EUROGRAPHICS Symposium on Rendering (2012)
F. Durand and D. Gutierrez (Guest Editors)

Supplemental Document for "Polynomial Optics"

Matthias B. Hullin[†1] Johannes Hanika[2] Wolfgang Heidrich [1]

[1]The University of British Columbia [2]Weta Digital

Abstract

Our paper on Polynomial Optics, presented at the EUROGRAPHICS Symposium on Rendering 2012, describes the mapping of light rays through an optical system in terms of a system of multivariate polynomials. With this supplemental document, we provide additional information on two main aspects:

1. ***Scalar-valued and other functions.*** *In the paper, we focus on the geometric mapping of rays onto rays. As a matter of fact, other functions of the ray parameters can be treated in a very similar way. This includes the attenuation of light, for instance by Fresnel terms, absorption or occlusion. We construct an example system that modulates both geometry and intensity of a light ray.*
2. ***Introduction to the C++ library.*** *Furthermore, we introduce the interface to our C++ implementation of Polynomial Optics. The full code, along with usage examples, is provided on the project homepage,* `http://www.cs.ubc.ca/labs/imager/tr/2012/PolynomialOptics/`.

1. Ray intensity and other scalar modulation

In our paper on Polynomial Optics, we elaborate on how the geometric mapping of light rays through an optical system can be described in terms of polynomial functions. A very similar approach can also be used to describe scalar functions of the ray paramaters, most prominently the change in radiance along a ray. Various factors can contribute to such modulation; the most prominent examples are arguably partial reflection or transmission (Fresnel terms), and occlusion of the light beam by lens barrels and apertures.

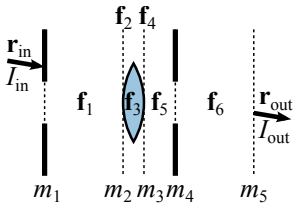

Figure 1: *A system consisting of a lens and two apertures, resulting in the ray mappings ($\mathbf{f}_i$) and modulations (m_i).*

Consider the optical system shown in Figure 1. It consists of elements that affect both the geometry and intensity of a ray that passes through it. The ray mappings $\mathbf{f}_1 \ldots \mathbf{f}_6$ describe propagation and refraction of the ray: $\mathbf{f}_{\{1,3,5,6\}}$ are propagations and $\mathbf{f}_{\{2,4\}}$ spherical refractive interfaces.

The scalar *modulation functions* $m_1 \ldots m_5$ describe the relative change of intensity that an incoming ray experiences at a given interface, as a function of the ray parameter vector:

$$m(\mathbf{r}) = \frac{I_{\text{out}}(\mathbf{r})}{I_{\text{in}}} \tag{1}$$

The total ray mapping of the system is given as:

$$\mathbf{f}_{\text{system}}(\mathbf{r}) = (\mathbf{f}_6 \circ \mathbf{f}_5 \circ \mathbf{f}_4 \circ \mathbf{f}_3 \circ \mathbf{f}_2 \circ \mathbf{f}_1)(\mathbf{r}) \tag{2}$$

In order to determine the modulation factor in the various planes, only a part of this concatenation needs to be evaluated. For instance, if the function m_3 describes the Fresnel transmission through the second lens interface, the attenuating factor M_3 of that surface is computed as:

$$M_3(\mathbf{r}) = (m_3 \circ \mathbf{f}_3 \circ \mathbf{f}_2 \circ \mathbf{f}_1)(\mathbf{r}) \tag{3}$$

The total attenuation M_{total} is then simply the product of the individual factors $M_1 \ldots M_5$.

If analytical expressions are known for all the modulation functions involved, we can determine their product and,

† hullin@cs.ubc.ca

again, approximate it by means of a Taylor expansion:

$$M(r_x, r_y, d_x, d_y) = \sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\sum_{c=0}^{\infty}\sum_{d=0}^{\infty} \beta_{(a,b,c,d)} \cdot r_x^a \cdot r_y^b \cdot d_x^c \cdot d_y^d \tag{4}$$

This is not always the case; some functions are not continuous and therefore their value has to be computed per ray. A typical example is occlusion; let us assume that the entrance pupil of our system, m_1, is of round shape with radius R_1:

$$m_1(r_x, r_y, d_x, d_y) = \begin{cases} 1 & \text{if } \sqrt{r_x^2 + r_y^2} < R_1 \\ 0 & \text{else} \end{cases} \tag{5}$$

