as

$$\rho = \frac{L_o}{L_i \cos\theta_i \Delta\sigma_i}. \tag{1.4}$$

A BRDF allows us to describe how materials behave within an environment, and BRDFs can either be derived from physics describing the environment or measured from the actual object and stored in a table [7].

All rendering methods try to accurately model the behavior of light as it scatters off the surfaces of different objects [9]. In computer graphics, accurately modeling the equilibrium of light incident and radiating from a particular location ultimately leads to a more realistic rendering, and we can describe a mathematical means to compactly express this relationship.

We define S to be the union of all the surfaces of all the objects in a scene and x, x', and x'' to be points that range over all of these surfaces. If we were to take the expression for a BRDF and rearrange it so that it accounts for the summation of incoming light rays and write it in terms of surface radiances, we would end up with what is known as the "rendering equation"

$$I(x, x') = g(x, x')\left[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''\right]. \tag{1.5}$$

Here, $I(x, x')$ relates the intensity of light passing from point x' to point x, $g(x, x')$ relates to geometry of the system, $\epsilon(x, x')$ relates the intensity of light emitted from x' to point x, and $\rho(x, x', x'')$ relates the intensity of reflected light from x'' to point x after reflecting off a surface located at x', and all these terms carry with them information about the particular wavelength and polarization of light. At its essence,

this equation simply equates the energy flow from one surface to another [9]. This equation can easily be expanded to include other physical effects we might observe such as the interaction of light of different phases or the refraction of light through inhomogeneous materials. Clearly, this integral equation quickly grows in complexity the more realistic the rendering becomes. At the core of ray tracing, we are trying to accurately evaluate the rendering equation in order to realistically map how light is represented within a scene, and this requires a solid understanding of the physical properties of light and how it interacts with the environment.

Exploring light transport theory is not specific to computer graphics but expands to a large field of different physical topics because of its similarity with radiation and particle transport problems. Aside from the the obvious, exploring how light behaves and affects what we see, light transport theory is used in studying radiative heat transfer, neutron transport and the design of nuclear devices, and radar and acoustic problems [10].

1.2 Monte Carlo Methods

A *Monte Carlo method* is a process that uses random sampling to estimate solutions to a quantitative problem. It often offers a way to quickly compute complex integrals, like the one found in the rendering equation [11]. Physicists first used Monte Carlo methods in the 1950s when designing thermonuclear weapons [12].

A *continuous random variable* X can randomly take on the value of x, which is described by a *probability density function* p, which must always produce a positive number and be normalized. The probability that the value of x will be within some interval a to b is determined by

$$Pr\{a \leq X \leq b\} = \int_a^b p(x)dx. \tag{1.6}$$

The *expected value*, E, is the average value of a random variable, $Y = f(X)$, and it can be expressed as

$$E[Y] = \int_\Omega f(x)p(x)dx, \tag{1.7}$$

where Ω is the domain of integration. The *variance*, V, is a measure of how much the values for a random

variable differs from the expected value for that variable, and it can be expressed as

$$V[Y] = E[(Y - E[Y])^2]. \tag{1.8}$$

We define a function, F, to be an *estimator* if its expected value is a close approximation to some unknown quantity, the *estimand* [11].

Consider evaluating the integral

$$I = \int_{\Omega} f(x)dx \tag{1.9}$$

where $f : \Omega \to \Re$ is a real-valued function. We can approximately evaluate this integral by sampling N points, $X_1, X_2, \ldots, X_N$. If we chose our samples uniformly and randomly, we compute an approximation, F_N for the value of the integral

$$I \approx F_N = \frac{1}{N} \sum_{i=1}^{N} f(X_i). \tag{1.10}$$

With this approximation, the error is proportional to $1/\sqrt{N}$ [11]. This means that we must increase the number of samples we take by four times in order to decrease the error by a half.

However, it is rarely the case where our points are truly picked uniformly and randomly. In this case, we can rewrite Equation (1.9)) as

$$I = \int_{\Omega} \frac{f(x)}{p(x)} p(x)dx. \tag{1.11}$$

After applying our Monte Carlo approximation, we can write

$$I \approx F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}, \tag{1.12}$$

which is just our general case estimator [11].

This technique, known as Monte Carlo integration, offers several advantages. Firstly, the error converges at a rate proportional to $N^{-1/2}$, which is independent of the dimension of the integral. Second, it is relatively simple to compute as it only samples and evaluates the function f at points. In other words, it runs in time $O(N \cdot t)$, where t is the time to evaluate f.

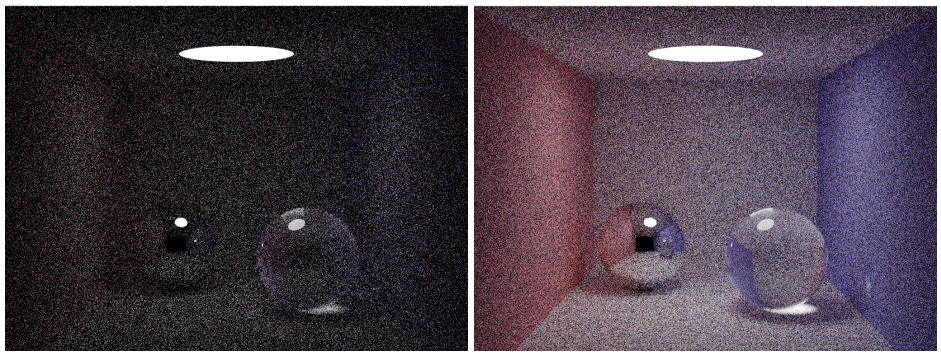
Many graphics problems involve the calculation of some integral over all space and that is difficult in the sense that the integrand may not be continuous over its entire domain, occupy a space of high dimension, or be singular, i.e., it reaches infinity for some points of its domain of integration [10]. Monte Carlo integration avoids these stumbling blocks and, as such, is often useful in graphics applications, like when modeling motion blur or depth of field [10]. With ray tracing, we can approximate the color of any particular pixel by randomly sampling points within that pixel [13]. The more points we sample, the more accurate the rendering becomes, but at a cost of increased computation time. Figure 1 shows several images rendered using ray tracing and Monte Carlo methods.

1.3 Lens Flares

Lens flares are a consequence of the physical limitations of lens systems. Light, when traveling through a series of lenses, can reflect off the glass or off particles on the lens and bounce through unexpected paths instead of just passing through the lenses. This results in light reaching the sensor at undesirable locations forming glares and other geometric patterns.

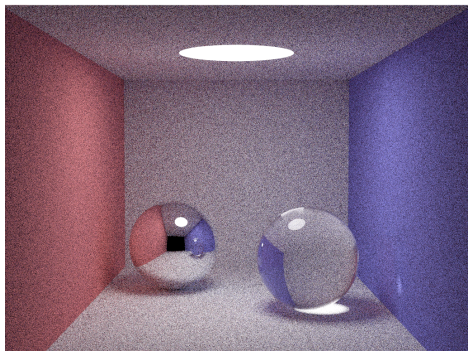
Lens flares add to the perceived realism of a synthetic environment, often suggesting to the observer the presence of bright light sources [14] and establishing a particular mood for the environment [2]. Because of this, a great deal of effort has gone into accurately generating lens flares and other camera defects in works such as movies and video games [3]. To do this, designers generally try to model their camera lens to great precision, detailing all of their flaws and limitations, in order to achieve a realistic image [2].

One way of modeling lens flares is to ray trace through an optical system. The result is a very accurate and detailed image, but this process is inefficient and takes on the order of several hours to complete just a single rendering. With ray tracing, we have to sample multiple rays of light at different points and of different colors to produce a realistic image. In the case of producing a lens flare, we have to simulate complicated paths through a lens system and how light behaves at every step, as shown in Figure 3. Additionally, these systems only consider effects generated from the geometry of the optics, while several other features of lens flares arise only when we consider wave-optical effects, i.e., effects that arise when we consider the wave nature of light (e.g., chromatic aberrations) [2].

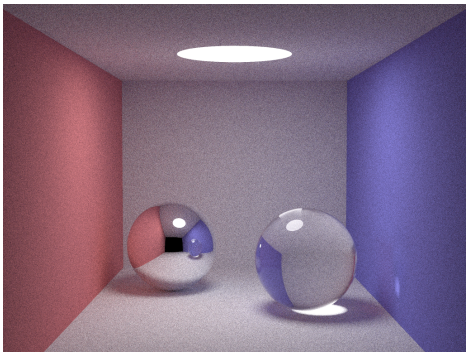


(a) 4 spp

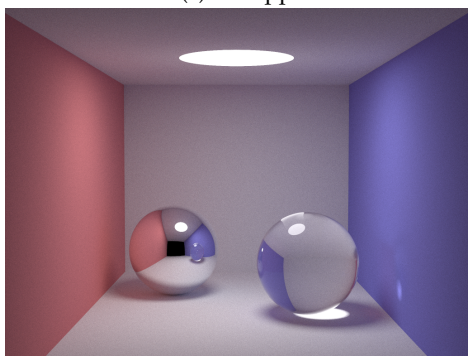
(b) 24 spp



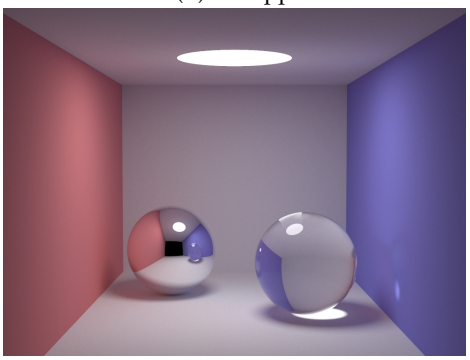
(c) 100 spp



(d) 500 spp



(e) 5000 spp



(f) 25000 spp

Figure 1: These are separate renderings of the same scene using ray tracing, each with a different number of samples per pixel (spp). When the number of samples is low, the image is not very clear and contains noticeable noise in the form of black spots. We get a decent rendering when we use 500 spp, but the image still has imperfections in the form these black spots. Increasing the sampling to 5000 spp drastically improves the quality and 25000 spp does so even more, but the cost to conduct such a large sampling outweighs the benefits since the differences between the two are hardly noticeable. (Images rendering using source code provided by [8].)



Figure 2: This is an example of a real lens flare formed from a lens system. Around the sun, we can observe streaks of light spanning in several directions. In addition to this, the geometric shapes that span from the light source to the bottom left corner of the image form what we typically consider to be a lens flare. The photo itself was taken in Saratoga, NY. Image courtesy of [15].

Another way of producing lens flares is used in real-time applications like games. With real-time rendering, pre-made starbursts, circles, and rings, or texture sprites, are placed within the world. It is then up to the artist to manage these effects by hand and handle how they behave in the environment as people interact with it so the lens flare looks believable. While this approach may convince some, the work needed to implement such a feature by hand is exorbitant and often fails to include intricate dynamics from a real lens flare [16, 2].

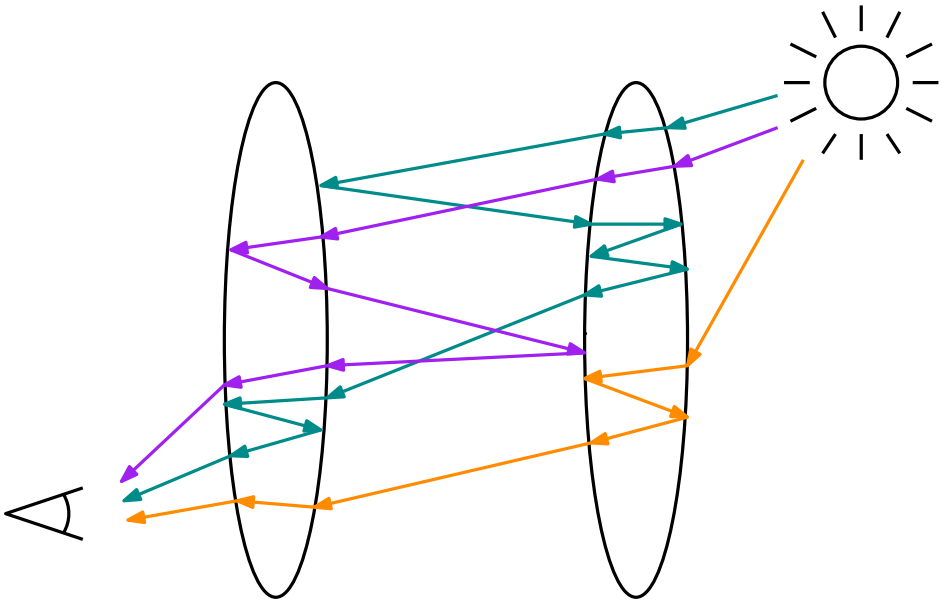


Figure 3: Lens flares are typically difficult to compute. As shown here, one way to compute them on a computer is via Monte Carlo methods, where we randomly sample multiple points of the scene we are trying to render to form an image. For each of these points, we have to worry about how they interact with all of the lenses in the system. In addition to this, to form a color image, we have to randomly sample points repeatedly focusing on how a particular wavelength of light would interact with the system.

1.4 Polynomial Optics

An alternative method of producing lens flares is via polynomial optics. In polynomial optics, we define the basic properties of the lenses used in lens systems, like the radius of the lenses, what type of glass they are made of, the spacing of the lenses, etc. Using these different attributes, we construct a lens system we want to simulate and define a path consisting of reflections off and refractions through the different optical elements of the lens system. We can then approximate these paths and lens systems using Taylor expansions to produce a set of polynomial maps known as *mapping functions*. A polynomial map is a function $f : \Re^m \rightarrow \Re^n$ such that $f(x_1, x_2, \ldots, x_m) \rightarrow f_1(x_1, x_2, \ldots, x_m), f_2(x_1, x_2, \ldots, x_m), \ldots, f_n(x_1, x_2, \ldots, x_m)$, where f_i are polynomials.

By taking a Taylor expansion of the ray parameters, we can generate polynomial maps that approximate how light reflects or refracts from one interface in the lens system to another. We define an *interface* to be

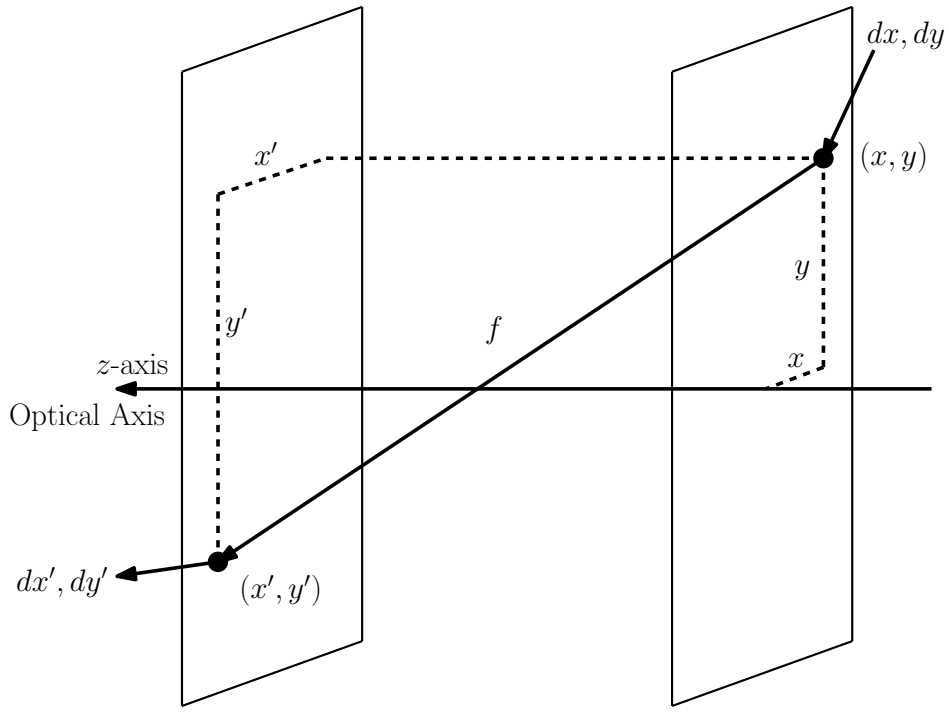


Figure 4: Polynomials maps describe how light moves from one interface to another. In this case, they take in some initial x and y position and an initial angle in the form of dx and dy to produce a function, f. This function transports us from one location on an interface to a new position, some x' and y' coordinates on a subsequent interface. It also defines for us the new angle, new dx' and dy' values, describing how the angle at which light leaves this new interface.

one side of a lens. These can be thought of as 2D planes whose centers are on a common optical axis (see Figure 4). The spacing of the interfaces on either side of the same lens is defined by the thickness of the lens, and the spacing of interfaces on two different lenses is defined by the distance between those lenses in the physical system.

The polynomial maps are parameterized by five different variables: an x and y position on an initial interface, the angle light leaves the initial interface, dx and dy, and a color, λ. In actuality, these polynomials describe how light travels from one interface to another by mapping the initial position of light, given by its x and y position on the interface, to a new position on a subsequent interface, an x' and y' position. The angle light leaves an interface with can be calculated from these values and the distance between the interfaces. Figure 4 shows how these polynomial maps transport us from one interface to another.

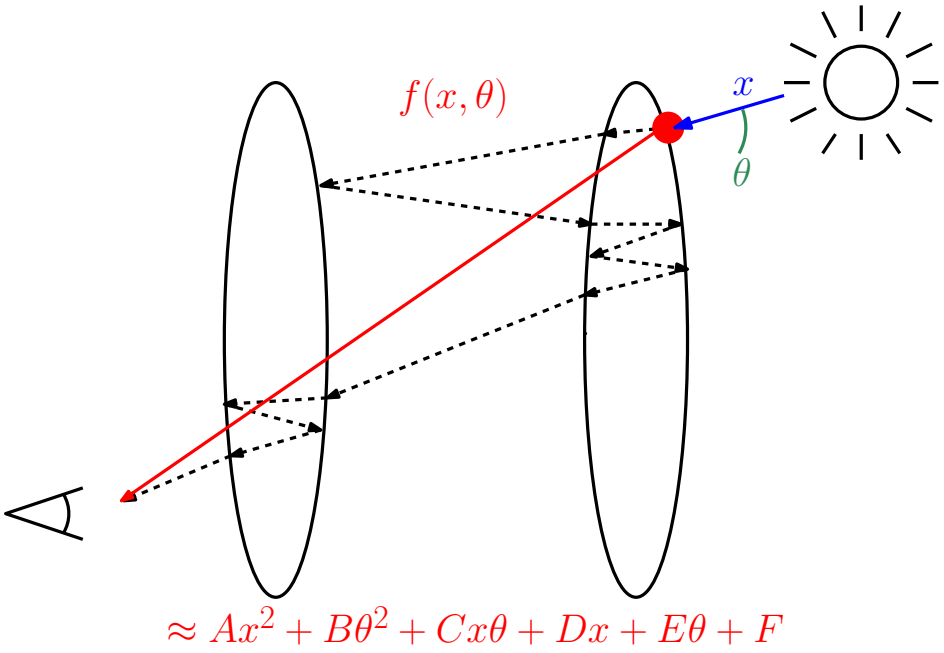


Figure 5: Instead of ray tracing through a lens system, shown with the dotted arrows, we can approximate an entire system using a mapping function. The mapping function is designed to compute paths through a lens system, and, given some input position, x, and some input angle, θ, tell us where light would end up as it exits the lens system as if we had ray traced through the entire system. This is shown via the solid red line, which goes from an entry point directly to a point on the sensor. The mapping function is a polynomial of a specific degree formed from the input parameters.

A mapping function for the entire system is formed by composing several of these polynomials maps describing how light moves from interface to interface. We define some path through a lens system and sequentially compose polynomial map after polynomial map until we form a path that leads from the scene we are trying to render to the sensor of the camera. The beauty of this approach is that, by the end, we have a set of polynomial map that defines how light enters the lens system and exits the system without us having to perform expensive operations to determine how light behaves at every interface, as is the case with ray tracing. Figure 5 gives more of an idea of how a mapping function works.

1.5 Non-Linear Least Squares Adjustments

The level of precision relative to a physical system of the mapping functions we generate can vary depending on the degree of the polynomials. We can alter the degree with a trade-off: increasing the degree of a polynomial increases its accuracy but also makes it more difficult to evaluate because it contains more terms, while decreasing the degree has the opposite effect.

An additional change that can be made to these polynomial equations that would allow us to increase the accuracy of the polynomials without increasing their degree is by adjusting the coefficients of the terms in a polynomial based on information gathered from some ground truth. This ground truth is, ideally, a detailed ray tracing through the lens system we are trying to approximate. By adjusting the coefficients using a non-linear least squares method [17], we produce polynomials that more accurately model the physical system without drastically increasing their complexity.

The idea behind a non-linear least squares problem is to find a minimum of a function. This function, F, is defined as the sum of the squares of non-linear, real-valued functions, f. Let $f : \Re^m \to \Re^n$ be a function that maps a parameter vector $\mathbf{p} \in \Re^m$ to a measurement vector $\hat{\mathbf{x}} \in \Re^n$ such that $f(\mathbf{p}) = \hat{\mathbf{x}}$. The goal is to then find a vector, $\mathbf{p}^+ \in \Re^m$, that minimizes $\mathbf{e}^\top \mathbf{e}$ given a parameter estimate, $\mathbf{p}_0 \in \Re^m$, and a measured vector, $\mathbf{x} \in \Re^n$, where $\mathbf{e} \in \Re^n = \mathbf{x} - \hat{\mathbf{x}}$. This problem can be approximated when we look at a small region around $\mathbf{p}$ defined by $\delta_\mathbf{p} \in \Re^m$. We can write

$$f(\mathbf{p} - \delta_\mathbf{p}) \approx f(\mathbf{p}) + \mathbf{J}\delta_\mathbf{p}, \tag{1.13}$$

where $\mathbf{J} \in \Re^{n \times m}$ is the Jacobian of f, defined as a matrix whose elements are

$$\mathbf{J} = \frac{\partial f(\mathbf{p})}{\partial \mathbf{p}}. \tag{1.14}$$

This process of finding a minimum starts with some $\mathbf{p}_0$ and iterates through subsequent vectors until $\mathbf{p}^+$ is found. This involves solving for $\delta_\mathbf{p}$, the solution to a linear system of equations

$$\mathbf{J}^\top \mathbf{J} \delta_\mathbf{p} = \mathbf{J}^\top \mathbf{e}. \tag{1.15}$$

In the context of polynomial optics, we use the Levenberg-Marquardt algorithm to improve the accuracy of the polynomial maps we construct. For simplicity's sake, we consider the coefficients of all possible terms for a given polynomial map with a given degree. We can calculate the total number of terms that can appear in a polynomial map, $N \in \Re$, given the degree, $d \in \Re$, of the polynomial map and the number of input variables in the polynomial map, $n \in \Re$, with the combination

$$N = \binom{d+n}{n}. \tag{1.16}$$

We also supply a set of data points we generate, $D \in \Re^n$ that we use to evaluate these polynomials. We also consider a series of ray-traced points, $T \in \Re^n$, to be the "true" values that act as our measurement vector. The minimization problem then tries to adjust the coefficients of the polynomial maps to minimize the difference between the values in the measurement vector and the values the polynomial maps produce when we evaluate them using the parameter vector.

In the rest of this paper, we discuss related efforts in producing lens flare and our approaches to produce polynomial maps that are easier to compute and produce images of high visual fidelity. After we discuss our approaches, we will go into how successful these approaches were and how we evaluated the images we produced.

