Supplemental Document for “Polynomial Optics”

Matthias B. Hullin† 1, Johannes Hanika2, 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 parameters, 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.

\[ M_3(r) = (m_3 \circ f_3 \circ f_2 \circ f_1)(r) \]

The total attenuation \(M_{\text{total}}\) 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,
again, approximate it by means of a Taylor expansion:
\[
M(r_x, r_y, a_x, a_y) = \sum_{a=0}^{\infty} \sum_{b=0}^{\infty} \sum_{c=0}^{\infty} \sum_{d=0}^{\infty} \beta_{a,b,c,d} \cdot r_x^a \cdot r_y^b \cdot a_x^c \cdot a_y^d
\]
(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 \(R_1\):
\[
m_1(r_x, r_y, a_x, a_y) = \begin{cases} 
1 & \text{if } \sqrt{r_x^2 + r_y^2} < R_1 \\
0 & \text{else}
\end{cases}
\]
(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:
\[
M_{\text{total}}(r) = M_1(r) \cdot M_F(r) \cdot M_3(r)
\]
(6)

where \(M_F\) combines the Fresnel terms \(m_2\) and \(m_3\) as well as the Lambertian factor \(m_5 = \cos \beta_a \cdot \hat{\beta}_b\). The individual terms are as follows:
\[
M_1(r) = m_1(r),
M_F(r) = \left((m_2 \cdot f_1) \cdot (m_3 \cdot f_2 \cdot f_3 \cdot f_4) \cdot (m_5 \otimes f_{\text{system}})\right)(r),
M_3(r) = (m_3 \otimes f_3 \cdot f_2 \cdot f_1)(r).
\]

\(M_3(r)\) can in principle be Taylor-expanded, but the apertures \(M_1(r)\) and \(M_3(r)\) will need to be evaluated separately for each ray \(r\), and then multiplied to obtain \(M_{\text{total}}(r)\).

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 **System2f** (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 two_plane_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.

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.

```cpp
float lambda = 550; // nanometers
float n1 = glass.get_index(lambda);
float dist0 = 200, R1 = 100, R2 = -100, thickness = 10;

float n1, float n2);
refract_spherical_5(R1, 1.f, n1)
propagate_5(thickness)
refract_spherical_5(R2, n1, 1.f);
float focal_dist = find_focus_X(system);
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];
M. Hullin et al. / Polynomial Optics, Supplemental Document

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.

© The Eurographics Association 2012.