To determine the total attenuation of our ray from Fresnel terms, Lambertian cosine and apertures combined, it therefore sometimes makes sense to combine both concepts:

$$M_{\text{total}}(\mathbf{r}) = M_1(\mathbf{r}) \cdot M_{\text{F}}(\mathbf{r}) \cdot M_4(\mathbf{r}) \tag{6}$$

where M_{F} combines the Fresnel terms m_2 and m_3 as well as the Lambertian factor $m_5 = \cos \angle(\mathbf{r}, \hat{\mathbf{z}})$. The individual terms are as follows:

$$\begin{aligned} M_1(\mathbf{r}) &= m_1(\mathbf{r}), \\ M_{\text{F}}(\mathbf{r}) &= ((m_2 \circ \mathbf{f}_1) \cdot (m_3 \circ \mathbf{f}_3 \circ \mathbf{f}_2 \circ \mathbf{f}_1) \cdot (m_5 \circ \mathbf{f}_{\text{system}}))(\mathbf{r}), \\ M_4(\mathbf{r}) &= (m_4 \circ \mathbf{f}_4 \circ \mathbf{f}_3 \circ \mathbf{f}_2 \circ \mathbf{f}_1)(\mathbf{r}). \end{aligned}$$

$M_{\text{F}}(\mathbf{r})$ can in principle be Taylor-expanded, but the apertures $M_1(\mathbf{r})$ and $M_4(\mathbf{r})$ will need to be evaluated separately for each ray $\mathbf{r}$, and then multiplied to obtain $M_{\text{total}}(\mathbf{r})$.

2. C/C++ Library

Our implementation of polynomial optics is divided in two layers: a set of classes that provide very general algebraic functionality, and a library of optical elements that produces instances of these classes. Additional helper tools assist the process.

Class templates. The following set of C++ classes is provided, each templated with the scalar type and the number of input/output variables:

`PolyTerm<scalar,numvars>` – A monomial, consisting of a coefficient, and nonnegative, integer exponents for each of the variables.
`TruncPoly<scalar,numvars>` – A truncated polynomial, consisting of a list of polyTerms and a truncation degree.
`TruncPolySystem<scalar,numVarsIn,numVarsOut>` – A system of `numVarsOut` truncated polynomials, each in `numVarsIn` variables of type `scalar`.

The classes implement basic algebraic functionality including inter-class operators where they make sense (e.g., addition and multiplication of terms and polynomials). Such operations are automatically truncated according to the current truncation degree of a polynomial.

For the most likely template parameters, predefined types are provided, such as `Term2f` (a single-precision monomial in two variables), `Poly3d` (a double-precision polynomial in three variables) and `System42f` (a system of two equations, each a single-precision polynomial in four variables).

Library of optical elements. Most optical elements are abstracted as `TruncPolySystem<float,4,4>` of degree 5, where input and output are both reduced ray vectors. Geometric derivation and Taylor expansion of the ray tracing solution was performed using the symbolic algebra package Maple, then converted to C/C++.

```
System44f refract_spherical_5(float R, float
n1, float n2);
System44f reflect_spherical_5(float R);
System44f refract_cylindrical_x_5(float R,
float n1, float n2);
...
System44f propagate_5(float dist);
```

Rays are often conveniently parameterized by their intersection with two parallel planes. The conversion to reduced ray vectors, when done beforehand, can cause excessive roundoff error if the distance between the planes is large. The library therefore offers an element that allows the user to integrate two-plane parameterization into the polynomial system, which resolves such problems in most cases.

```
System44f two_plane_5(float dist);
```

Additional helper tools, as well as a database of optical glasses, complement the library.

Usage example. The following code snippet shows the construction of a simple biconvex lens from scratch.

```
OpticalMaterial glass("N-BK7");
float lambda = 550; // nanometers
float n1 = glass.get_index(lambda);

float dist0 = 200, R1 = 100, R2 = -100,
thickness = 10;
System44f system = two_plane(dist0)
  >> refract_spherical_5(R1, 1.f, n1)
  >> propagate_5(thickness)
  >> refract_spherical_5(R2, n1, 1.f);
float focal_dist = find_focus_X(system);
system = system >> propagate_5(focal_dist);
float magni = get_magnification_X(system);
std::cout << system; // output equations

// Define ray as world x y, pupil x y:
float ray_in[4] = {1,2,1,2};
float ray_out[4];
system.evaluate(ray_in, ray_out);
```

Data export. Coefficients and exponents of the terms in a polynomial are stored linearly in memory, making it easy to perform the high-level polynomial handling in C++ and transferring the resulting data over to other computing frameworks such as OpenCL or CUDA.