2 Related Work

Shirley et al. [6] and Gortler [7] provide excellent references regarding the origins and mathematics involved in ray tracing and Monte Carlo approximations. The rendering equation, Equation (1.5), and its physical origins and intricacies is described by Kajiya [9]. He also describes several methods of reducing this equation and proper sampling techniques.

2.1 Rendering Techniques and Monte Carlo Methods

Veach, in his Ph.D. thesis [10], describes a more detailed mathematical foundation for ray tracing and explores several different rendering methods and their effectiveness.

First, Veach and Guibas offer a sampling technique called multiple importance sampling, further described in [18], which combines several samplings (in this case, the combination of a sampling of the BRDF and a sampling of the light source) of a scene to reduce the noise of a rendered scene. Their method suggests that it is possible, with little extra work, to combine several good methods of rendering a scene and preserve the benefits that each provide. But this method requires some additional computation and is only appropriate if we cannot find a single way to sample a particular scene.

A second approach is bidirectional path tracing. This method generates a path by combining two independent samples, one beginning at the optical sensor and the other beginning at the light source. A problem with this approach is that there is no obvious and efficient way to sample the light sources, it cannot handle difficult geometry within the scene, and it can miss the contributions of some paths.

Finally, Veach overviews the metropolis light transport (MLT) algorithm. This algorithm works by sampling light paths from the light source to the lens. Each new path taken is a random mutation of the previous, and these permutations are accepted or rejected based on their contribution to the final image (a ray that hits a wall is less likely to be accepted compared to one that does not). Each path updates the pixels to form a rendered image. This method leads to lower noise in more complex images, but it is not obvious beforehand what the most efficient set of parameters that define the mutation is.

Another way of sampling an image, proposed by Kollig and Keller [19], is through a combination of jittered sampling and Latin hypercube sampling. This method removes some of the randomness with sampling, and divides the scene into different portions such that each portion is equally sampled, which ultimately improves the approximation and leads to a faster convergence.

2.2 Lens-System Approximations

These sampling methods are a means to lessen the computational burden of modeling how light moves through a scene. Kolb et al. [4] suggests that along with worrying about how to correctly model the transport of light throughout a scene, we need to also worry about modeling the components of the camera used for the realistic rendering. Focusing on this aspect of the rendering helps to enhance the effects that add to the realism of the scene, like a motion blur, depth of field, or even lens flares.

Hullin et al. [5] simplified the problem of ray tracing through an optical system by performing a Taylor expansion on the ray parameters, essentially giving a low-order approximation of an analytical solution and enabling shorter computation times. They generated a construction-kit result that accurately models many complex optical systems. Building off of this approximation, Hanika and Dachsbacher [1] further refined Hullin et al.'s results to develop an efficient method for rendering lens systems that requires much less overhead compared to previous algorithms. Their method gives an approach for rendering polynomial optics that closely matches ground-truth ray tracing and, ultimately, avoids the need for ray tracing through a system of lenses to form an image. What they did not explore, but left as an open question, was whether rendering lens flares can be similarly accelerated using their technique.

2.3 Lens Flares

The goal with trying to realistically model how light interacts with lens systems is so that we can notice the more subtle side-effects that arise from the geometry and physics of the system. Yet, for some time, these effects, like glares and lens flares, have been added in as a static addition to the scene.

Prior to accurate, real-time lens approximations, designers had to grossly reduce the complexity and level of detail in how they produced lens flares. An early way of adding a lens flare to an image using fast texture mapping techniques was done by Kilgard [20]. A variable brightness was added to these rendered lens flares by using occlusion queries to see how many paths reach the sensor for any particular location [21]. Oat [22] developed a steerable filter kernel that added light streaks to the scene, indicating relative brightness of light sources in relation to the background. Alspach [23] describes a method for generating vector lens flares that are based on user input and whose colors and display are affected and determined by the rest of the scene. All of these methods offer ways of displaying lens flares in a rendering, but none actually consider the physics of the optical systems that generate the phenomena.

Path tracing is another method for rendering a scene, and Keshmirian [24] used it to approximate a lens system and simulate a lens flare. This was a first step towards physical realism, and his approach takes less time to compute compared to ray tracing but lacks the finesse to generate high-quality images that cover all the aberrations generated within a lens system.

Hullin et al. [2] aimed to rectify the inaccuracies of previous methods by basing their approximation more on the physics of a lens system. They used the work done by Oh et al. [25] to preprocess light effects for different apertures using Fourier transforms. Their approach uses ray tracing, but is flexible enough to be efficient during real-time rendering by removing artifacts like ghosts that added very little to the final image. While they made advances in efficiency, their end results varied significantly from what is actually observed with different lens systems. Lee and Eisemann [16] simplified this approximation even further, doing away with a ray tracing approach, and used matrix approximations based on a minimal form of polynomial approximations to greatly improve the real-time rendering capabilities of this system. They found a significant increase in the number of frames per second they were able to produce but at the cost of many details relating to optical effects generated from a lens system.

3 Methods

The polynomial maps are generated using the Polynomial Optics library developed by Hullin et al. [5]. Also, to adjust the polynomial maps, we use the levmar library [17]. Below, we discuss all of the non-obvious nuances that go into producing images of lens flares and interfacing with each of these libraries.

3.1 Algorithms for Producing Lens Flares

There are two problems that we solved in order to use the Polynomial Optics library to generate lens flares. The first is determining the places where reflections occur, and the second is generating a set of mapping functions that follow this path through the lens system.

The first problem of generating a set of interfaces we reflect at can be solved using a dynamic programming algorithm. To do this, we generate a list of numbers which indicate, in order, the the interfaces we reflect at.

Let Γ be the function responsible for generating all of the possible paths through a system given the number of interfaces in the system, n, the desired number of reflections the path should take, r, and the interface we want to start generating the paths from, i. We require that r be an even number, otherwise we

would have paths that leave the lens system and travel towards the scene and never reach the sensor. These paths are completely useless for our purpose of generating lens flares. We can then derive the conditions for this algorithm as follows

$$\Gamma(n, r, i) = \begin{cases} << >> & \text{if } r = 0 \\ \bigcup\limits_{n > r_1 > i} \left(\bigcup\limits_{0 \leq r_2 \leq r_1} (<< r_1, r_2 >> \times \Gamma(n, r-2, r_2)) \right) & \text{otherwise} \end{cases} \tag{3.1}$$

where $\times$ means, given two sets of sequences of interfaces, S_1 and S_2, $S_1 \times S_2 = \{s_1 \text{ append } s_2, \forall s_1 \in S_1, s_2 \in S_2\}$. The result of completing this is a set of all the possible paths through a lens system with i interfaces, following r reflections.

With a path through the lens system, we can generate the mapping functions by traveling to each interface in order as they appear in the path, refracting and reflecting appropriately.

3.2 Algorithms for Evaluating and Manipulating Polynomials

Once we produce the mapping functions, there is still the issue of efficiently evaluating them. While mapping functions offer a much quicker way of simulating a lens flare compared to ray tracing, they still come with some issues. Many of these polynomial maps contain a lot of exponentiations and multiplications that we need to evaluate (see Figure 6 for an example of a degree-five polynomial). In particular, exponentiation is an expensive computation to complete on a processor, so evaluating these polynomial maps quickly becomes expensive. This especially becomes troublesome when we want to consider evaluating these polynomial maps a multitude of times to render an image in real-time.

The question then becomes, how do we best evaluate these polynomials? This question really has two parts to it: how do we both quickly evaluate these polynomials, and how do we produce polynomials that retain enough information such that they produce images of high fidelity? We propose two methods: removing terms from the polynomial based on some threshold we set, and memoizing computations to more quickly evaluate the polynomials. We also propose using the Levenberg-Marquardt non-linear least squares adjustment algorithm to retain more information within the polynomials.

```
x0'= 0.069134191*x1 + 0.00036023074*x1*x2 - 0.0001688411*x1*x2^2 + 3.2458257*E -
   7*x1*x2^3 - 9.2272451*E - 8*x1*x2^4 - 0.00016840819*x1^3 + 3.2458172*E -
   7*x1^3*x2 - 1.8454372*E - 7*x1^3*x2^2 - 9.2268486*E - 8*x1^5 - 7.3207025*E -
   6*x0 + 4.1412478*E - 9*x0*x2 - 1.0656848*E - 9*x0*x2^2 + 2.6268905*E -
   12*x0*x2^3 - 5.6287706*E - 13*x0*x2^4 - 3.1231979*E - 9*x0*x1^2 + 7.5700566*E
   - 12*x0*x1^2*x2 - 2.9830253*E - 12*x0*x1^2*x2^2 - 2.4200917*E - 12*x0*x1^4 -
   3.8649941*E - 14*x0^2*x1 + 6.0552599*E - 17*x0^2*x1*x2 - 2.5149545*E -
   17*x0^2*x1*x2^2 - 3.9291562*E - 17*x0^2*x1^3 + 2.2452928*E - 20*x0^3 +
   9.3615152*E - 23*x0^3*x2 - 3.570534*E - 23*x0^3*x2^2 - 2.6169659*E -
   22*x0^3*x1^2 - 4.3705215*E - 28*x0^4*x1 - 6.140602*E - 34*x0^5
x1'= 1.2794179 + 0.068410441*x2 + 0.00054647529*x2^2 - 0.00016929924*x2^3 +
   4.2295244*E - 7*x2^4 - 9.2268486*E - 8*x2^5 + 0.00018560429*x1^2 -
   0.0001688676*x1^2*x2 + 5.2132975*E - 7*x1^2*x2^2 - 1.8454072*E - 7*x1^2*x2^3 +
   9.8373164*E - 8*x1^4 - 9.2272437*E - 8*x1^4*x2 + 4.1412287*E - 9*x0*x1 -
   2.0685693*E - 9*x0*x1*x2 + 7.5700566*E - 12*x0*x1*x2^2 - 1.8572524*E -
   12*x0*x1*x2^3 + 2.6268844*E - 12*x0*x1^3 - 1.8572429*E - 12*x0*x1^3*x2 -
   3.9239987*E - 15*x0^2 - 1.4970528*E - 14*x0^2*x2 + 2.4679362*E - 17*x0^2*x2^2 -
   1.0118632*E - 17*x0^2*x2^3 + 2.7296609*E - 17*x0^2*x1^2 - 2.4260539*E -
   17*x0^2*x1^2*x2 + 9.3615506*E - 23*x0^3*x1 - 1.0550567*E - 22*x0^3*x1*x2 +
   1.0732069*E - 28*x0^4 + 9.8601372*E - 29*x0^4*x2
x2'= 0.0017209146 - 0.0022226868*x2 + 0.00071769056*x2^2 + 0.00071769056*x1^2 +
   1.2717995*E - 8*x0*x1 + 5.6343012*E - 14*x0^2
```

Figure 6: Above is a system of degree five polynomials generated from the Polynomial Optics library. Here, $x0$ and refers to the x position, $x1$ refers to the dx position, and $x2$ refers to the dy position. The prime variables refer to the new values of their unprimed counterparts. At this point, the y position of current points we sample have been baked into the other three equations.

The motivation for removing terms stems from the following observation: the coefficients of some terms in Figure 6 are multiplied by, in the most extreme case, 10^{-34}. This is an incredibly small number–there are 33 zeros after the decimal place before we get to the coefficient. Thus, if we could remove the terms that contributed very little to the end result, we could save on the amount of work necessary to evaluate a polynomial. Another observation that led to idea of memoization is the fact that several terms of the same degree appear multiple times within the same system of polynomials; $(x2)^2$ appears multiple times in the equations in Figure 6, for example. So, if we could store this computation once and reuse this result every time a particular term appears, we could waste less time computing the polynomials. Below, we discuss several different algorithms to best deal with this question.

To remove terms, we first compute the maximal values that the input values can take on. Then, these values are used to evaluate every term of the polynomials, and we check to see if the resultant was below a set threshold. If it was below a threshold, that terms was removed from the polynomial.

To memoize data in this system, one can easily compute all useful exponentiations of given input variables. The only detail that is important to note is how we index the array that stores the memoized exponents. We need both an efficient data structure that has quick access times to retrieve the exponentiations we compute and we need a key into particular indices of this data structure that are computationally easy

to evaluate. Arrays proved to be the most efficient to both create, manipulate, and access for our purposes. In this particular library, we are allowed to create polynomials with a maximum degree of 5 and at most 6 input variables. This means, at most, we have 36 different exponentiations to store, and the array had to be at least this big to hold all of the values. Then, to index into this array, a unique key for every exponentiation of every variable is used. This key, k, has to be efficient so that it does not bog down the evaluation of the polynomial, so we compute it as follows

$$k = (v << 3) + v + e, \tag{3.2}$$

where, v is the number of the current input variable we are iterating over and e is the current exponentiation we are computing for this input variable.

3.2.1 Methods for Non-Linear Least Squares Adjustments

We were interested in producing more accurate polynomials in addition to efficiently evaluating them. To do this, we used the Levenberg-Marquardt algorithm implemented in the levmar library. In order to do so, we needed to design several different algorithms for the minimization library to take advantage of.

The first function that is required by the levmar is a function that relates a parameter vector to a measurement vector. In other words, given a set of data points that we generate, we want to produce a set of output values that the mapping function produces when we evaluate the polynomials with the given data. Our implementation of this is shown in Algorithm 1.

Additionally, we need to generate a Jacobian matrix of this polynomial system. The Jacobian is of size $n \times m$. The rows of the Jacobian can be thought of as groups of four. The first row in a grouping contains the partial derivative of the first polynomial in the mapping function with respect to the coefficients of the polynomial. These values are stored in the first m/Number of Variables indices. The rest of the row is then zeros. The second row in this set is similarly the partial derivative of the second polynomial of the mapping function with respect to the coefficients of the polynomial, and these values are stored in the next m/Number of Variables indices of the Jacobian function while the rest of the indices are zeros. This pattern continues for the rest of polynomials and for the rest of the data points we generate. This method is shown

`GenerateFunction`(p, m, n, c, v, $data$)

Data: p, the set of coefficients of the polynomial, m, the size of our parameter vector, n, the size of our measurement vector, c, the number of coefficients in each polynomial, v, the number of variables in the polynomial, $data$, the input points.

Result: The set, x, that contains the measurements produced by evaluating our polynomial maps.

```
x ← ∅
for d ∈ data do
    for i = 0 to n/v do
        for j = 0 to v do
            outputValue ← 0
            for k = 0 to m/c do
                outputValue += p[k] × Exponentiate (d)
            end
            Append outputValue onto x
        end
    end
end
```

Algorithm 1: Here we show a functional relation describing how to evaluate the polynomials of a mapping function. Given the parameters that levmar wants these functions to have, we pass in the coefficients of the terms in the polynomials, the size of the problem at hand, and the data points we want to evaluate the polynomials at, and store the results in an array, x.

`GenerateFunction`(p, m, n, c, v, $data$)

Data: p, the set of coefficients of the polynomial, m, the size of our parameter vector, n, the size of our measurement vector, c, the number of coefficients in each polynomial, v, the number of variables in the polynomial, $data$, the input points.

Result: The set, jac, that contains the Jacobian matrix.

```
jac ← ∅
for d ∈ data do
    curIndex ← 0
    for i = 0 to v do
        for j = 0 to m/v do
            outputValue ← 0
            if p[curIndex ] ≠ 0 then
                outputValue += Exponentiate (d)
            end
            Append outputValue onto jac
            curIndex += 1
        end
    end
end
```

Algorithm 2: Here we show how to form the Jacobian of a mapping function. Each polynomial equation takes up a row of the Jacobian, where the partial derivatives of these polynomials with respect to its coefficients take up the values of the row. The unfilled spots of each row are filled with zeros.

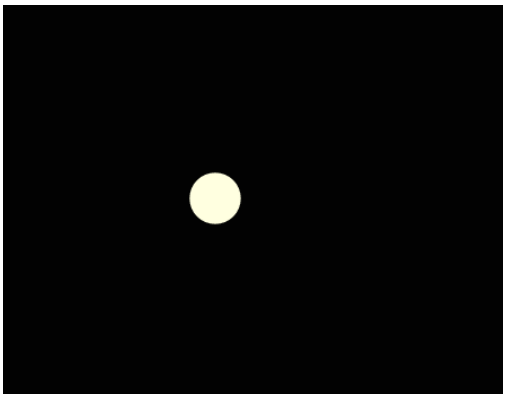


Figure 7: This is the scene we rendered in all of the experiments to produce a lens flare. To ensure that it is a encoded as a color image, the dot is a a faint yellow color over a black background.

in Algorithm 2.

4 Experiments

This project and the generated mapping functions were built off the Polynomial Optics library [5]. In each of these experiments, we simulated light traveling through an achromatic lens system, Figure 7 shows the image we used as a basis to render the lens flares in the subsequent experiments. We simulated all the paths of light through this lens system that consisted of two and four reflections, and generated polynomials of degree three. Figure All trials were run on a 2.3GHz Intel Core i7, with 1 processor and 4 cores. All timing data was taken using Instruments, a program that comes bundled with Apple's Xcode. Within Instruments, the Time Profiler was used to monitor the process computing the lens flares. We are only interested in measuring how long it takes to compute the polynomial maps. The reasoning behind this is that everything done prior to these evaluations can be considered preprocessing. If we were to implement something like this in real time, the construction and optimization of all the polynomial maps can be done once and then stored to be reused whenever a lens flare needs to be rendered. Instruments breaks down the amount of time a process spends in each function call, so we only recorded the "self" time of the EVALUATE function.

One improvement that was made that reduced the time it took to evaluate polynomials was the writing

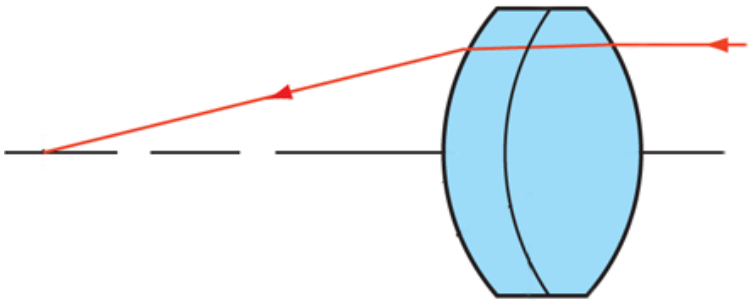


Figure 8: This is an achromatic lens that was used for our testing. It consists of two different lenses and has three different interfaces that we can reflect/refract light through. Image courtesy of [26].

of a new exponentiation function. Since the most expensive operation when evaluating these polynomials is the exponentiation of each of the input variables, this procedure was optimized, reducing the amount of time it took to evaluate a series of polynomial maps from $\approx 120,000$ ms to $\approx 40,000$ ms when no other approximations were introduced.

Another approach that was considered, but not implemented was Horner's method [27]. Horner's method is a way of rearranging a polynomial so that we only have to compute a reduced number of multiplications and additions in order to evaluate it; no exponentiations need to be computed. This method was not implemented because we thought the overhead of parsing through the information for each polynomial would outweigh any benefits we might see with this algorithm.

4.1 Specifics for Removing Terms

When we removed terms from the polynomials, we did so in two manners. The first was to remove terms based on a threshold, and if the coefficient of a term was below that threshold, then that term was removed. The values the threshold took on in this case were 1E-10, 1E-15, 1E-17, 1E-19, and 1E-21.

The second approach was to consider the maximal values the input variables to the polynomials could take on. Every term in each polynomial was evaluated using these maximal values and removed if the resulting number was below a set threshold. We used a range of thresholds from 1E-2 to 1E2 and incremented the threshold by 3 dB/decade.

4.2 Specifics for Memoizing

In an effort to further speed up the process of evaluating polynomials, we also memoized the exponentiation of variables. Again, there were two different implementations of memoization. The first was a more active version of memoization, where, as we evaluated different exponentiations of input variables within the terms of the polynomial, we check to see if that particular exponentiation has been computed before or not. If the value was not computed before, then it was computed and stored in an array, and if it was computed before, it was looked up in the array and substituted into the computation instead of having to compute the exponentiation directly.

In our second approach to memoizing the data, we computed all of the possible exponentiations of each variable prior to evaluating the polynomials. These precomputations are stored in an array, and whenever they are needed within the evaluation of a term, they are retrieved from this array.

4.3 Specifics for Non-Linear Least Squares Adjustments

To adjust the polynomials, we used the levmar library [17], which implements the Levenberg-Marquardt minimization algorithm. The algorithms to produce the necessary functions required for this library are described in Section 3.2.1. In addition to this, the library requires several other parameters passed to it (the names for parameters are the same names used in the documentation for this library):

p, m: This array holds our initial parameter estimates. In our case, we are adjusting the coefficients of the polynomials, so this holds all of the possible coefficients we might have. If a particular term does not exist within a polynomial, the coefficient is entered as a 0. The size of this array should be m, the dimension of the parameter vector.

x, n: This array holds the "true" values for $x, y, dx,$ and dy our polynomials should ideally produce. The size of this array should be n, the size of our measurement vector.

$adata$: This array contains the generated input data points used in the calculations.

5 Results

Ideally, all of this work together should greatly reduce the amount of time we spend evaluating the polynomial maps. Below, we discuss how our approximation methods affect the performance of the system.

5.1 Improving Performance: Removing Terms and Memoizing

We used two techniques of manipulating the polynomial maps in order to speed up the evaluation of them, the first being to remove terms based on some set threshold and the maximal values of the input variables and the other to precompute and memoize all of the possible exponentiations of the input variables. Figure 10 shows the result from both methods. In this figure, we look at how long it takes to evaluate the polynomials versus the average number of terms we evaluate. The average number of terms we evaluate is a function of the threshold we set for the current trial.

5.1.1 Results from Removing Terms

Two different methods of removing terms were tried. One of the lead motivations for removing terms was the notion that some of the coefficients of the terms were incredibly small and thus these terms contributed very little to the output of evaluating the polynomials. An initial attempt of simply removing these terms based on their coefficient yielded no benefit in terms of a speedup in computing the polynomials. The time it took to consistently evaluate each of the polynomials was contained ranged from 40,000 to 45,000 ms. Additionally, this method produced images that did not even contain lens flare characteristics.

A second approach was implemented where we considered the maximal values the input variables could take on and scanned through the polynomial evaluating the terms and removing those that were below a set threshold. The results from these experiments are shown in the orange line in Figure 10. As we would expect, removing terms shows a linear decrease in the amount of time needed to evaluate the polynomials as we increase the threshold and remove more and more terms.

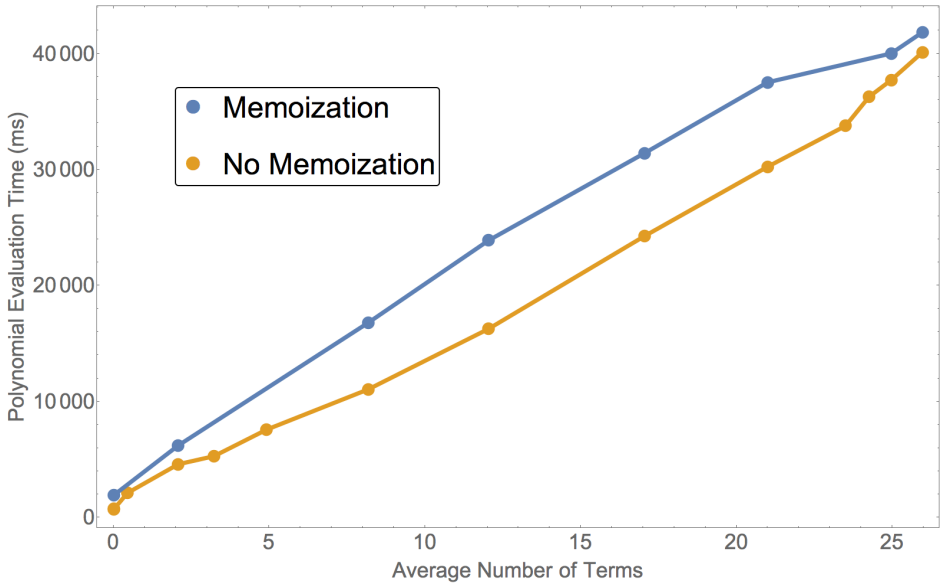


Figure 9: The orange line is the time it took to evaluate the polynomial when we remove terms, and the blue line is when we actively memoized data. With this method of memoization, the time it takes to evaluate the polynomials increased relative to just removing terms.

5.1.2 Results from Memoizing

A second method of evaluating the polynomials is to memoize the data and remove terms. The first implementation of removing terms involved active memoization. The overhead of checking to see if we already computed a value or not was too cumbersome and caused the evaluation of the polynomials using this techniques to be strictly worse than if we were to just remove terms. Figure 9 shows the results of this implementation. One explanation for this is that there is overhead with branching due to the involved if statements in this implementation.

The second implementation was to compute all of the exponentiations for each of the input variables and then look them up as needed in the evaluation process. The results from this implementation are shown in the blue line in Figure 10. Here, we see an immediate decrease in the time it takes to evaluate the polynomial when we keep all of the terms in the polynomial. These faster polynomial evaluation times continues until the overhead of computing all of the exponentiations outweighs the time spent evaluating

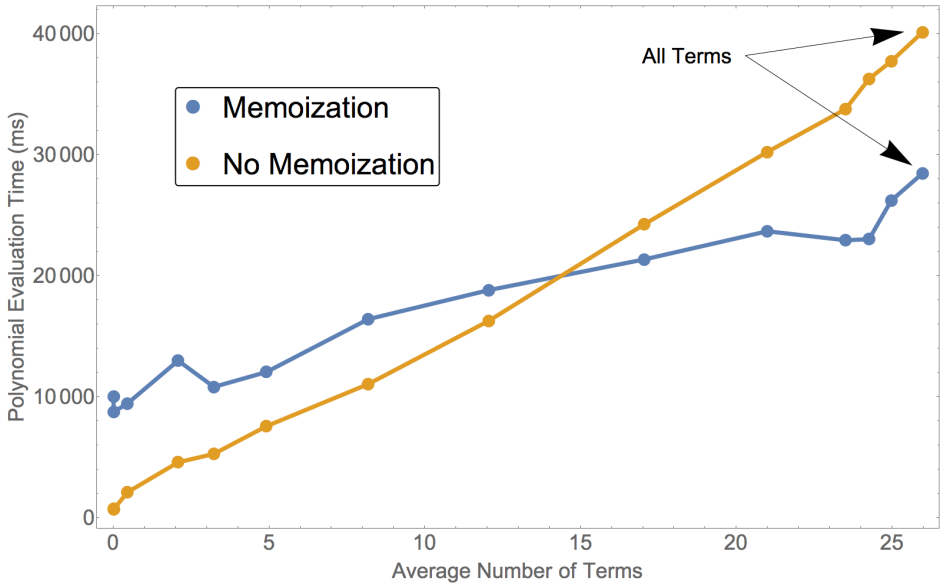


Figure 10: Here we look at two different procedures: the orange line is the time it took to evaluate the polynomials when we remove terms, while the blue is how long it took to evaluate the polynomials as we remove terms and memoize the exponentiations. Unsurprisingly, as we remove terms, we see a linear decrease in the amount of time it takes to evaluate the polynomials. Interestingly, once we start to memoize the data, we notice an immediate decrease in how long it takes to evaluate the polynomials, even when we keep all of the terms. This benefit, however, disappears at a certain point where the overhead of precomputing all of the exponentiations outweighs the number of times we reuse values.

the polynomial and looking up values.

We would assume that with larger polynomials (polynomials that have more terms and a higher degree) that we would notice a similar reduction in evaluation time from memoizing data or even a more pronounced reduction compared to evaluating the polynomials normally. Surely there would be more overhead in precomputing exponentiations if there were more variables and higher possible degrees, but the probability of the same terms being repeated becomes greater as the polynomials gets longer. So, this slight increase in overhead would be worth the reduction in evaluation time, especially when we do little to no approximations. We would still expect, however, that there is a point we can reach where memoizing the exponentiations is more work than the act of computing the polynomials.

5.2 Improving Fidelity: Non-Linear Least Squares Adjustments

Finally, we tried to improve the accuracy of these mapping functions by adjusting the coefficients of the polynomials. Ideally, this method should alter the coefficients, even if it is by some small amount. However, when we ran a non-linear least squares fit using the levmar library, the adjusted coefficients matched exactly to the original coefficients of the polynomials.

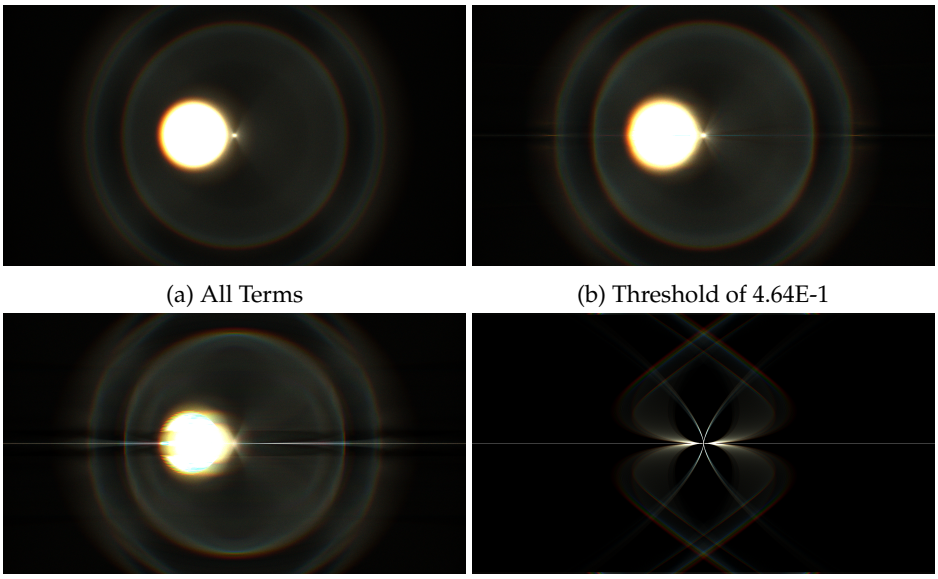
There are several explanations as to why this might be the case. The first reason why our implementation does not alter the polynomials could be because of the ground truth that we use. The majority of the ground truth values we calculate start to differ from the values the polynomials already output beginning at the second or third decimal place. It could be that the polynomials we already have are close enough to the minimal solution. It could also be that our ground truth is too similar to what our polynomials already produce and do not actually represent what physically happens within a lens system. Hanika and Dachsbacher [1] use an analytic ground truth obtained from ray tracing through their lens system. This ground truth might differ more from what our mapping functions already produce and actually alter our polynomial maps after we pass them through levmar.

This also might indicate that the results Hanika and Dachsbacher report are not that substantial. Within our system, we noticed no updates to our equations, so, even with a larger, more complex system, performing a non-linear least squares adjustment might alter the polynomial maps by an insignificant, unnoticeable amount such that any effort in this regard is not worth it.

Finally, it could be that the system and image we used were not detailed enough to need fixing. We used a very simple image and a minimal lens system, so it could mean that we could very accurately approximate how light would behave in this situation with polynomials maps. If we tried this same approach simulating a professional camera system with a busier image, it might be harder to approximate using these polynomial maps, and there might be several adjustments levmar could make.

5.3 Final Images

The images we rendered using these methods are shown in Figure 11, and Appendix A shows a more complete set of lens flares rendered with various thresholds.



(a) All Terms (b) Threshold of 4.64E-1

(c) Threshold of 1 (d) Threshold of 1E1

Figure 11: Shown above are a series of rendered lens flares made with a variety of different thresholds. Both reducing terms and memoization produced the same images. Here we see the effect on the image quality as we increase the threshold, i.e., removing terms from the polynomial.

One major question after producing these images is which of these look like lens flares and which do not? Many of the renders, when we have a low threshold, look very similar to the original image produced with all terms. The minor defect introduced is a line that crosses horizontally through the middle of the image. Images 11a and 11b both look like convincing lens flares. It is when we start using a threshold of 1 or greater do we begin to see major defects in the produced images in terms of distortions in the light source itself to patterns that are not naturally found (see Figure 11d). Appendix B shows some of the rendered images compared to the original to emphasize the differences between the two.

5.3.1 Image Evaluation

A topic that not much of the literature covers is how to evaluate the produced images. We want to measure how much a rendered image looks like a lens flare. Hanika and Dachsbacher [1] looked at how similar two images were by taking a pixel difference of the two images. A pixel difference looks at the pixels of either image and either subtracts the two values or performs some other distance measurement to produce

a new image. To this effect, we can exaggerate and highlight the differences between two images. Figure 12 shows a few difference images between the image rendered with all the terms in the polynomial and images rendered when we removed polynomials. Appendix C shows a wider collection if these images. But, while this is good at showing regions where images differ, it fails to give a single numerical evaluation of how an image compares to some original.

One approach to generating a numerical evaluation is to treat each image as its own vector of pixel values in a vector space, $\mathbf{v} \in (\Re^3)^{w \times h}$ where every pixel contains an RGB value, and the number of pixels we have is the width, w, times the height, h, of the image. With this, we can use different metrics to evaluate how far off two images are from one another.

A metric a more general construct than a norm, which is used to assign a size to a given vector within a vector space. Given a vector $\mathbf{x} \in \Re^n$, the n-dimensional Euclidean Space, the one norm of $\mathbf{x}$ is

$$\|\mathbf{x}\|_1 := \sum_{i=1}^{n} |x_i|, \tag{5.1}$$

the two norm of $\mathbf{x}$ is

$$\|\mathbf{x}\|_2 := \sqrt{\sum_{i=1}^{n} |x_i|^2}, \tag{5.2}$$

and the infinity norm of $\mathbf{x}$ is

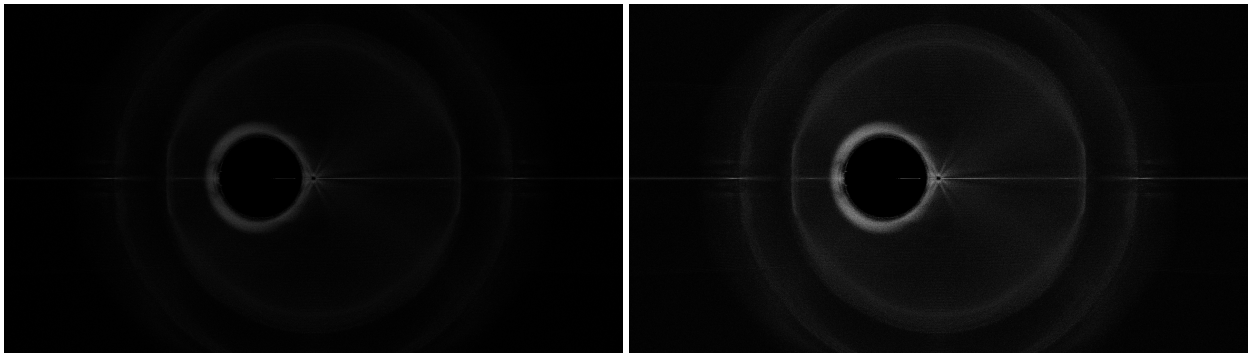
$$\|\mathbf{x}\|_\infty := \textsc{Max}(|x_1|, |x_2|, \ldots, |x_n|). \tag{5.3}$$

From these norms, we can define the metrics that tell us the distance between two vectors. The taxicab metric, built off the one norm, is defined as

$$d_1(\mathbf{x}, \mathbf{y}) := \|\mathbf{x} - \mathbf{y}\|_1, \tag{5.4}$$

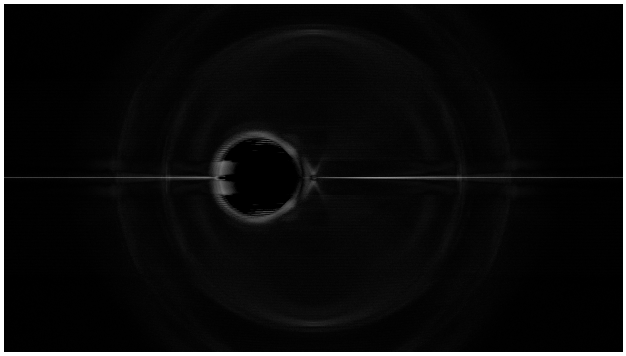
where $\mathbf{x}$ and $\mathbf{y}$ are the vectors within the vector space we are interested in quantifying; the Euclidean distance, built off the two norm, is defined as

$$d_2(\mathbf{x}, \mathbf{y}) := \|\mathbf{x} - \mathbf{y}\|_2; \tag{5.5}$$

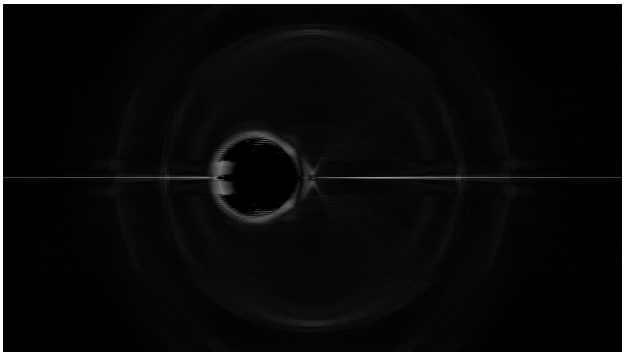


(a) Difference with Threshold of 4.64E-1

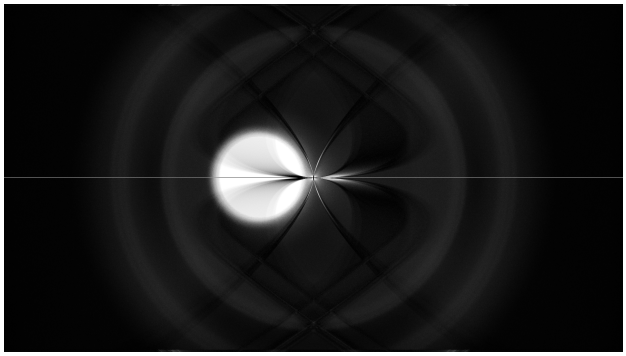
(b) Scaled Difference with Threshold of 4.64E-1



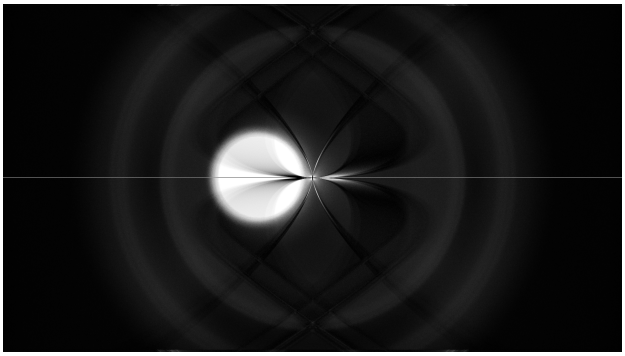
(c) Difference with Threshold of 1



(d) Scaled Difference with Threshold of 1



(e) Difference with Threshold of 1E1



(f) Scaled Difference with Threshold of 1E1

Figure 12: A Euclidean distance was taken between each of the rendered images generated with a threshold with respect to the image rendered with all of the terms in the polynomial. These are the images shown on the left. Scaled images are shown on the right, and these are scaled so that the biggest difference appears as white, the smallest difference appears as black, and everything else spans between those two values. The images on the left show us which pixels are different between the two images, where the brightest tells us how large of a difference the two pixels has. The image on the right is just to exaggerate these differences so we can easily see them.

the infinity metric, built off the infinity norm, is defined as

$$d_\infty(\mathbf{x}, \mathbf{y}) := \|\mathbf{x} - \mathbf{y}\|_\infty. \tag{5.6}$$

In addition to these evaluations, we can also look at a cosine similarity between two images. From a mathematical sense, the cosine of the angle between two vectors is a measure from 0 to 1 of how parallel the vectors are, with 0 meaning the two vectors are perpendicular to one another and a value of 1 meaning they are parallel. The idea behind using this as a means of evaluating the images is that we know on a scale from 0 to 1 how close the images are to one another, as opposed to using the metrics that have values that can cover an almost arbitrary range of numbers that seemingly have no meaning. To compute this similarity, we use

$$\text{SIMILARITY} = \cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2}, \tag{5.7}$$

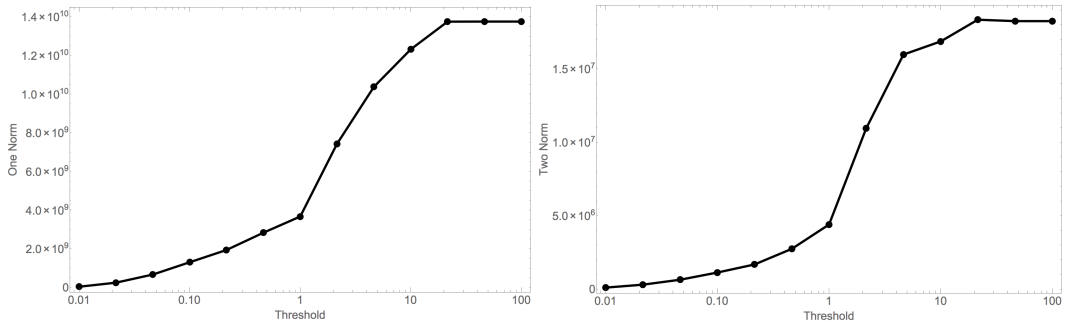
where θ is the angle between the two vectors and $\|\|_2$ refers to the magnitude of the vector it encloses.

Figure 13 shows how all four of these different methods faired at evaluating the images. For each series of tests, the image produced with all of the terms, Figure 11a, was used as the "true" image and every subsequent image was compared to that.

All of these measurements behave as we expected. For the one, two, and infinity norms, we would expect to have low values for the images generated with a low threshold and for this value to increase as we remove larger and larger terms. For the cosine similarity, we observe that we have values close to 1 for the images where we keep most of the terms and this value drops off as we increase the threshold.

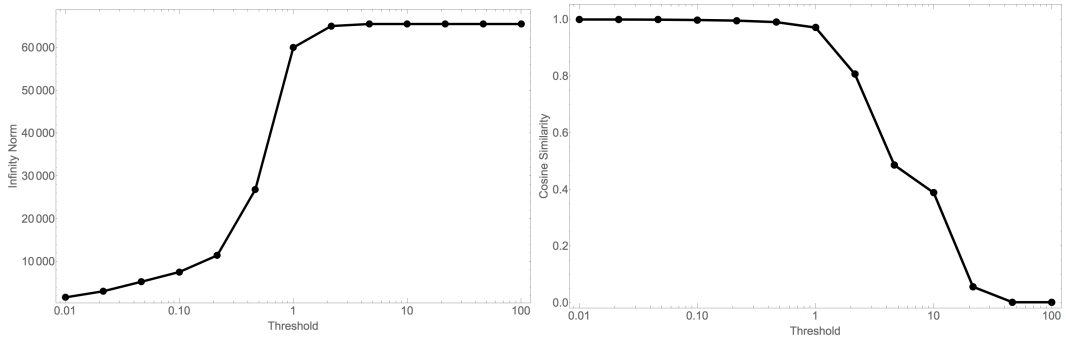
The infinity norm, however, seems to be the only measure that picks up on the tiniest changes between the images. The infinity norm is a measure of the largest pixel difference between two images. This measure works particularly well at identifying when defects are introduced into the images as we remove terms, and it seems to be more sensitive to these minor changes compared to the other methods presented.

Ideally we would hope to have a quick polynomial evaluation time and a low pixel difference with the rendered image. In other words, our approximation methods should significantly reduce the time it takes to evaluate the polynomials while still producing images of comparable quality to the true image. Figure



(a) One Norm vs. Threshold

(b) Two Norm vs. Threshold



(c) Infinity Norm vs. Threshold

(d) Cosine Similarity vs. Threshold

Figure 13: Here are four different ways we evaluate how close an image is to the original image rendered using all of the terms. Shown are the how the one norm, the two norm, the infinity norm, and the cosine similarity evaluate the similarity between two images. A large norm value means that the two images are very different from one another, and a low cosine similarity means the two images are very different.

14 shows these data points for the rendered image as we remove terms from the polynomial. Here we see that even the image formed with a threshold of 0.464 is an acceptable lens flare, albeit its slightly higher pixel difference. Nonetheless, the fact that computing these polynomials is $\approx 2.5\times$ faster than evaluating the polynomials with all the terms in them means we have made strides to improving the efficiency of generating lens flares.

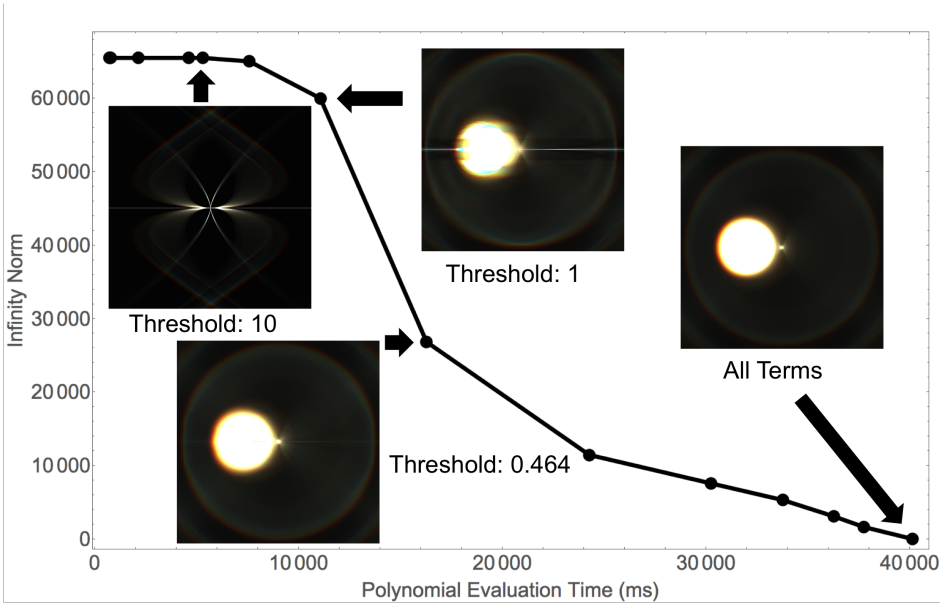


Figure 14: When we look at the largest pixel difference between rendered images (the infinity norm) and how long it took to evaluate the polynomials when we just remove terms, we would hope to find that we have a small evaluation time and a small pixel difference. The images produced with a threshold of 0.464 still look like a lens flare and achieve $\approx 2.5\times$ speedup in evaluation time compared to the image produced when we kept all of the terms.

6 Conclusion

Lens flares–and realistic rendering in general–are hard problems to tackle, but progress has been made towards understanding the difficulties in rendering lens flares realistically. We presented several methods that both help decrease the polynomial evaluation time and maintains a level of quality when approximations are introduced.

With regards to our first effort, some terms within a polynomial expression may contribute very little to the result of evaluating said polynomial. Thus, after we consider the values that the input variables can take, we can remove the terms that contribute the least to the final product by setting some threshold describing which terms to keep. In doing this, we managed to produce acceptable lens flares that could be computed $\approx 2.5\times$ faster than if we did not remove any terms within our particular system.

Another method we implemented to reduce the polynomial evaluation time is memoization. Using

this method, we precompute all of the exponentiations of the input variables and substitute them into the evaluation as needed. The benefit behind this is that we do not need to repeatedly compute the same exponentiations of variables if they appear more than once in the polynomial expression. This precomputation benefits our evaluation time up to a point. At this point, the overhead of memoizing all of the exponentiations outweighs the amount of time we save reusing values.

In an effort to make the polynomials encapsulate more detail of the system, we also adjusted the coefficients of the polynomials via a non-linear least squares fit. This used the Levenberg-Marquardt method of adjusting the coefficients.

Finally, we also describe a means of evaluating the fidelity of produced images. This looks at the infinity norm between two images. For our purposes, this is intended to measure how close an image produced with approximations and a more detailed image are to one another. The infinity norm measures the largest pixel value difference between the two images and, based on our experiments, serves as a good indication at when an image is drastically different from the original.

7 Future Work

While improvements were definitely introduced with this work, other methods and concerns should be addressed to add to this project. One of the main concerns with this project is floating point errors introduced by using a computer. Making sure that our computation is numerically stable, i.e., these values do not drift, is essential in future iterations of this project since the precision of every coefficient and every evaluation of variables is important.

In terms of new methods that can be applied to visual fidelity of rendered lens flares, there are several different additional implementations that can take place. Firstly, there are a multitude of different fitting methods, e.g. the Gauss-Newton Method, Powell's Dog Leg Method, etc., that can be used to adjust the coefficients of the polynomials. It would be interesting to see how each of these different fitting methods fair at improving the accuracy of the polynomials.

In a similar vein, there are several different methods for producing what we consider the ground truth used to adjust the polynomials. In this paper, we evaluate how light behaves in a lens system at every

point along its path, but Hankika et al. [1] ray traced through a lens system to render their ground truth image that they used to adjust the polynomials. First, this ray tracing implementation should be tested to see its effect on adjusting the polynomials compared to the ground truth we propose. Then, to speed up the production of the ground truth, different ray tracing techniques described by Veach [10] can be implemented to test their effect on the end results.

Finally, the method we render lens flares could itself be optimized. We currently do a quasi-importance sampling technique where we thoroughly sample only those portions of the scene that have a high luminosity. While this behavior is beneficial for our purposes, testing other sampling techniques and evaluation techniques could yield interesting results in terms of how quickly we can generate these lens flares. Similarly to the previous point, different Monte Carlo methods could be tested here so we can gauge their effectiveness.

8 Acknowledgments

I firstly thank my advisor, Professor Matthew Anderson of the Computer Science Department at Union College, for putting up with me for a year on this project. Working with him was a pleasure, and he provided insightful and meaningful ideas and comments. Secondly, I thank the professors of the computer science department and my computer science academic advisor, Professor Aaron Cass, for making my time here enjoyable and for making computer science something truly worthwhile. I also thank my parents, Dennis and Laura DiIorio, who have supported me throughout my life so that I could get to the point I am at today.

References

[1] Johannes Hanika and Carsten Dachsbacher. Efficient Monte Carlo rendering with realistic lenses. *Computer Graphics Forum (Proceedings of Eurographics)*, 33(2):323–332, April 2014.

[2] Matthias B. Hullin, Elmar Eisemann, Hans-Peter Seidel, and Sungkil Lee. Physically-based real-time lens flare rendering. *ACM Trans. Graph. (Proc. SIGGRAPH 2011)*, 30(4):108:1–108:10, 2011.

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

[4] Craig Kolb, Don Mitchell, and Pat Hanrahan. A realistic camera model for computer graphics. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 317–324, New York, NY, USA, 1995. ACM.

[5] Matthias Hullin, Johannes Hanika, and Wolfgang Heidrich. Polynomial Optics: A construction kit for efficient ray-tracing of lens systems. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering)*, 31(4), July 2012.

[6] Peter Shirley, Michael Ashikhmin, and Steve Marschner. *Fundamentals of Computer Graphics*. CRC Press, 2009.

[7] Steven J Gortler. *Foundations of 3D Computer Graphics*. MIT Press, 2012.

[8] Kevin Beason. smallpt: Global illumination in 99 lines of C++. `http://www.kevinbeason.com/smallpt/`, November 2010.

[9] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[10] Eric Veach. *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, 1997.

[11] Alexander Keller, Simon Premoze, and Matthias Raab. Advanced (quasi) Monte Carlo methods for image synthesis. In *ACM SIGGRAPH 2012 Courses*, SIGGRAPH '12, pages 21:1–21:46, New York, NY, USA, 2012. ACM.

[12] Nicholas Metropolis. The beginning of the Monte Carlo method. *Los Alamos Science*, (15):125, 1987.

[13] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2003.

[14] Tobias Ritschel, Matthias Ihrke, Jeppe Revall Frisvad, Joris Coppens, Karol Myszkowski, and Hans-Peter Seidel. Temporal glare: Real-time dynamic simulation of the scattering in the human eye. *Computer Graphics Forum*, 28(2):183–192, 2009.

[15] Harrisonn William Robert Griffin. Untitled. Picture, August 2014.

[16] Sungkil Lee and Elmar Eisemann. Practical real-time lens-flare rendering. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering)*, 32(4):1–6, 2013.

[17] M.I.A. Lourakis. levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++. `http://www.ics.forth.gr/~lourakis/levmar/`, Jul. 2004.

[18] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 419–428, New York, NY, USA, 1995. ACM.

[19] Thomas Kollig and Alexander Keller. Efficient multidimensional sampling. *Computer Graphics Forum*, 21(3):557–563, 2002.

[20] Mark J. Kilgard. Fast opengl-rendering of lens flares. `http://www.opengl.org/archives/resources/features/KilgardTechniques/LensFlare/`, July 2000.

[21] Dean Sekulic. Efficient occlusion culling. *GPU Gems*, pages 487–503, 2004.

[22] Chris Oat. A steerable streak filter. *ShaderX 3 Advanced Rendering with DirectX and OpenGL*, pages 341–348, 2004.

[23] Ted Alspach. Vector-based representation of a lens flare, April 17 2007. US Patent 7,206,725.

[24] Arash Keshmirian. A physically-based approach for lens flare simulation. Master's thesis, University of California, San Diego, 2008.

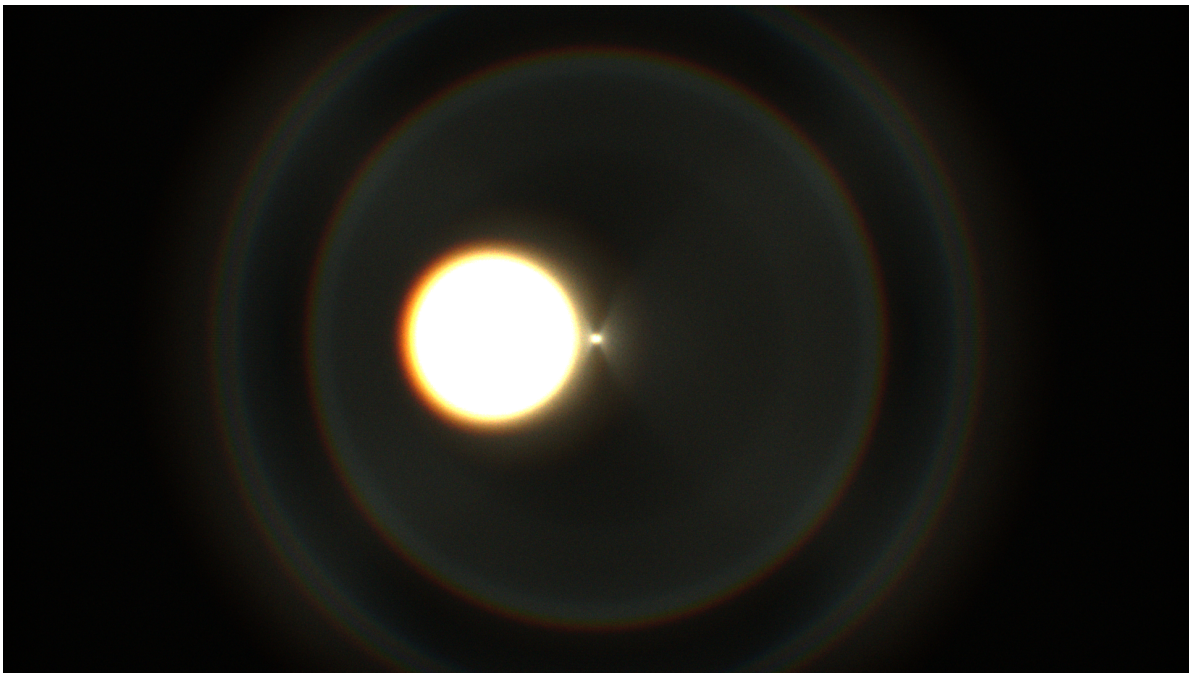
[25] Se Baek Oh, Sriram Kashyap, Rohit Garg, Sharat Chandran, and Ramesh Raskar. Rendering wave effects with augmented light field. *Computer Graphics Forum*, 29(2):507–516, 2010.

[26] Edmund Optics Inc. Why use an achromatic lens?, 2014.

[27] Donald E Knuth. *The Art of Programming*, volume 2. Addison Wesley, Reading, MA, 1981.

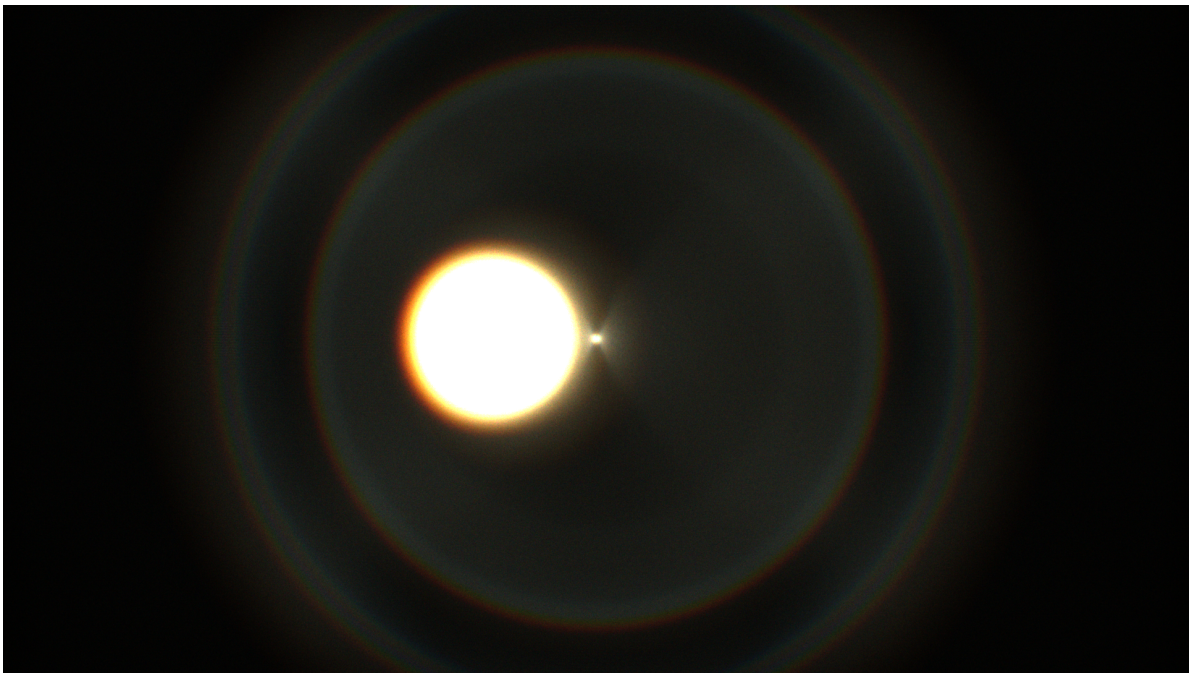
Appendices

A Rendered Lens Flare Images

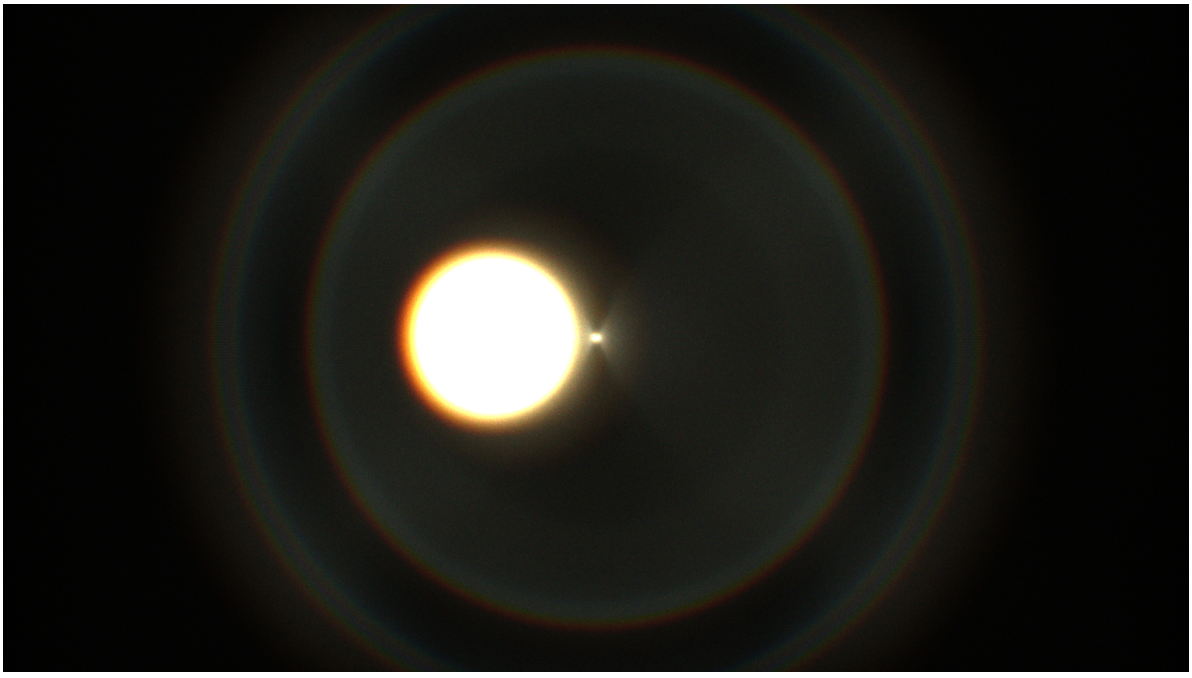


(a) All Terms

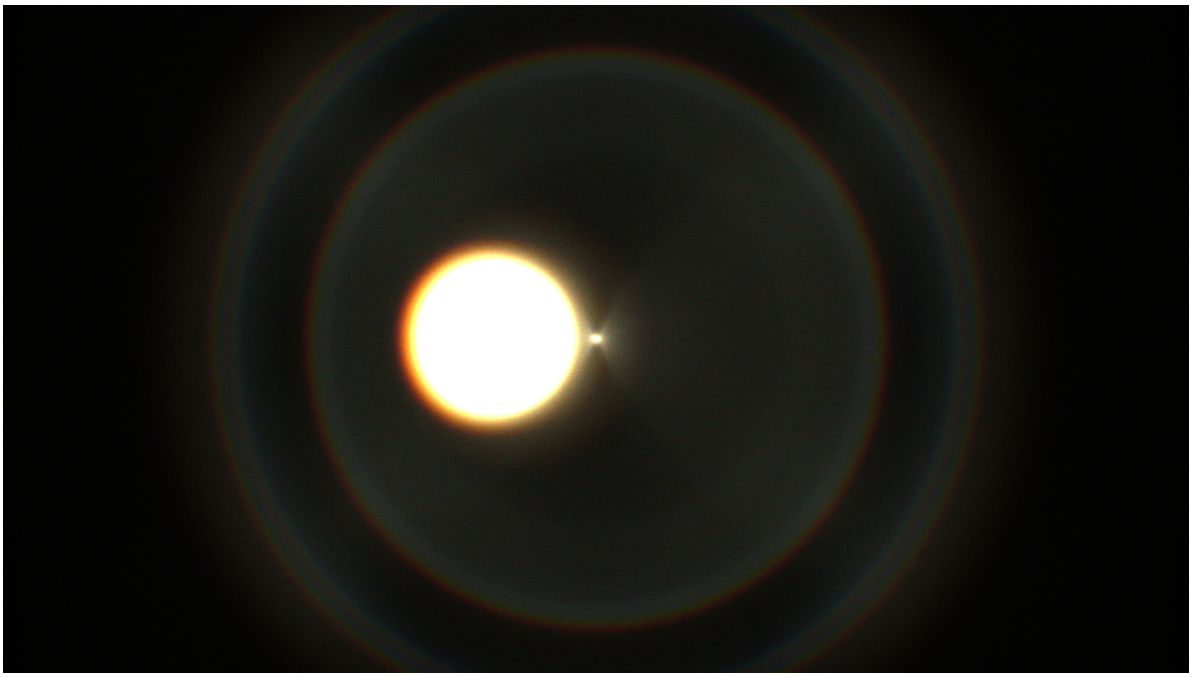
Figure 15: Here we show all of the lens flares we rendered using the Polynomial Optics library and the methods mentioned within this paper. This is an extension of Figure 11 and shows more flares at different thresholds.



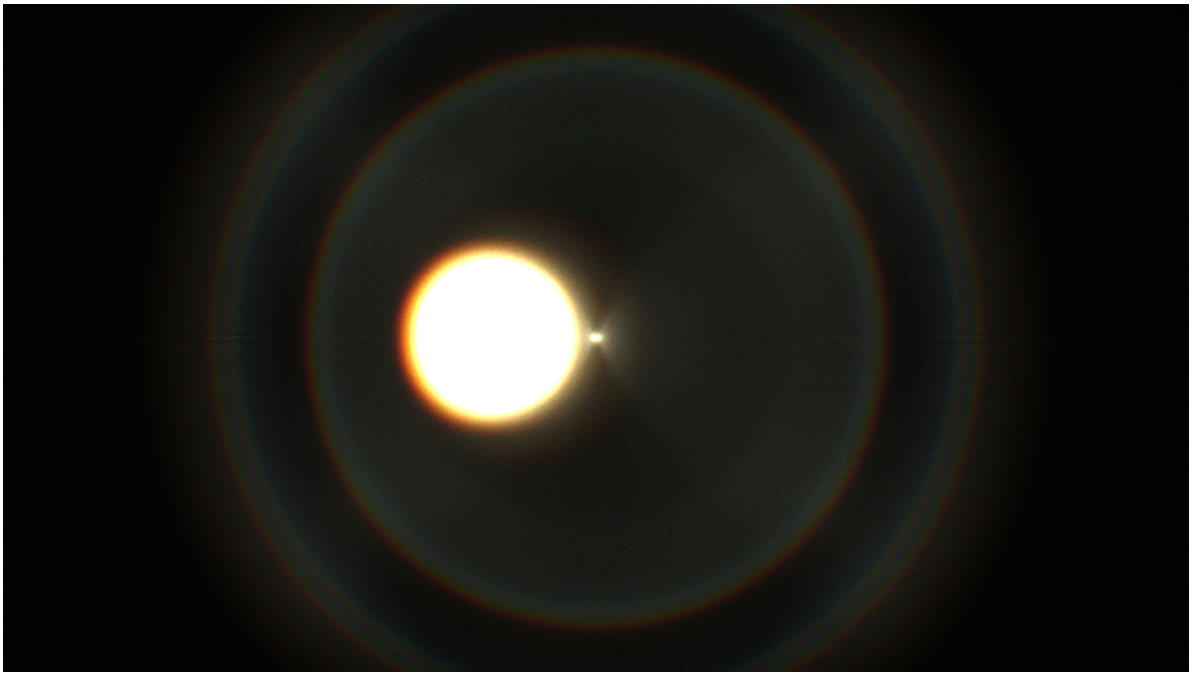
(b) Threshold of 1E-2



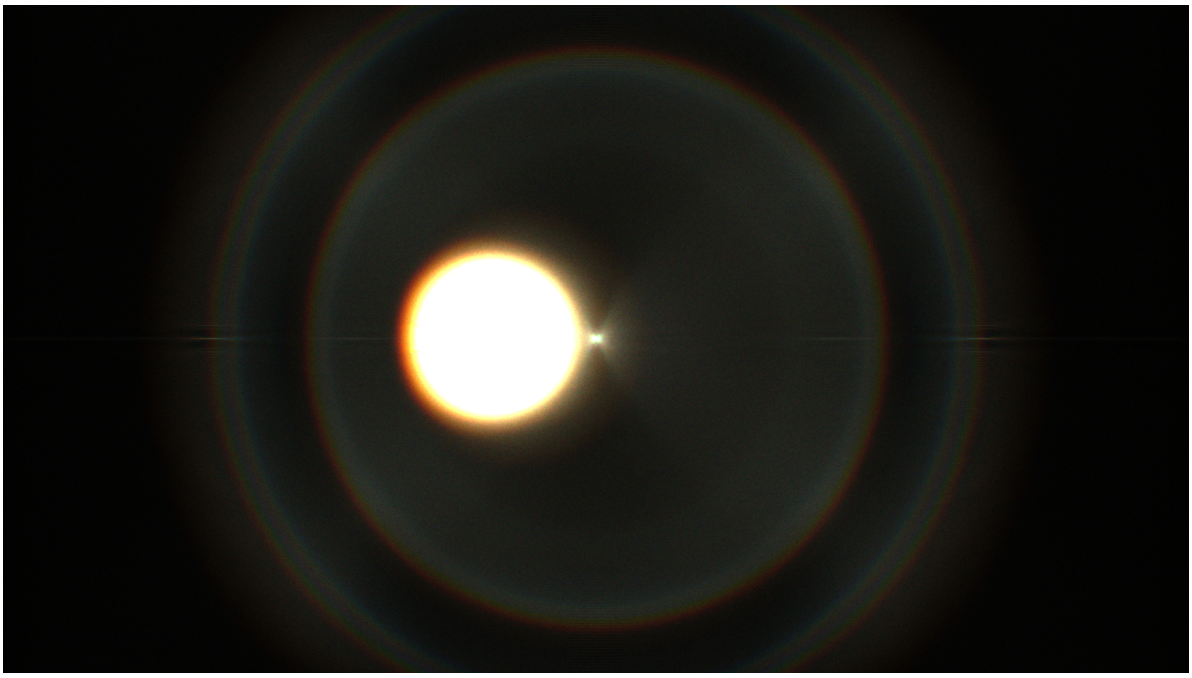
(c) Threshold of 2.15E-2



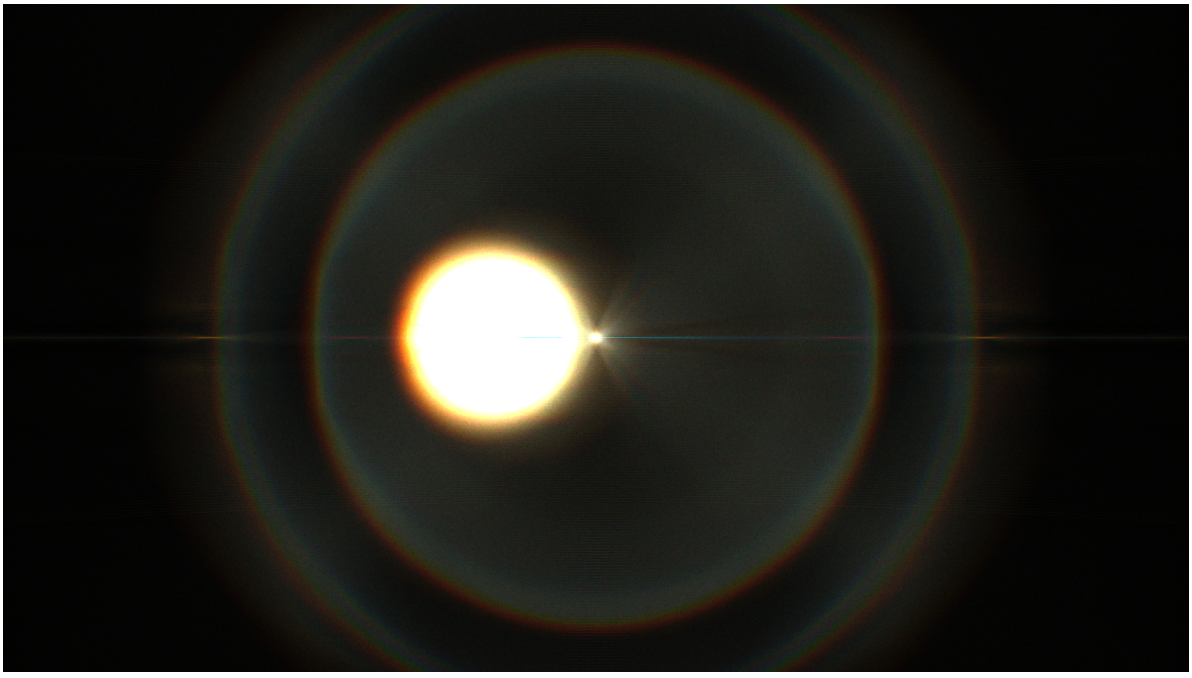
(d) Threshold of 4.64E-2



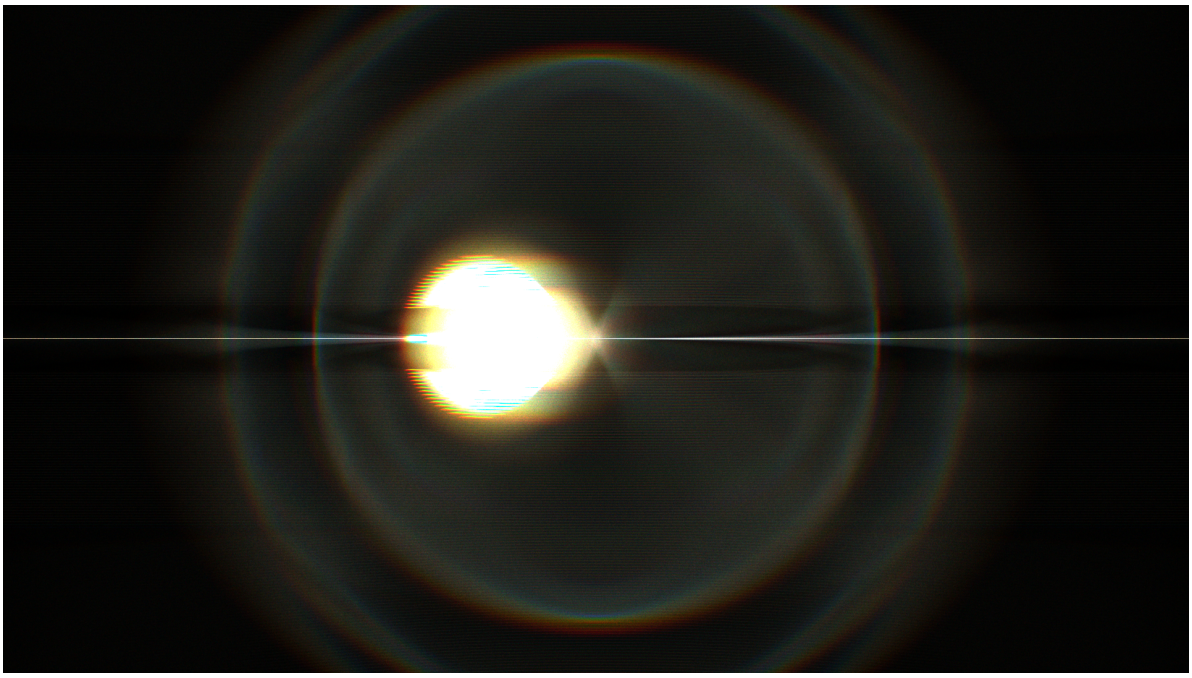
(e) Threshold of 1E-1



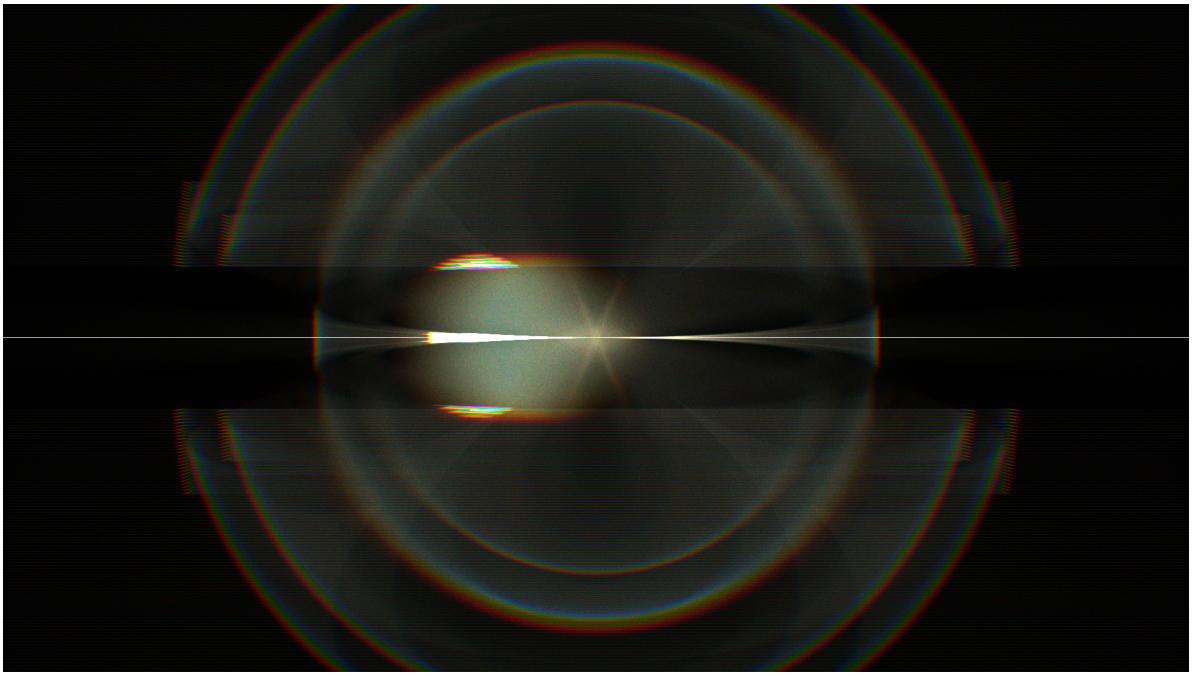
(f) Threshold of 2.15E-1



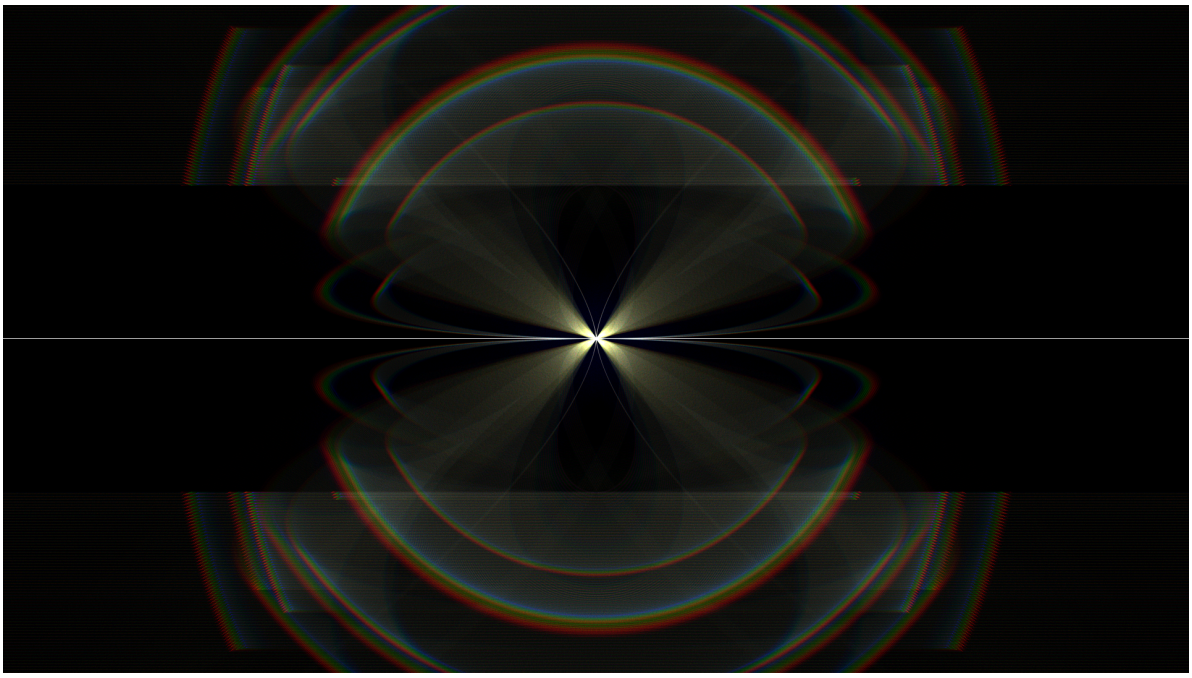
(g) Threshold of 4.64E-1



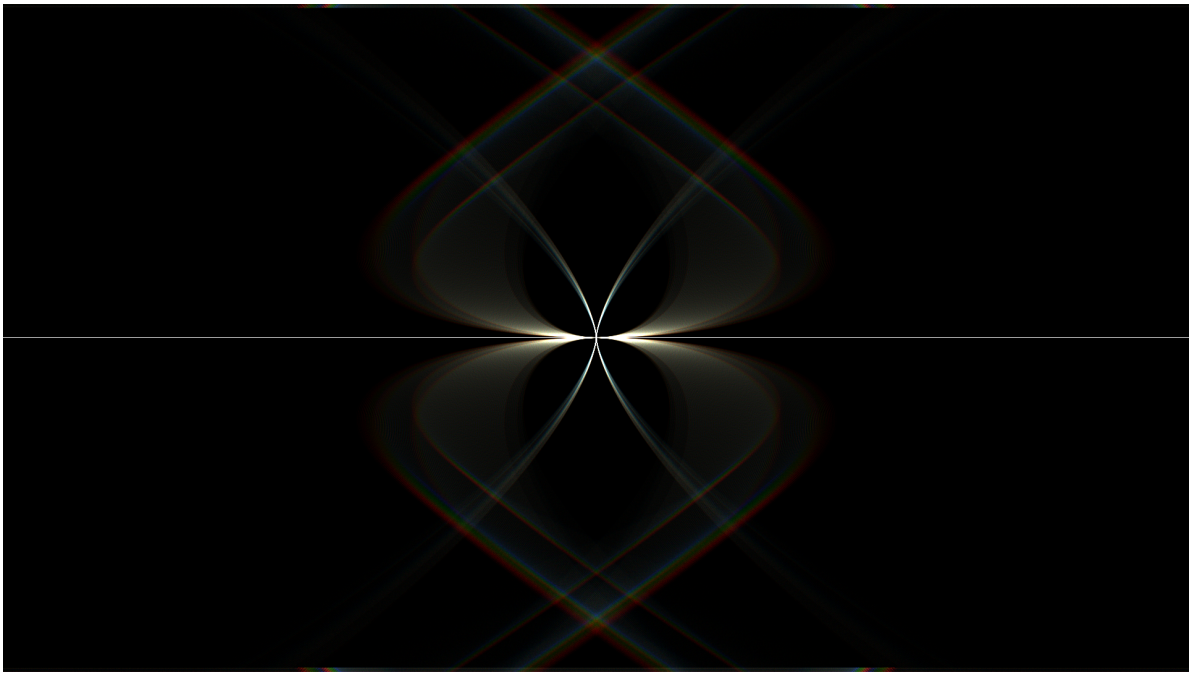
(h) Threshold of 1



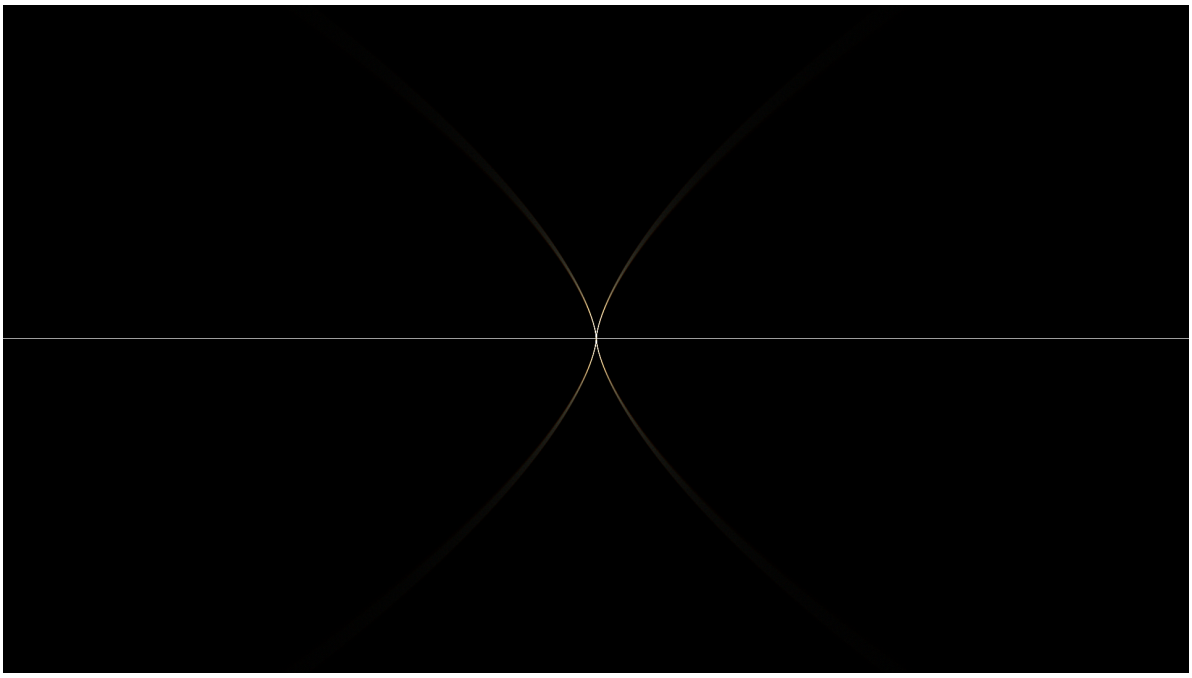
(i) Threshold of 2.15



(j) Threshold of 4.64



(k) Threshold of 1E1

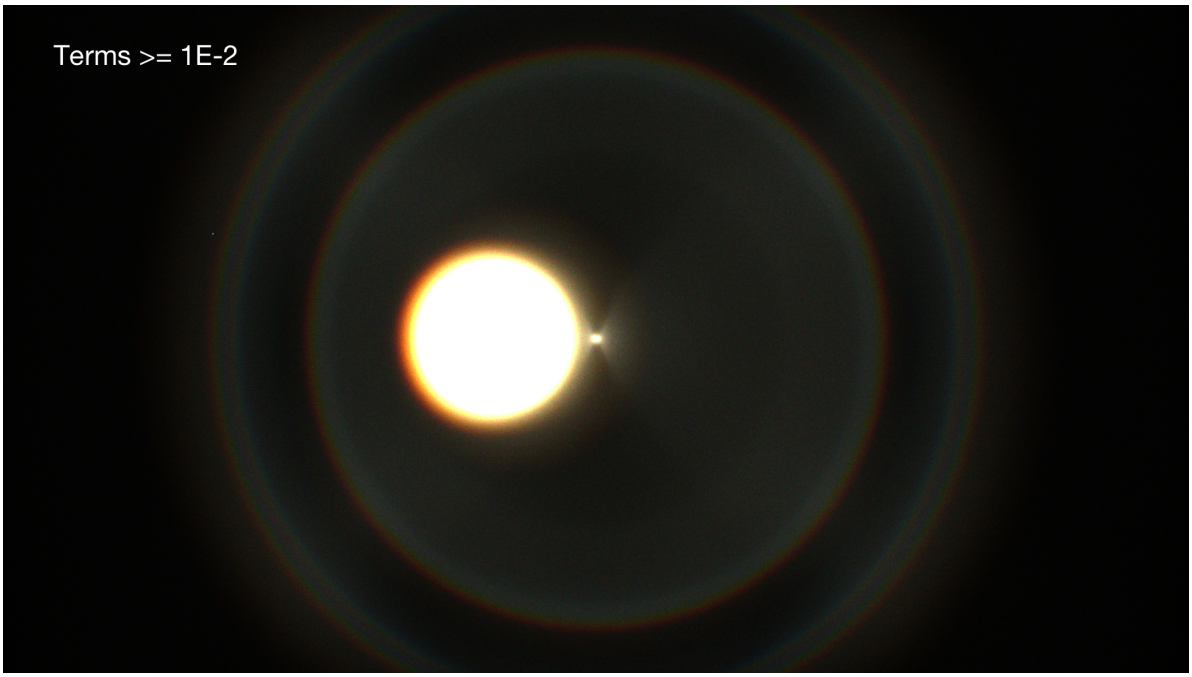


(l) Threshold of 2.15E1



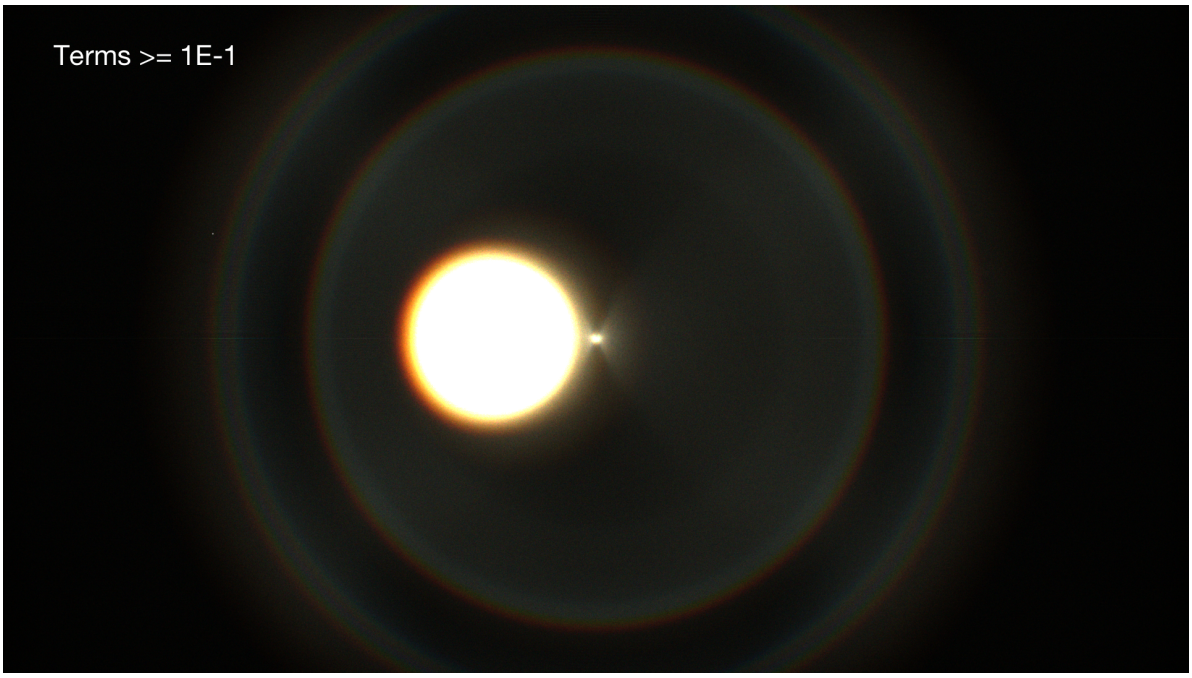
(m) Threshold of 4.64E1 and 1E2

B Lens flare Comparisons

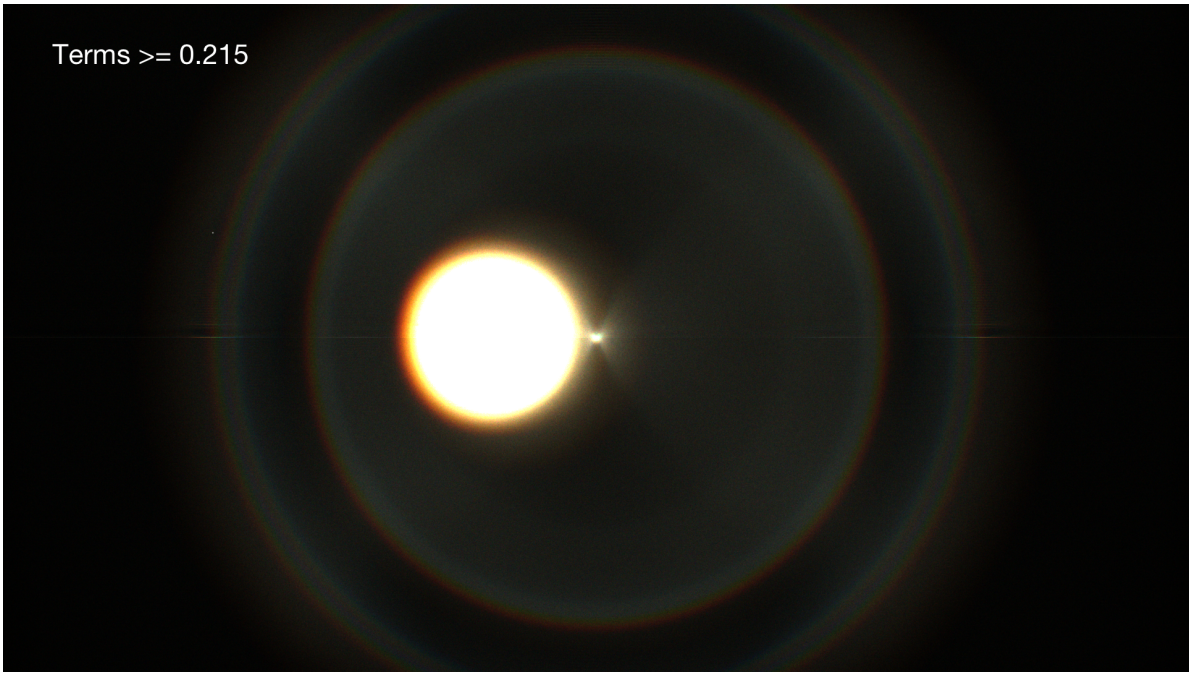


(a) Threshold of 1E-2 Compared with All Terms

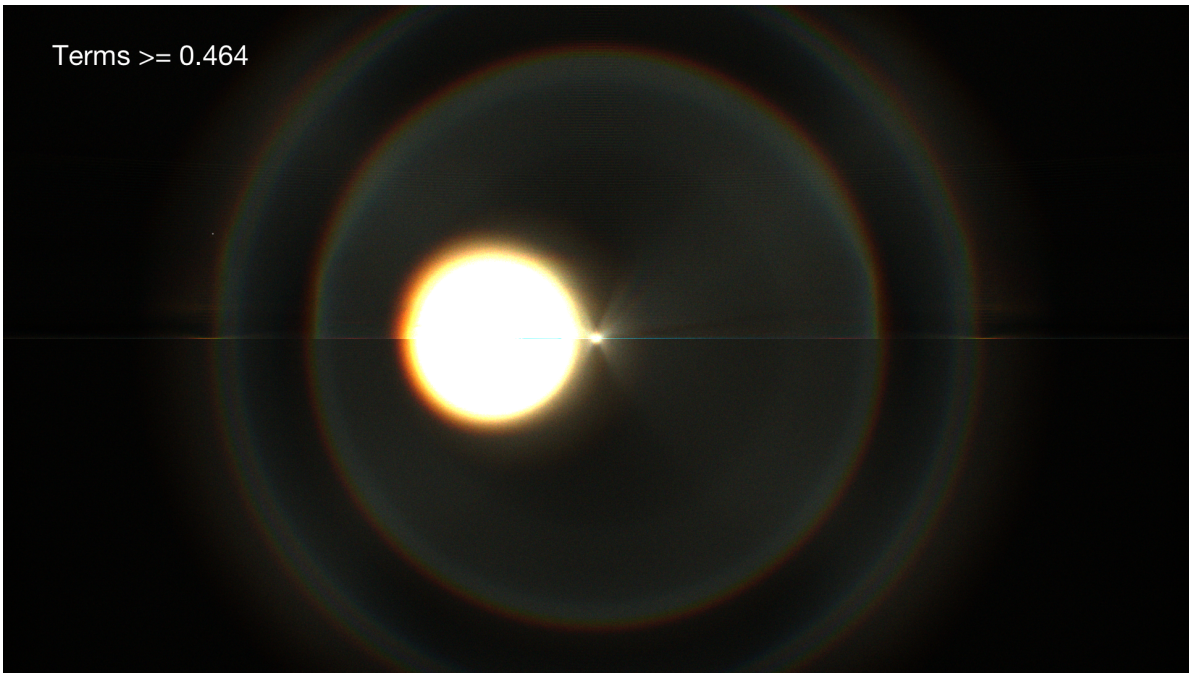
Figure 16: Here are several comparison images. The bottom halves of all of these images is the image rendered when all of the terms were kept in the polynomial, Figure 11a. The top halves of these images are the images produced when we remove terms from the polynomial. The threshold of these top halves are indicated by their caption, and a full image of these lens flares can be found in Figure 15.



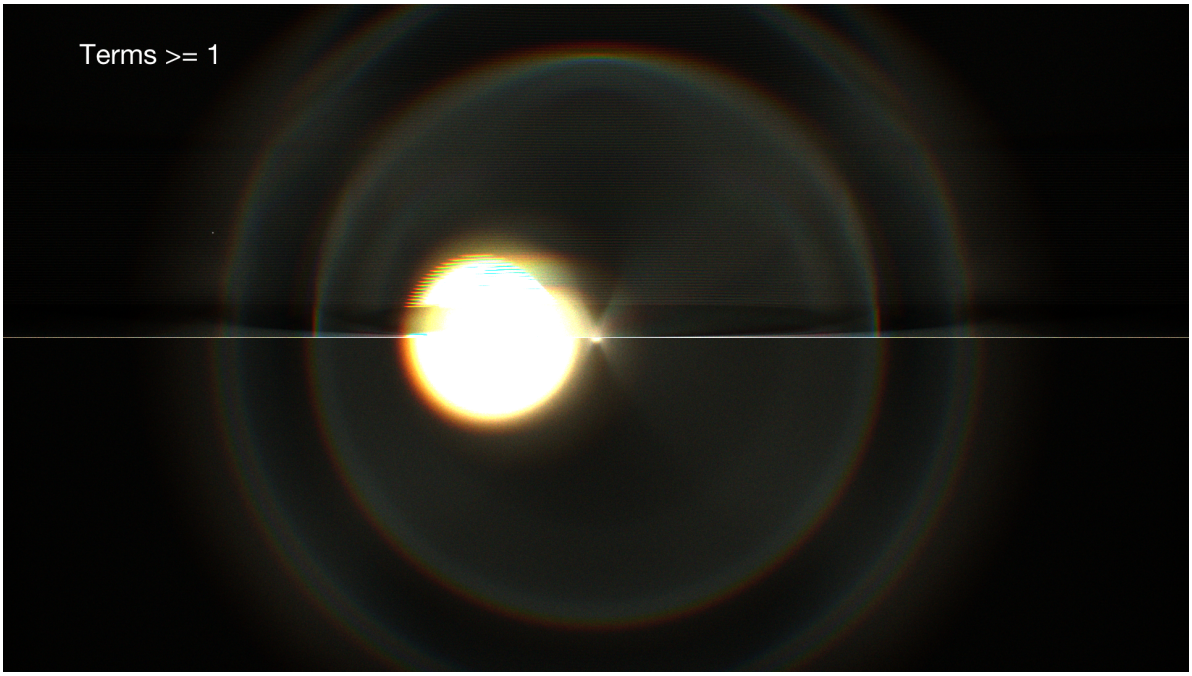
(b) Threshold of 1E-1 Compared with All Terms



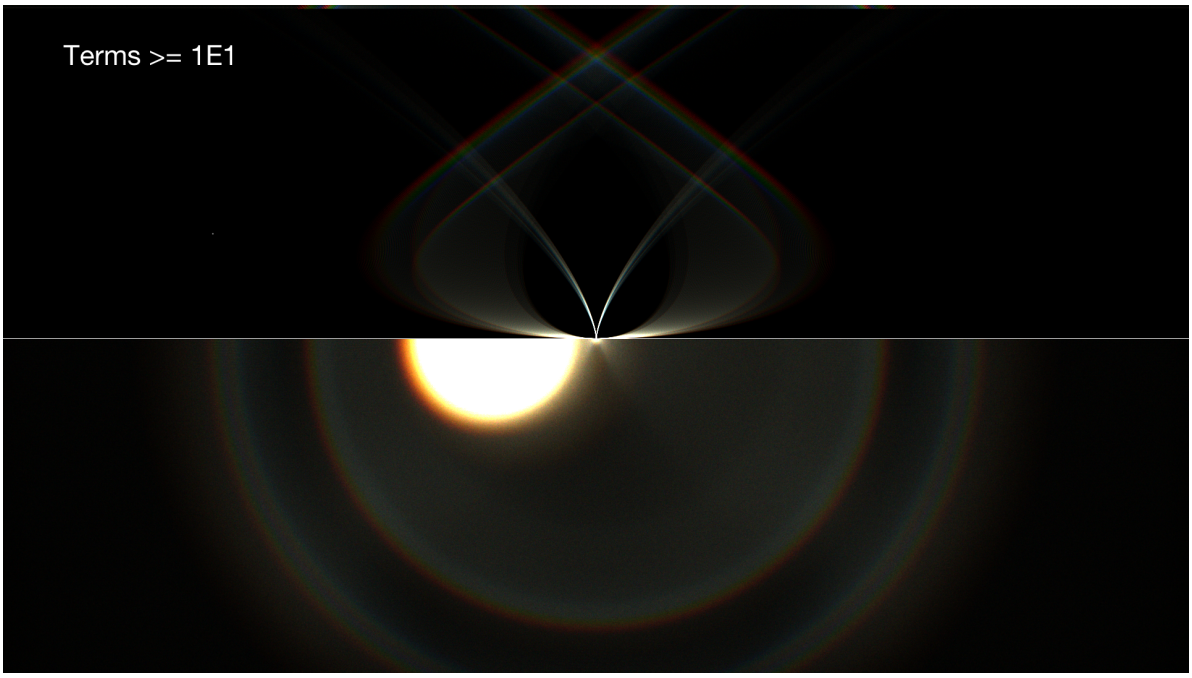
(c) Threshold of 2.15E-1 Compared with All Terms



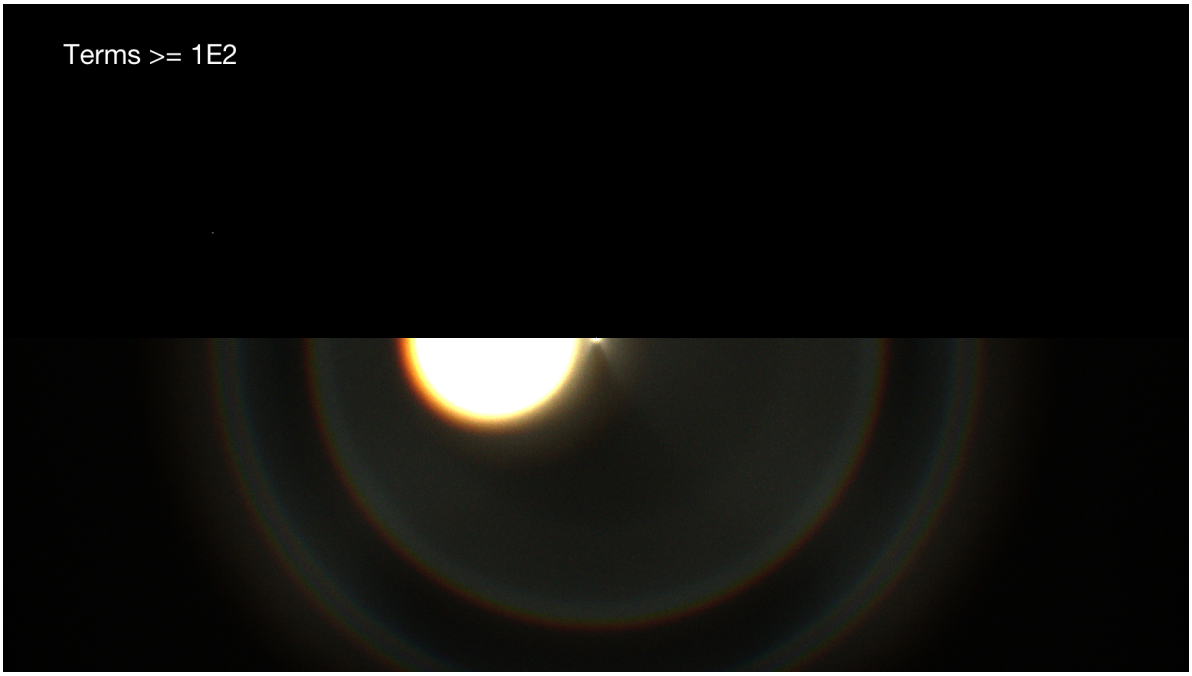
(d) Threshold of 4.64E-1 Compared with All Terms



(e) Threshold of 1 Compared with All Terms

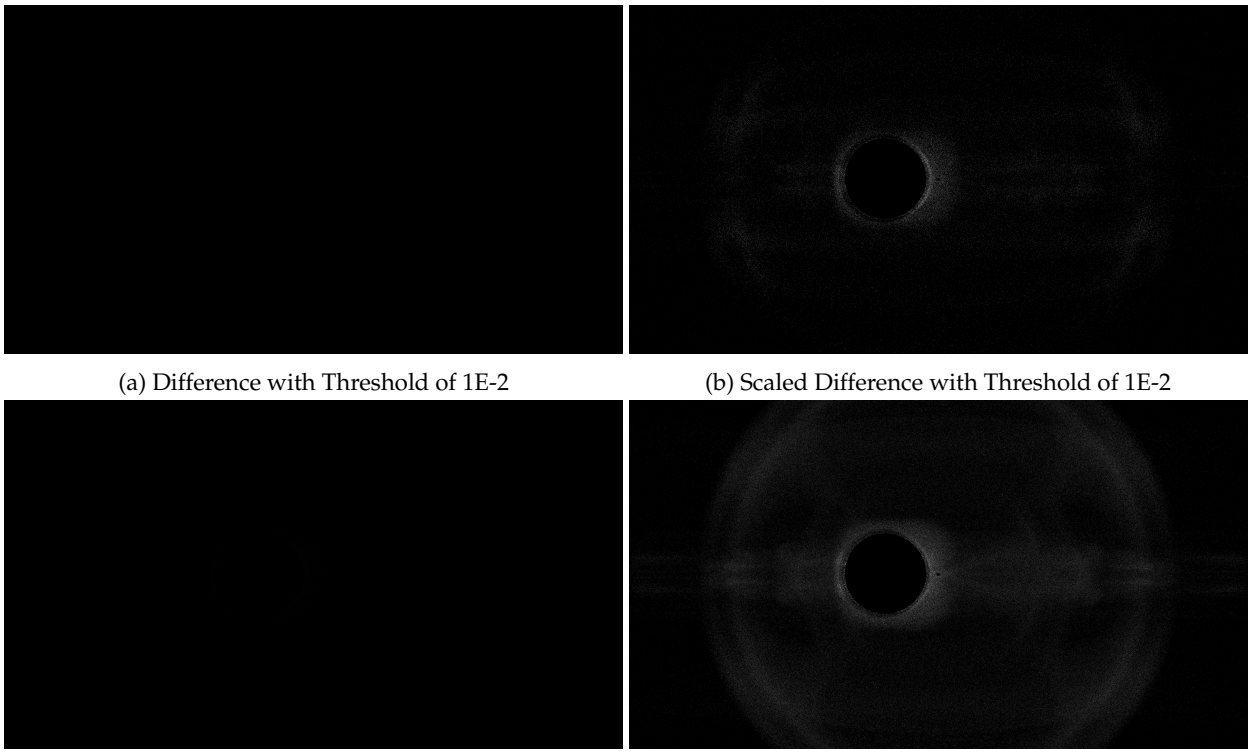


(f) Threshold of 1E1 Compared with All Terms



(g) Threshold of 1E2 Compared with All Terms

C Difference Images of Rendered Lens Flares



(a) Difference with Threshold of 1E-2

(b) Scaled Difference with Threshold of 1E-2

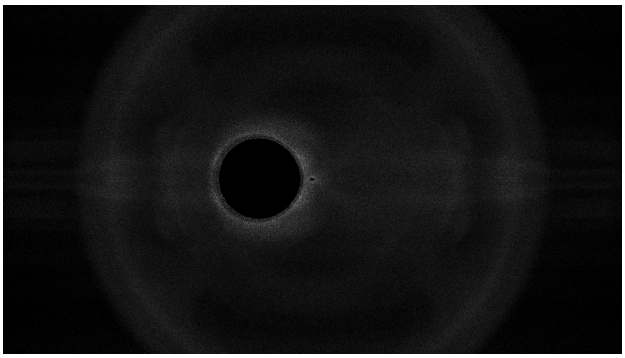
(c) Difference with Threshold of 2.15E-2

(d) Scaled Difference with Threshold of 2.15E-2

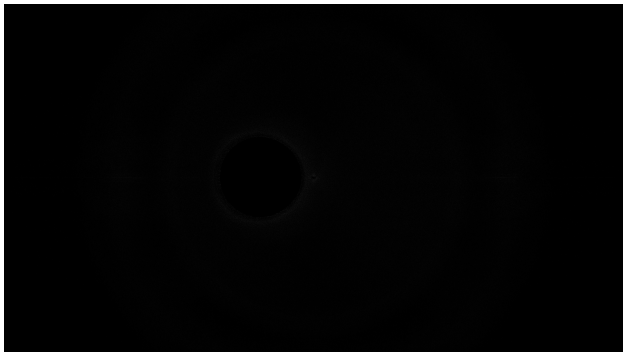
Figure 17: Here we show all of the comparison images between the images in Figure 15 and Figure 11a. This is an extension of Figure 12.



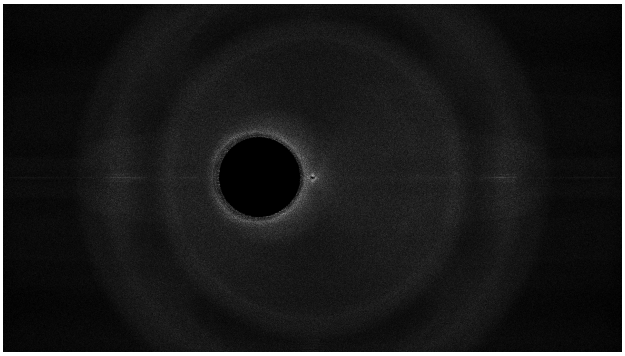
(e) Difference with Threshold of 4.64E-2



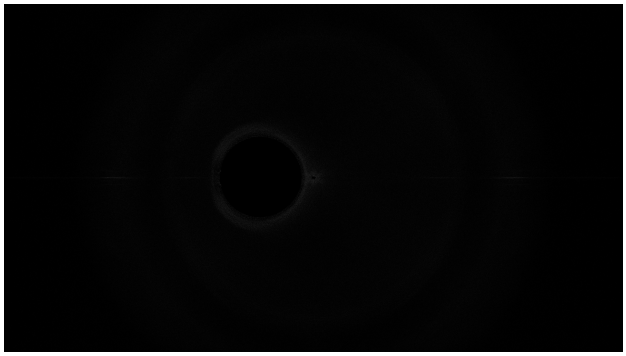
(f) Scaled Difference with Threshold of 4.64E-2



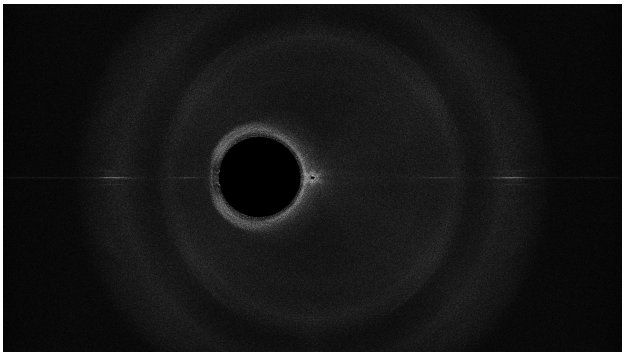
(g) Difference with Threshold of 1E-1



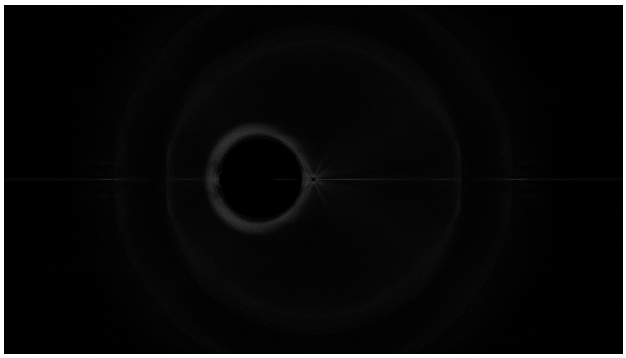
(h) Scaled Difference with Threshold of 1E-1



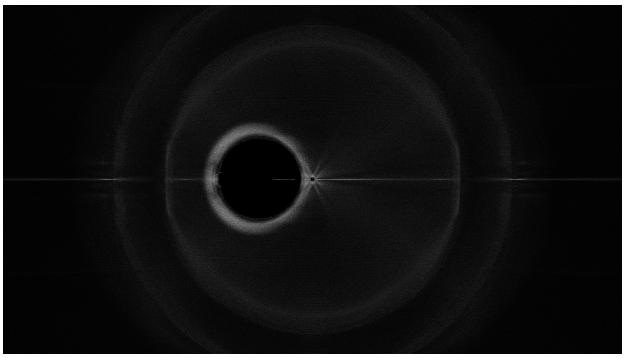
(i) Difference with Threshold of 2.15E-1



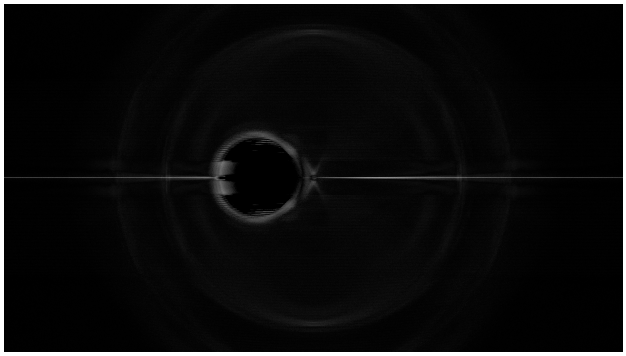
(j) Scaled Difference with Threshold of 2.15E-1



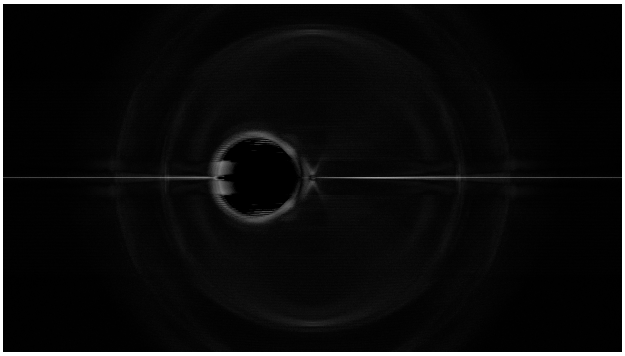
(k) Difference with Threshold of 4.64E-1



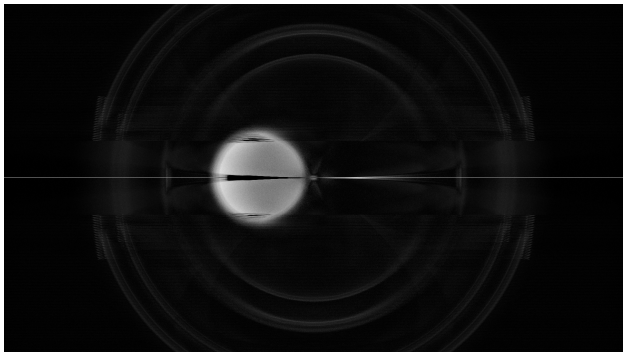
(l) Scaled Difference with Threshold of 4.64E-1



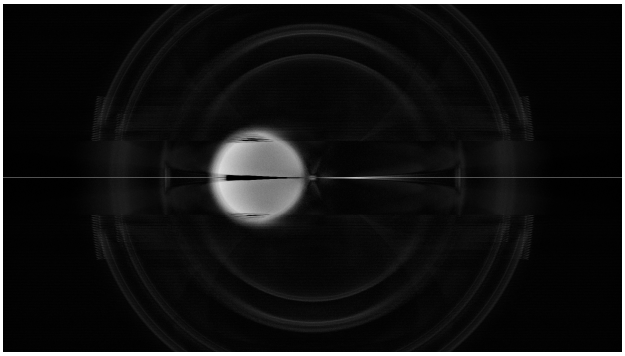
(m) Difference with Threshold of 1



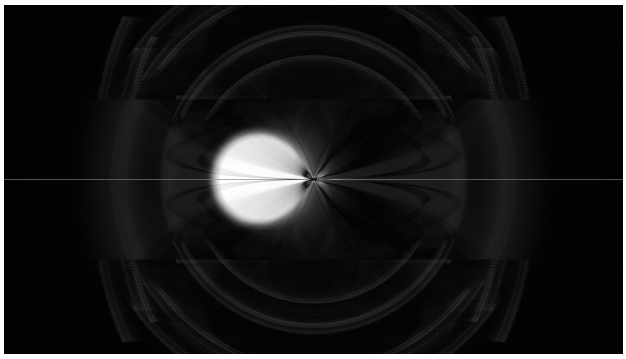
(n) Scaled Difference with Threshold of 1



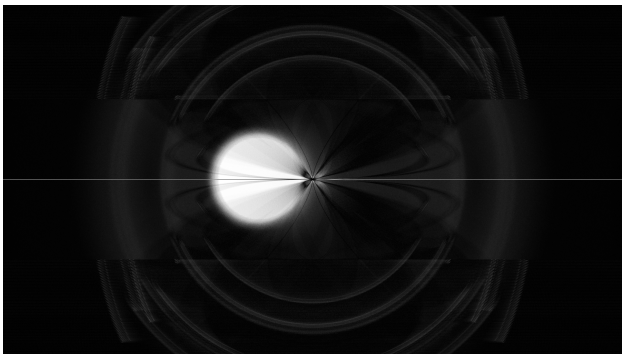
(o) Difference with Threshold of 2.15



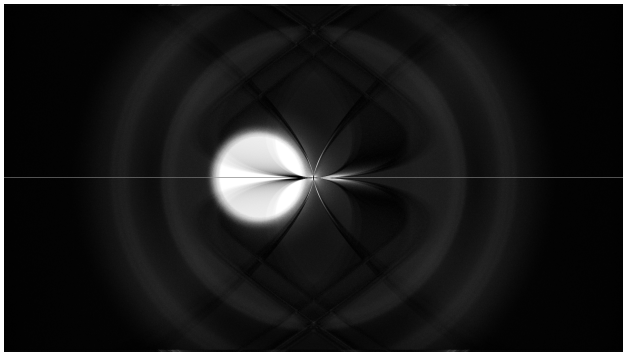
(p) Scaled Difference with Threshold of 2.15



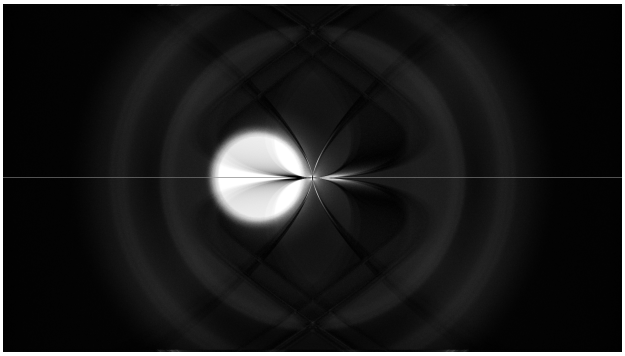
(q) Difference with Threshold of 4.64



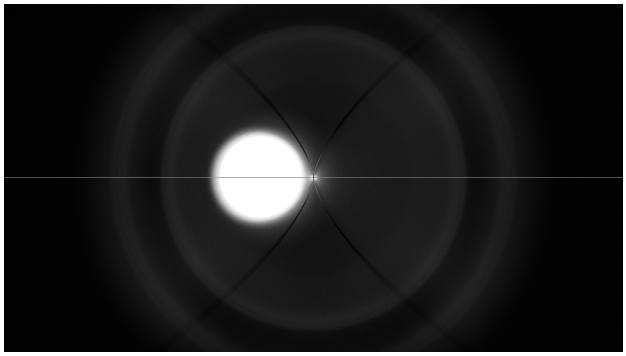
(r) Scaled Difference with Threshold of 4.64



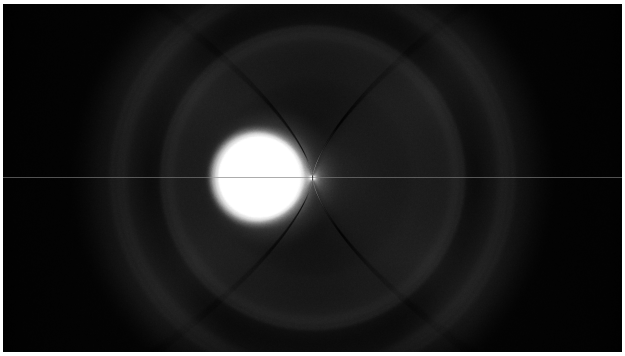
(s) Difference with Threshold of 1E1



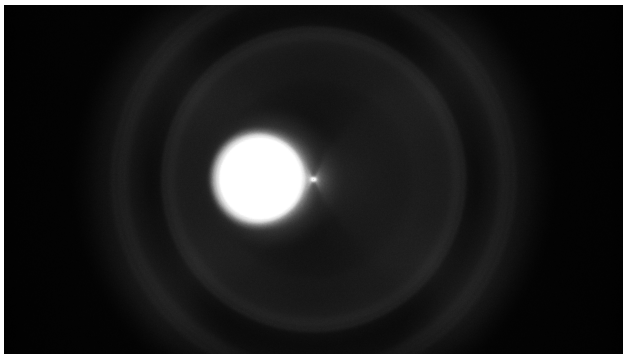
(t) Scaled Difference with Threshold of 1E1



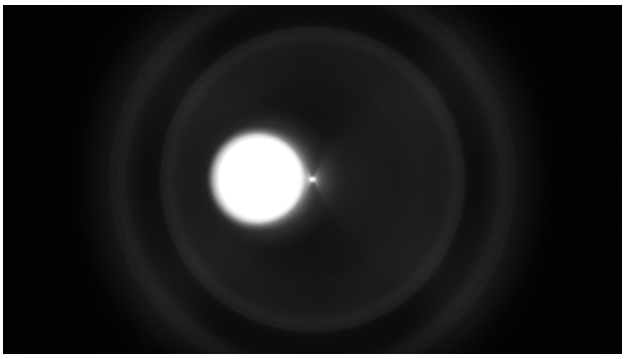
(u) Difference with Threshold of 2.15E1



(v) Scaled Difference with Threshold of 2.15E1



(w) Difference with Threshold of 4.64E1 and 1E2



(x) Scaled Difference with Threshold of 4.64E1 and 1E2

D C++ Code For Generating Lens Flares

Below is the code used for generating lens flares for this project. This calls from it several outside libraries, not including in this document, that are publicly available to download online. This file contains within it all of the implemented algorithms mentioned that are not specific to manipulating the polynomials themselves. This code is adapted from the example provided in [5].

```
/* Polynomial Optics
 * (C) 2012, Matthias Hullin <hullin@cs.ubc.ca>
 * (University of British Columbia)
 *
 * This file is in the public domain.
 * Different licenses may apply for the included libraries.
 *
 * Edits made by Stephen DiIorio
 */

#include <TruncPolySystem.hh>

#include <OpticalMaterial.hh>
#include <Spherical5.hh>
#include <Cylindrical5.hh>
#include <Propagation5.hh>
#include <TwoPlane5.hh>

#include <FindFocus.hh>

#include <iostream>
#include <stdlib.h>

#define cimg_display 0

#include <CImg.h>
#include <spectrum.h>
#include <levmar.h>

#include <vector>
#include <algorithm>

using namespace cimg_library;

// Global variables specific to the polynomial we are dealing with
const int DEGREE = 3;
const int NUMVARS = 4;

// Properties of the lens system we are approximating
OpticalMaterial glass1("N-SSK8", true);
OpticalMaterial glass2("N-SF10", true);
// Also try: const float d0 = 5000; // Scene is 5m away
const float d0 = 5000000; // Scene is 5km away
const float R1 = 65.22;
```

```
const float d1 = 9.60;
const float R2 = -62.03;
const float d2 = 4.20;
const float R3 = -1240.67;


/**
 * Computes the factorial of the given number.
 * @param  n an integer >= 1 to compute the factorial of
 * @return   the factorial of the provided number, n
 */
inline int factorial(int n) {
  if (n == 1) {
    return n;
  } else {
    return n * factorial(n - 1);
  }
}

/**
 * Computes the combination of "p choose k." Not very precise for large numbers, but it gets
      the job done.
 * @param  p the number you choose from
 * @param  k the number you "choose"
 * @return   the combination of p choose k
 */
inline int combination(int p, int k) {
  if (k > p) {
    return 0;
  }
  return factorial(p) / (factorial(k) * factorial(p-k));
}

/**
 * Takes the elements from vector2 and appends them, in order, to the end of vector1, and
     returns the resulting combination.
 * @param  vector1 first vector to take elements from
 * @param  vector2 second vector to take elements from
 * @return         a vector with the elements from vector1 and vector2
 */
template<typename T>
vector<T> append(vector<T> vector1, vector<T> vector2) {
  vector<T> toReturn;
  toReturn.reserve(vector1.size() + vector2.size());
  toReturn.insert(toReturn.end(), vector1.begin(), vector1.end());
  toReturn.insert(toReturn.end(), vector2.begin(), vector2.end());

  return toReturn;
}

/**
 * Given the number of variables in an polynomial system, and the desired degree of the
     polynomial system, produces all possible ways to exponentiate those variables such that
      they are within the specified degree.
 * @param  degree  the degree of the polynomial
 * @param  numVars the number of variables that the polynomial consists of
 * @return         A vector containing all the possible ways to exponentiate the variables
```

```
    of
 *                  the polynomial. This vector contains other vectors that list the exponent
 *                  of the variables in sequence. So, a vector containing <1,2> would
 *                  correspond to a polynomial of two variables and mean to raise the first
 *                  variable to the power of 1 and the second to the power of 2.
 */
vector<vector<int> > generateExponents(int degree, int numVars) {
  // The storage for this algorithm is as follows. The highest level vector holds all of the
  // possible exponentiations we have. The vectors within this higher container represent,
  // for a given degree, the possible combinations of exponents equaling that degree. So,
      the
  // first vector within that larger vector refers to the combinations of exponents
      resulting
  // in a total degree of 0, the second element refers to all the combinations of exponents
  // responding to a degree of 1, and so on until we reach the max degree. The third layer
      of
  // vectors just contain the numbers representing the exponents in them.

  // This represents the current column containing all of the possible exponent combinations
  // we have for the current number of variables. We first start filling this up as if we
      had
  // one variable, then keep updating its contents for each incremental variable we compute.
  vector<vector<vector<int> > > columnExp;

  if (numVars > 0) {
    // Begin by initializing the column with the base case. In the base case, when we have 1
    // variable, the possible values are just exponentiating that variable by every number
        up
    // to the desired degree.
    for (int i = 0; i <= degree; ++i) {
      vector<vector<int> > exponents;
      vector<int> v;
      v.push_back(i);
      exponents.push_back(v);
      columnExp.push_back(exponents);
    }

    // By the end of the initialization, the first column of our table (the vector) looks
    // like: < <<0>>, <<1>>, <<2>>, . . ., <<degree>> >
    // If we only have one variable in our polynomial, this is processed and returned,
    // otherwise we continue.

    // Continue editing contents until we reach the desired number of variables.
    for (int curVar = 2; curVar <= numVars; ++curVar) {
      vector<vector<vector<int> > > tmpExps; // Store the new exponents
      for (int d = 0; d <= degree; ++d) {
        vector<vector<int> > newExponents;
        for (int dprime = 0; dprime <= d; ++dprime) {
          for (vector<vector<int> >::iterator curSeq = columnExp.at(dprime).begin();
               curSeq != columnExp.at(dprime).end();
               ++curSeq) {
            vector<int> newSeries((*curSeq));
            newSeries.push_back(d-dprime); // Append the next element to the sequence
            newExponents.push_back(newSeries);
          }
        }
        tmpExps.push_back(newExponents);
```

```
      }
      columnExp = tmpExps;
    }
  }

  // Massage the data into something we can return.
  vector<vector<int> > toReturn;

  for (vector<vector<vector<int> > >::iterator it = columnExp.begin(); it != columnExp.end()
      ; ++it) {
    for (vector<vector<int> >::iterator it2 = (*it).begin(); it2 != (*it).end(); ++it2) {
      toReturn.push_back(*it2);
    }
  }

  return toReturn;
}

/**
 * Given a number of interfaces in a lens system, the number of reflections allowed (must be
     even) and the starting interface to build to paths from, generates all of the
    interfaces we can reflect off of to form a legal path.
 * @param  numOfInterfaces   the number of interfaces in the lens system
 * @param  numOfReflections  the number of reflections allowed for each path
 * @param  startingInterface the interface to start generating paths from
 * @return                   the set of all possible interfaces we can reflect off of
 */
vector<vector<int> > generateReflections(int numOfInterfaces, int numOfReflections,
                                         int startingInterface) {
  vector<vector<int> > reflections;
  if (numOfReflections > 0) {
    for (int r1 = startingInterface + 1; r1 < numOfInterfaces; r1++) {
      for (int r2 = 0; r2 < r1; r2++) {
        vector<int> newReflection;
        newReflection.push_back(r1);
        newReflection.push_back(r2);

        vector<vector<int> > prevRefs = generateReflections(numOfInterfaces,
                                                            numOfReflections - 2, r2);
        if (!prevRefs.empty()) {
          for (vector<vector<int> >::iterator it = prevRefs.begin(); it != prevRefs.end();
               it++) {
            reflections.push_back(append(newReflection, *it));
          }
        } else {
          reflections.push_back(newReflection);
        }
      }
    }
  }

  return reflections;
}

/**
 * Given a set of interfaces we reflect upon, generates all possible paths we can take
     through the lens system including those reflections.
```

```
 * @param  reflections     the set of lists containing the reflections in the path in order
 * @param  degree          the degree of the polynomial
 * @param  lambda          the wavelength of light we are approximating the system with
 * @param  numOfInterfaces total number of interfaces
 * @return                 the set of lists of paths possible with the given reflections
 */
vector<Transform4f> generatePath(vector<vector<int> > reflections, int degree, float lambda,
                                 int numOfInterfaces) {

  float radii[] = {R1, R2, R3};
  float distances[] = {d1, d2};
  float indices[] = {1.f, glass1.get_index(lambda), glass2.get_index(lambda), 1.f};

  vector<Transform4f> paths;

  for (vector<vector<int> >::iterator it = reflections.begin(); it != reflections.end(); ++
      it) {
    Transform4f lensSystem = two_plane_5(d0, degree);
    int currentInterface = 0;
    for (vector<int>::iterator it2 = (*it).begin(); it2 != (*it).end(); it2 += 2) {
      while (currentInterface < *it2) {
        lensSystem = lensSystem >> refract_spherical_5(radii[currentInterface],
                                                       indices[currentInterface],
                                                       indices[currentInterface + 1],
                                                       degree)
                                >> propagate_5(distances[currentInterface], degree);
        currentInterface++;
      }
      lensSystem = lensSystem >> reflect_spherical_5(radii[currentInterface], degree);
      while (currentInterface > *(it2 + 1)) {
        if (currentInterface - 1 == *(it2 + 1)) {
          lensSystem = lensSystem >> propagate_5(distances[currentInterface - 1], degree);
        } else {
          lensSystem = lensSystem >> propagate_5(distances[currentInterface - 1], degree)
                                  >> refract_spherical_5(radii[currentInterface - 1],
                                                         indices[currentInterface],
                                                         indices[currentInterface - 1],
                                                             degree);
        }
        currentInterface--;
      }
    }
    lensSystem = lensSystem >> reflect_spherical_5(radii[currentInterface], degree);
    while (currentInterface < numOfInterfaces - 1) {
      lensSystem = lensSystem >> propagate_5(distances[currentInterface], degree)
                              >> refract_spherical_5(radii[currentInterface + 1],
                                                     indices[currentInterface + 1],
                                                     indices[currentInterface + 2], degree);
      currentInterface++;
    }
    paths.push_back(lensSystem);
  }

  return paths;
}

/**
```

```
 * Given a set of interfaces we reflect upon describing a path through the lens system and
     some initial ray information, creates a "ground truth" measurement and produces results
      more accurate than just evaluating an approximated, combined lens system.
 * @param  reflections     the set of lists containing the reflections in the path in order
 * @param  ray             the initial conditions of the incoming ray incident on the lens
 *                         system (x, y, dx, dy)
 * @param  degree          the degree of the polynomial
 * @param  lambda          the wavelength of light we are approximating the system with
 * @param  numOfInterfaces total number of interfaces
 * @return                 the set of lists of paths possible with the given reflections
 */
vector<float> generateTruePath(vector<int> reflections, float ray[4], int degree,
                               float lambda, int numOfInterfaces) {

  float radii[] = {R1, R2, R3};
  float distances[] = {d1, d2};
  float indices[] = {1.f, glass1.get_index(lambda), glass2.get_index(lambda), 1.f};

  float ray1[NUMVARS];
  float ray2[NUMVARS];

  vector<float> outputs;

  Transform4f lensSystem = two_plane_5(d0, degree);
  lensSystem.evaluate(ray,ray1,degree);
  int currentInterface = 0;
  for (vector<int>::iterator it = reflections.begin(); it != reflections.end(); it += 2) {
    while (currentInterface < *it) {
      lensSystem = refract_spherical_5(radii[currentInterface], indices[currentInterface],
                                       indices[currentInterface + 1], degree);
      lensSystem.evaluate(ray1,ray2,degree);
      lensSystem = propagate_5(distances[currentInterface], degree);
      lensSystem.evaluate(ray2,ray1,degree);

      currentInterface++;
    }
    lensSystem = reflect_spherical_5(radii[currentInterface], degree);
    lensSystem.evaluate(ray1,ray2,degree);
    while (currentInterface > *(it + 1)) {
      if (currentInterface - 1 == *(it + 1)) {
        lensSystem = propagate_5(distances[currentInterface - 1], degree);
        // This is safe because we never, in any iteration of the loop, go into the else
        // statement after entering the if portion. This is good, otherwise we would have
        // to worry about clobbering values stored in the different rays.
        lensSystem.evaluate(ray2,ray1,degree);
      } else {
        lensSystem = propagate_5(distances[currentInterface - 1], degree);
        lensSystem.evaluate(ray2,ray1,degree);
        lensSystem = refract_spherical_5(radii[currentInterface - 1],
                                         indices[currentInterface],
                                         indices[currentInterface - 1], degree);
        lensSystem.evaluate(ray1,ray2,degree);
      }
      currentInterface--;
    }
  }
  lensSystem = reflect_spherical_5(radii[currentInterface], degree);
```

```
  // This is safe because we always execute the if portion last in the previous loop.
  lensSystem.evaluate(ray1,ray2,degree);
  while (currentInterface < numOfInterfaces - 1) {
    lensSystem = propagate_5(distances[currentInterface], degree);
    lensSystem.evaluate(ray2,ray1,degree);
    lensSystem = refract_spherical_5(radii[currentInterface + 1],
                                     indices[currentInterface + 1],
                                     indices[currentInterface + 2], degree);
    lensSystem.evaluate(ray1,ray2,degree);
    currentInterface++;
  }
  outputs.assign(ray2,ray2+NUMVARS);

  return outputs;
}

/**
 * Given a series of input parameters, computes, for a given path through the lens system,
     the resulting series of ground truth output values.
 * @param  reflections     the set of lists containing the reflections in the path in order
 * @param  inputData       the set of input values or "rays" to evaluate the system with
 * @param  numValues       the number of input values (should be a multiple of 4)
 * @param  degree          the degree of the polynomial
 * @param  lambda          the wavelength of light we are approximating the system with
 * @param  numOfInterfaces the number of interfaces in the lens system
 * @return                 a container with all of the "true" values for the lens system
 */
vector<float> collectTrueValues(vector<int> reflections, float *inputData, int numValues,
                                int degree, float lambda, int numOfInterfaces) {
  vector<float> toReturn;

  for (int i = 0; i < numValues; i += NUMVARS) {
    float currentPoint[] = {inputData[i], inputData[i+1], inputData[i+2], inputData[i+3]};
    vector<float> result = generateTruePath(reflections, currentPoint, degree, lambda,
        numOfInterfaces);
    toReturn = append(toReturn, result);
  }

  return toReturn;
}

/**
 * Computes exponentiation, base^exponent, in an efficient and quick manner.
 * @param  base     the base of the exponent
 * @param  exponent the power the base is being raised to
 * @return          the result of base^exponent
 */
inline float newPow(float base, int exponent) {
  float result = 1;
  int exp_ = exponent;
  float base_ = base;

  while (exp_) {
    if (exp_ & 1) {
      result *= base_;
    }
    exp_ >>= 1;
```

```
    base_ *= base_;
  }

  return result;
}

/**
 * Given the values of variables, and the power to which we should raise them, computes the
     sum of raising those variables to their associated powers.
 * @param  exp    the set containing which power to raise which exponents to
 * @param  values the initial values of the variables we exponentiate
 * @return        the sum of all the initial values raised to their respective exponents
 */
inline float exponentiateVariables(vector<int> exps, float values[NUMVARS]) {
  float toReturn = 0;
  int index = 0;
  for (vector<int>::iterator it = exps.begin(); it != exps.end(); ++it) {
    toReturn += newPow(values[index], *it);
    index++;
  }
  return toReturn;
}

/**
 * Creates the results of evaluating the provided polynomial. Some parameters are multiplied
      by 4 because the polynomials we are dealing with have in them 4 equations.
 * @param p    The coefficients of all the terms in the given polynomial
 * @param x    The array we fill with the results of computing each polynomial
 * @param m    4 * the max number of coefficients we can have in a polynomial given its
     degree
 * @param n    the number of points we sample
 * @param data Contains the input parameters (rays) for each data point (x, y, dx, dy values
     )
 */
void polyFunc(float *p, float *x, int m, int n, void *data) {
  int dataIndex = 0;
  vector<vector<int> > exponents = generateExponents(DEGREE, NUMVARS);

  // go over the rows of the output array (x)
  for (int xrow = 0; xrow < (n / NUMVARS); ++xrow) {
    float inputValues[NUMVARS]; // the array to store the current input values
    // collect the input values from the data array
    for (int count = 0; count < NUMVARS; ++count) {
      inputValues[count] = ((float *)data)[dataIndex++];
    }

    // go over the columns of the output array (x) and is used to go over the rows of p
    for (int xcolprow = 0; xcolprow < NUMVARS; ++xcolprow) {
      // the current output value is the result of evaluating the appropriate polynomial
      // with the current input data
      float outputValue = 0.0;
      // go over the columns of the parameter array (p)
      for (int pcol = 0; pcol < (m / NUMVARS); ++pcol) {
        outputValue += p[(m/NUMVARS)*xcolprow + pcol] * exponentiateVariables(exponents[pcol
            ], inputValues);
      }
      x[NUMVARS*xrow + xcolprow] = outputValue;
```

```
    }
  }
}

/**
 * Creates the jacobian for the supplied polynomial following the levmar specification.
 * @param p    The coefficients of all the terms in the given polynomials
 * @param jac  The array to store the jacobian
 * @param m    4 * the max number of coefficients we can have in a polynomial given its
     degree
 * @param n    the number of points we sample
 * @param data Contains the input parameters (rays) for each data point (x, y, dx, dy values
     )
 */
void polyJacobFunc(float *p, float *jac, int m, int n, void *data) {
  int dataIndex = 0;
  vector<vector<int> > exponents = generateExponents(DEGREE, NUMVARS);

  // go over the rows of the jac matrix
  for (int row = 0; row < n; row += NUMVARS) {
    float inputValues[NUMVARS]; // store the current parameters
    // collect the input values from the data array
    for (int count = 0; count < NUMVARS; ++count) {
      inputValues[count] = ((float *)data)[dataIndex++];
    }

    int curIndex = 0;
    // repeat this for the four different rows in the jac
    for (int curEq = 0; curEq < NUMVARS; ++curEq) {
      // set all the values in the current row to 0
      for (int col = 0; col < m; ++col) {
        jac[m*(row+curEq) + col] = 0.0;
      }
      // only fill in the current values we want to
      for (int pos = 0; pos < (m/NUMVARS); ++pos) {
        // the current output is the partial derivative of the coefficient, which is just
            the
        // value of the coefficients they multiply the value is zero if the coefficient is
        // zero, though
        float outputValue = 0.0;
        if (p[curIndex] != 0) {
          outputValue = exponentiateVariables(exponents[pos], inputValues);
        }
        jac[m*(row+curEq) + curIndex] = outputValue;
        ++curIndex;
      }
    }
  }
}

/**
 * Generates a series of data points corresponding to the x, y position from the scene and
     the x, y position on the aperture that are used as input parameters to evaluate a
     polynomial.
 * @param data          the array to store the data points generated must be of size
 *                      4 * width * height
 * @param w             width of image in pixels
```

```
 * @param h             height of image in pixels
 * @param r_pupil       the radius of the pupil of the aperture
 * @param sensor_width  the width of the sensor
 * @param magnification the magnification of the system
 */
void generateSamplePoints(float *data, int w, int h, float r_pupil, float sensor_width,
    float magnification) {
  int currentIndex = 0;
  float x_ap, y_ap;
  for (int i = 0; i < h; ++i) {
    float y_sensor = ((i - h/2)/(float)w) * sensor_width;
    float y_world = y_sensor / magnification;
    for (int j = 0; j < w; ++j) {
      float x_sensor = (j / (float)w - 0.5) * sensor_width;
      float x_world = x_sensor / magnification;
      do {
        x_ap = (rand() / (float)RAND_MAX - 0.5) * 2 * r_pupil;
        y_ap = (rand() / (float)RAND_MAX - 0.5) * 2 * r_pupil;
      } while (x_ap * x_ap + y_ap * y_ap > r_pupil * r_pupil);
      data[currentIndex++] = x_world; // xpos
      data[currentIndex++] = y_world; // ypos
      data[currentIndex++] = x_ap; // xpos on aperture
      data[currentIndex++] = y_ap; // ypos on aperture
    }
  }
}

/**
 * Generates a new series of polynomials, encapsulating them in a TruncPolySystem, given the
     set of coefficients for the terms in the polynomial.
 * @param  coeff the set of the coefficients for the new polynomial
 * @param  trunc the degree to truncate the polynomials to
 * @return       the set of polynomials described by the provided coefficients
 */
Transform4f makeSystem(vector<float> coeff, int trunc = 5) {
  Transform4f pt;
  pt.trunc_degree = trunc;
  int term = combination(DEGREE+NUMVARS, NUMVARS);
  vector<vector<int> > exponents = generateExponents(DEGREE, NUMVARS);
  int index = 0;
  int numTerms[4] = {0, 0, 0, 0};
  int eqIndex = 0;

  PT4fData *element[4];
  for (int i = 0; i < 4; ++i) {
    element[i] = new PT4fData[1024];
  }
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < term; ++j) {
      if (coeff[index] != 0) {
        element[i][eqIndex++] = PT4fData(coeff[index],exponents.at(j).at(0),
                                                      exponents.at(j).at(1),
                                                      exponents.at(j).at(2),
                                                      exponents.at(j).at(3));
        numTerms[i] += 1;
      }
      index++;
```

```
    }
    eqIndex = 0;
  }

  for (int i = 0; i < 4; ++i) {
    Poly4f poly(numTerms[i],element[i],sizeof(PT4fData));
    pt[i] = poly;
    pt[i].trunc_degree = trunc;
  }
  pt.consolidate_terms();

  for (int i = 0; i < 4; ++i) {
    delete[] element[i];
  }
  return pt;
}

int main(int argc, char *argv[]) {
  // sleep(15);

  // int degree = 3;
  // if (argc >= 2) {degree = atol(argv[1]);}

  float sample_mul = 1000;
  if (argc >= 3) sample_mul = atof(argv[2]);
  cout << "sample_mul: "<<sample_mul<<endl;

  float r_entrance = 19.5;
  if (argc >= 4) r_entrance = atof(argv[3]);

  int num_lambdas = 12;
  if (argc >= 5) num_lambdas = atol(argv[4]);

  int filter_size = 1;
  if (argc >= 6) filter_size = atol(argv[5]);

  int frame_from = 0;
  if (argc >= 7) frame_from = atol(argv[6]);

  int frame_to = 199;
  if (argc >= 8) frame_to = atol(argv[7]);

  // Paper figure 13: frame 26 and 299. Homepage also shows frame 216.
  //int frame_list[3] = {26, 216, 299};
  int frame_list[1] = {26};
  int num_frames = 1;

  // Sensor scaling
  const float sensor_width = 36;
  const int sensor_xres = 1920;
  const int sensor_yres = 1080;
  const float sensor_scaling = sensor_xres / sensor_width;
  cout << "Sensor scaling: "<< sensor_scaling << endl;

  const float lambda_from = 440;
  const float lambda_to = 660;
```

```
int beforeLambdaTermNum = 0;
int afterLambdaTermNum = 0;
int beforeYTerm = 0;
int afterYTermNum = 0;
int afterDropTermNum = 0;
int lambdaCounter = 0;
int yCounter = 0;

CImg<float> img_in("image.pfm");
int width = img_in.width();
int height = img_in.height();

vector<vector<int> > reflections2 = generateReflections(3, 2, 0);
vector<vector<int> > reflections4 = generateReflections(3, 4, 0);

for (int frame_idx = 0; frame_idx < num_frames; ++frame_idx) {
  int frame = frame_list[frame_idx];

  cout << endl << endl << "[[ FRAME "<<frame<<" ]]"<<endl<<"=============="<<endl;
  float r_pupil = r_entrance;
  if (frame < 100) {r_pupil = 0.1*sqrt(1.f+frame) * r_entrance;}
  cout << "Pupil radius: "<<r_pupil<<endl;

  CImg<float> img_out(sensor_xres, sensor_yres, 1, 3, 0);

  vector<Transform4f> ref2Path550 = generatePath(reflections2, DEGREE, 550, 3);
  vector<Transform4f> ref4Path550 = generatePath(reflections4, DEGREE, 550, 3);
  vector<Transform4f> ref2Path500 = generatePath(reflections2, DEGREE, 500, 3);
  vector<Transform4f> ref4Path500 = generatePath(reflections4, DEGREE, 500, 3);
  vector<Transform4f> ref2Path600 = generatePath(reflections2, DEGREE, 600, 3);
  vector<Transform4f> ref4Path600 = generatePath(reflections4, DEGREE, 600, 3);

  vector<Transform4f> paths[] = {ref2Path550, ref2Path500, ref2Path600, ref4Path550,
      ref4Path500, ref4Path600};

  for (int reflPath = 0; reflPath < 6; reflPath += 3) {
    vector<Transform4f> flares550 = paths[0+reflPath];
    vector<Transform4f> flares500 = paths[1+reflPath];
    vector<Transform4f> flares600 = paths[2+reflPath];
    for (int refl = 0; refl < flares550.size(); refl++) {
      // Focus on 550nm
      Transform4f system1 = flares550[refl];

      // Determine back focal length from degree-1 terms (matrix optics)
      float d3 = find_focus_X(system1);
      cout << "Focus: " << d3 << endl;
      // Compute magnification and output equation system
      float magnification = get_magnification_X(system1 >> propagate_5(d3));
      cout << "Magnification: " << magnification << endl;

      vector<float> coeff = system1.collectCoeff(generateExponents(DEGREE, NUMVARS));
      float *coeffArr = (float *)malloc(coeff.size()*sizeof(float));

      for (int copyID = 0; copyID < coeff.size(); ++copyID) {
        coeffArr[copyID] = coeff[copyID];
      }
```

```
float *data = (float *)malloc(NUMVARS*width*height*sizeof(float));
generateSamplePoints(data, width, height, r_pupil, sensor_width, magnification);

vector<float> trueValues;

if (reflPath < 3) {
  trueValues = collectTrueValues(reflections2[refl], data, NUMVARS*width*height,
      DEGREE, 550, 3);
} else {
  trueValues = collectTrueValues(reflections4[refl], data, NUMVARS*width*height,
      DEGREE, 550, 3);
}

float *valueArr = (float *)malloc(trueValues.size()*sizeof(float));
for (int copyID = 0; copyID < trueValues.size(); ++copyID) {
  valueArr[copyID] = trueValues[copyID];
}

float info[LM_INFO_SZ];
float opts[LM_OPTS_SZ];
opts[0]=LM_INIT_MU; opts[1]=1E-15; opts[2]=1E-15; opts[3]=1E-20;
opts[4]=LM_DIFF_DELTA;

slevmar_der(polyFunc, // the function that generates the polynomials
            polyJacobFunc, // the function that generates the jacobian
            coeffArr, // the array that stores the coefficients of the polynomials
            valueArr, // the real values determined by the ground truth
            NUMVARS*combination(DEGREE+NUMVARS, NUMVARS), // m, the number of
                unknowns
            NUMVARS*width*height, // n, the number of measurements we took
            10000, // number of trials to do
            opts, // options
            info, // output information
            NULL,
            NULL,
            data); // the points we use initially to evaluate the system

vector<float> newCoeffV;
newCoeffV.assign(coeffArr,coeffArr+coeff.size());

system1 = makeSystem(newCoeffV, DEGREE);

// Add that propagation, plus a little animated defocus to the overall system;
Transform4f prop = propagate_5(d3 - ((frame>=100)?(0.02*(frame-100)):0), DEGREE);
system1 = system1 >> prop;

// Precompute spectrum
float rgb[3 * num_lambdas];
for (int ll = 0; ll < num_lambdas; ++ll) {
  float lambda = lambda_from + (lambda_to - lambda_from) * (ll / (float)(num_lambdas
      -1));
  if (num_lambdas == 1) {lambda = 550;}
  // Convert wavelength to spectral power
  spectrum_p_to_rgb(lambda, 1, rgb + 3 * ll);
}

// Sample optical system at two spectral locations
```

```
Transform4d system_spectral_center1 = flares500[refl] >> prop;
Transform4d system_spectral_right1 = flares600[refl] >> prop;

// Obtain (xyworld + xyaperture + lambda) -> (ray) mapping including chromatic
    effects,
// by linear interpolation of the two sample systems drawn above
System54f system_spectral = system_spectral_center1.lerp_with(system_spectral_right1
    , 550, 600);

// dx and dy after propagation are really only needed for Lambertian
// term; hence: combine them to obtain sin^2 = 1 - cosine^2 term in equation 2:
system_spectral[2] = (system_spectral[2] * system_spectral[2] + system_spectral[3] *
     system_spectral[3]);
system_spectral[2] %= 2;
System53d system_lambert_cos2 = system_spectral.drop_equation(3);

// Support of an input image pixel in world plane
float pixel_size = sensor_width/(float)width/magnification;

// Calculate max possible values for input variables when removing terms smartly
// maxValues[0] = xMax
// maxValues[1] = dxMax
// maxValues[2] = dyMax
float maxValues[5];
maxValues[0] = ((0.5 * sensor_width) / magnification) + pixel_size * 0.5;
maxValues[1] = r_pupil;
maxValues[2] = r_pupil;

for (int ll = 0; ll < num_lambdas; ++ll) {
  float lambda = lambda_from + (lambda_to - lambda_from) * (ll / (float)(num_lambdas
      -1));
  if (num_lambdas == 1) {lambda = 550;}
  cout << "["<<lambda<<"nm]"<<flush;

  beforeLambdaTermNum += system_lambert_cos2.count_terms();

  // Bake lambda dependency
  System43f system_lambda = system_lambert_cos2.bake_input_variable(4, lambda);
  system_lambda %= DEGREE;

  afterLambdaTermNum += system_lambda.count_terms();

  ++lambdaCounter;

  // Bake lambda into derivatives as well:

  for (int j = 0; j < height; j++) {
    if (!(j%10)) {cout << "." << flush;}
    const float y_sensor = ((j - height/2)/(float)width) * sensor_width;
    const float y_world = y_sensor / magnification;

    beforeYTerm += system_lambda.count_terms();

    // Bake y dependency
    System33f system_y = system_lambda.bake_input_variable(1, y_world);

    afterYTermNum += system_y.count_terms();
```

```
745
            // Remove terms based on input values
            // Uncomment this line and set a threshold
            // system_y.smart_drop_terms(1E2, maxValues);

750         afterDropTermNum += system_y.count_terms();

            ++yCounter;

            for (int i = 0; i < width; i++) {
755           const float x_sensor = (i / (float)width - 0.5) * sensor_width;
              const float x_world = x_sensor / magnification;

              // Sample intensity at wavelength lambda from source image
              const float rgbin[3] = {
760           img_in.linear_atXY(i, j, 0, 0, 0),
              img_in.linear_atXY(i, j, 0, 1, 0),
              img_in.linear_atXY(i, j, 0, 2, 0)};
              float L_in = spectrum_rgb_to_p(lambda, rgbin);

765           // Quasi-importance sampling:
              // pick number of samples according to pixel intensity
              int num_samples = max(1,(int)(L_in * sample_mul));
              float sample_weight = L_in / num_samples;

770           // With that, we can now start sampling the aperture:
              for (int sample = 0; sample < num_samples; ++sample) {

                // Rejection-sample points from lens aperture:
                float x_ap, y_ap;
775             do {
                  x_ap = (rand() / (float)RAND_MAX - 0.5) * 2 * r_pupil;
                  y_ap = (rand() / (float)RAND_MAX - 0.5) * 2 * r_pupil;
                } while (x_ap * x_ap + y_ap * y_ap > r_pupil * r_pupil);

780             float in[5], out[4];

                // Fill in variables and evaluate systems:
                in[0] = x_world + pixel_size * (rand()/(float)RAND_MAX - 0.5);
                in[1] = x_ap;
785             in[2] = y_ap;

                // Uncomment one of these to evaluate the system with the described methods
                // system_y.evaluate(in, out);
                // system_y.memo_evaluate(in,out);
790             // system_y.memo_all_evaluate(in,out,DEGREE);

                // Scale to pixel size:
                out[0] = out[0] * sensor_scaling + sensor_xres/2;
                out[1] = out[1] * sensor_scaling + sensor_yres/2;
795
                // out[2] contains one minus square of Lambertian cosine
                float lambert = sqrt(1 - out[2]);
                if (lambert != lambert) {lambert = 0;} // NaN check

800             img_out.set_linear_atXY(lambert * sample_weight * rgb[0 + 3 * ll], out[0],
                    out[1],0,0, true);
```

```
                    img_out.set_linear_atXY(lambert * sample_weight * rgb[1 + 3 * ll], out[0],
                        out[1],0,1, true);
                    img_out.set_linear_atXY(lambert * sample_weight * rgb[2 + 3 * ll], out[0],
                        out[1],0,2, true);
                  }
                }
              }
            }
            free(coeffArr);
            free(valueArr);
            free(data);
          }
        }


        // Fix gamut problem (pure wavelengths sometimes result in negative RGB)
        for (int j = 0; j < sensor_yres; ++j) {
          for (int i = 0; i < sensor_xres; ++i) {
            float max_value =  max(img_out.atXY(i, j, 0, 0),
                                   max(img_out.atXY(i, j, 0, 1),
                                   img_out.atXY(i, j, 0, 2)));
            img_out.atXY(i,j,0,0) = max(img_out.atXY(i,j,0,0), 0.02f*max_value);
            img_out.atXY(i,j,0,1) = max(img_out.atXY(i,j,0,1), 0.02f*max_value);
            img_out.atXY(i,j,0,2) = max(img_out.atXY(i,j,0,2), 0.02f*max_value);
          }
        }

        char fn[256];
        sprintf(fn,"Output-frame%03d.pfm",frame);
        img_out.save(fn);

      }

      cout << endl;

      cout << "Average number of terms before lambda bake: " << (float)beforeLambdaTermNum/(
          float)lambdaCounter << endl;
      cout << "Average number of terms after lambda bake: " << (float)afterLambdaTermNum/(float)
          lambdaCounter << endl;
      cout << "Average number of terms before y bake: " << (float)beforeYTerm/(float)yCounter <<
           endl;
      cout << "Average number of terms after y bake and before removing terms: " << (float)
          afterYTermNum/(float)yCounter << endl;
      cout << "Average number of terms after removing terms: " << (float)afterDropTermNum/(float
          )yCounter << endl;
    }
```

E C++ Code for Manipulating Polynomials

Below is the code added to the Polynomial Optics library [5]. These particular functions were added to the *TruncPolySystem.hh* file. Details on the abbreviations used in this code and the workings of the rest of this

library can be found in their paper and the documentation of the code itself.

The following lines of code were added to the TruncPoly class, which deals with manipulating single polynomial equations.

```
/**
 * This takes in a threshold and removes all the terms of a polynomial whose coefficients
     are above said threshold. This alters the polynomial this function is called on.
 * @param  maxCoeff The threshold we set to determine if a term stays within the
 *                  polynomial or not
 */
_tSN
void TruncPoly _SN::drop_term(scalar maxCoeff) {

  vector<PolyTerm _SN > newTerms;

  for (unsigned int i = 0; i < terms.size(); ++i) {
    if (abs(terms[i].coefficient) >= maxCoeff) {
      newTerms.push_back(terms[i]);
    }
  }

  terms = newTerms;
}

/**
 * This takes in the values the variables within a polynomial have and evaluates all of the
     terms. As the terms are being evaluated, if they are above the threshold set, then they
      are removed from the polynomial. This alters the polynomial by removing all of the
     terms that, once evaluated, do not meet the set threshold.
 * @param  maxValue    The threshold we set to determine if a term stays within the
 *                     polynomial or not
 * @param  inputValues An array specifying the values the variables take on
 */
_tSN
void TruncPoly _SN::smart_drop_term(scalar maxValue, scalar inputValues[num_vars]) {

  vector<PolyTerm _SN > newTerms;

  for (int j = terms.size() - 1; j >= 0; --j) {
    PolyTerm<scalar,num_vars> *term = &terms[j];
    scalar term_value = term->coefficient;
    for (int k = 0; k < num_vars; ++k) {
      term_value *= intpow(inputValues[k],term->exponents[k]);
    }
    if (abs(term_value) >= maxValue) {
      newTerms.push_back(terms[j]);
    }
  }

  terms = newTerms;
}
```

The following lines of code were added to the TruncPolySystem class, which deals with systems of polynomial equations.

```
/**
 * This takes in a threshold and removes all the terms of all the polynomials in the system
     whose coefficients are above said threshold.
 * @param  maxCoeff The threshold we set to determine if a term stays within the
 *                  polynomial or not
 */
_tSIO
void TruncPolySystem _SIO::drop_terms(scalar maxCoeff) {
  for (int i = 0; i < num_vars_out; ++i) {
    equations[i].drop_term(maxCoeff);
  }
}

/**
 * This takes in the values the variables within a polynomial have and evaluates all of the
     terms of every polynomial within the system. As the terms are being evaluated, if they
     are above the threshold set, then they are removed from the polynomial.
 * @param  maxValue    The threshold we set to determine if a term stays within the
 *                     polynomial or not
 * @param  inputValues An array specifying the values the variables take on
 */
_tSIO
void TruncPolySystem _SIO::smart_drop_terms(scalar maxValue, scalar inputValues[num_vars_in
    ]) {
  for (int i = 0; i < num_vars_out; ++i) {
    equations[i].smart_drop_term(maxValue, inputValues);
  }
}

/**
 * Evaluates a system of polynomials given the input variables' values and stores the output
      values in an array. This evaluates polynomials with memoization.
 * @param  x0    The array to store the values of the variables within the polynomials
 * @param  x1    The array to store the evaluations of the polynomials
 */
_tSIO
void TruncPolySystem _SIO::memo_evaluate(const scalar x0[num_vars_in], scalar x1[
    num_vars_out], bool print){

  // Don't worry about wasting some space, the array is small enough that it should not make
  // a huge deal. This is a better saving then having to evaluate a multiplication

  // 59 is the max number of different terms this library allows (6 input variables and deg
      5)
  // and 59 is the max index I allow with the key generation for this set of conditions
  scalar calculatedTerms[59] = {};

  for (int i = 0; i < num_vars_out; ++i) {
    scalar result = 0;
    for (int j = (int)equations[i].terms.size() - 1; j >= 0; --j) {
      PolyTerm<scalar,num_vars_in> *term = &equations[i].terms[j];
```

```
45        scalar term_value = term->coefficient;
          for (int k = 0; k < num_vars_in; ++k) {
            echar exp_ = term->exponents[k];
            int key_ = (k << 3) + k + (int)exp_; // Unique key number for all possible values
            if (calculatedTerms[key_]) {
50            term_value *= calculatedTerms[key_];
            } else {
              scalar result_ = intpow(x0[k],exp_);
              term_value *= result_;
              calculatedTerms[key_] = result_;
55          }
          }

          result += term_value;
        }
60
        x1[i] = result;
      }
    }

65  /**
     * Evaluates a system of polynomials given the input variables' values and stores the output
          values in an array. This evaluates polynomials with the more efficient manner of
         memoization.
     * @param  x0    The array to store the values of the variables within the polynomials
     * @param  x1    The array to store the evaluations of the polynomials
     * @param  deg   The degree of the polynomials we are working with
70   */
    _tSIO
    void TruncPolySystem _SIO::memo_all_evaluate(const scalar x0[num_vars_in], scalar x1[
        num_vars_out], int deg, bool print){

      // Don't worry about wasting some space, the array is small enough that it should not make
75    // a huge deal. This is a better saving then having to evaluate a multiplication

      // 59 is the max number of different terms this library allows (6 input variables and deg
          5)
      // and 59 is the max index I allow with the key generation for this set of conditions
      scalar calculatedTerms[59];
80
      for (int k = 0; k < num_vars_in; ++k) {
        for (int exp_ = 0; exp_ <= deg; ++exp_) {
          int index = (k << 3) + k + exp_;
          calculatedTerms[index] = intpow(x0[k], exp_);
85      }
      }

      for (int i = 0; i < num_vars_out; ++i) {
        scalar result = 0;
90      for (int j = (int)equations[i].terms.size() - 1; j >= 0; --j) {
          PolyTerm<scalar,num_vars_in> *term = &equations[i].terms[j];
          scalar term_value = term->coefficient;
          for (int k = 0; k < num_vars_in; ++k) {
            echar exp_ = term->exponents[k];
95          int key_ = (k << 3) + k + (int)exp_; // Unique key number for all possible values
            term_value *= calculatedTerms[key_];
          }
```

```
      result += term_value;
    }

    x1[i] = result;
  }
}

/**
 * Counts the number of terms within all of the polynomials in a system.
 * @return The number of terms within the system of equations
 */
_tSIO
scalar TruncPolySystem _SIO::count_terms() {
  scalar toReturn = scalar(0);
  for (int i = 0; i < num_vars_out; ++i) {
    toReturn += equations[i].getNumTerms();
  }

  return toReturn;
}

/**
 * Collects all of the coefficients of all the polynomials within a system.
 * @param  exps All of the possible exponentiations of the variables within a polynomial
 * @return      A vector containing all of the coefficients of all the terms within the
    system
 */
_tSIO
vector<scalar> TruncPolySystem _SIO::collectCoeff(vector<vector<int> > exps) {
  vector<scalar> coeff;
  for (int eq = 0; eq < num_vars_out; ++eq) {
    for (vector<vector<int> >::iterator it = exps.begin(); it != exps.end(); ++it) {
      coeff.push_back(t\Ttequations[eq].get_coeff((*it)[0],(*it)[1],(*it)[2],(*it)[3]));
    }
  }
  return coeff;
}
```