Fachhochschule Wedel, University of Applied Sciences
Faculty of Media Information Science

Realistic Materials and Lighting in Real-Time Rendering

30 August 2001

Diploma Thesis/Diplomarbeit of
Lutz Latta
Im Winkel 18
22880 Wedel
Germany

Supervisor
Prof. Dr. Andreas Kolb
Feldstrasse 143
22880 Wedel
Germany

Table of Contents

Table of Contents....2
List of Figures....4
Abstract....5
1 Introduction....6
2 Fundamentals....8
2.1 Vector Notation....8
2.2 Spherical Coordinates....9
2.3 Bidirectional Reflectance Distribution Function (BRDF)....9
2.3.1 BRDF Parameters....10
2.3.2 Simplification of Parameters....11
2.3.3 Definition of the BRDF....12
2.3.4 BRDF-based Lighting Computation....14
2.3.5 Data Sources for BRDFs....14
2.4 Sparse Matrices....15
2.5 Iterative Solvers....15
3 Prior Work....18
3.1 BRDF Approximation....18
3.1.1 BRDF Separation....18
3.1.2 Homomorphic Factorization (HF)....22
3.1.3 Evaluation of Separation Methods....28
3.2 Environment Mapping....29
3.2.1 Mirror-like Environment Maps....29
3.2.2 Glossy Environment Maps....29
4 Improvements to the Homomorphic Factorization....33
4.1 Flexible Parametrizations....33
4.2 Sample Selection....34
4.3 Texture Precision....35
4.3.1 Prescaling of Textures....37
4.3.2 Alpha Blending Contouring Prevention....38
5 Factorization of Lighting Computation....43
5.1 Motivation....43
5.2 Approach....44
5.3 Differences to BRDF Factorization....45
5.4 Parametrization....45
5.5 Texture Mapping....48
5.6 Function Sampling....49
5.7 Results....50
5.8 Evaluation....56
6 Implementation Details....57
6.1 Basic concepts....57
6.1.1 Image....57
6.1.2 BRDFSampler....58
6.1.3 LightingEquationSampler....58
6.1.4 DirectionIterator....58

6.1.5 SampleIterator....59
6.1.6 DirectionMapper....60
6.1.7 Parametrization....61
6.1.8 BRDFSeparator....62
6.1.9 Rendering Scene Graph....64
6.2 User's Guide....68
6.2.1 BRDFSeparation....68
6.2.2 BRDFViewer....69
7 Conclusion and Further Work....72
Acknowledgments....73
Bibliography....74

List of Figures

Figure 1: Spherical coordinate reference frame........9
Figure 2: Light surface interaction........10
Figure 3: Differential solid angle as a small area on the unit sphere........13
Figure 4: Light spreads out on a surface by cosine of incident angle........13
Figure 5: Incoming light hemisphere........14
Figure 6: Texture mapping types (hemispherical, parabolic, cubic)........19
Figure 7: LaPlace operator........26
Figure 8: Alpha computation precision........41
Figure 9: Contouring of separated gold BRDF (left: without alpha, right: with alpha).....42
Figure 10: Golden teapot with complex lighting........44
Figure 11: Layout of multiple maps for one texture (left: parabolic, right: cube map)....49
Figure 12: Environment maps (Loch, Desert, painted light sources)........50
Figure 13: Teapot with exact lighting computation........52
Figure 14: Results of lighting computation factorizations........55
Figure 15: Result of glossy environment map with specular Phong model........55
Figure 16: Rendering sequence in DagBRDFMaterial........66
Figure 17: BRDFViewer user interface........70

Abstract

Several techniques have been developed to approximate Bidirectional Reflectance Distribution Functions (BRDF) with acceptable quality and performance for real-time applications. In this thesis these techniques are presented and compared. The recently published “Homomorphic Factorization” [McCool2001] is a rather general approximation approach usable with various setups and for different quality requirements.

This thesis introduces a technique based on the homomorphic factorization, that does not approximate the BRDF, but instead the full lighting computation of an isotropic BRDF in a global illumination scenario. This can also be considered as a generalized approach to several environment map prefiltering techniques. A survey about these methods is given and the advantages of the new algorithm are explained.

During the process of developing the new technique several small improvements to the original BRDF factorization algorithm have been made and are presented as well.

1 Introduction

With steadily increasing hardware capabilities in real-time 3D graphics, a point has been reached where only increasing the polygon count of a scene does not offer significant improvements in visual quality and photo-realism. Instead some of the computation power of a graphics system can be used to render more complex and more realistic materials.

The Bidirectional Reflectance Distribution Functions (BRDF) of a material describes how the incoming light at a surface point is reflected towards the direction of the viewer. Since this function is usually quite complex, it can be considered to be not efficiently computable during real-time rendering. Some popular BRDFs (for example the Cook-Torrance model [Cook1981]) are however computable on a per-vertex basis on recent hardware systems for a very limited number of light sources.

Approximation techniques for BRDFs usually compute several textures that are mapped onto a geometry with real time generated texture coordinates. By multiplying or summing several textures either with multitexturing or during multiple render passes, an approximation of a BRDF is reconstructed.

The approximation of a BRDF however is only the simulation of a single light source. If a scene contains many lights, the BRDF approximation has to be applied multiple times to one object. A two texture approximation with only three light sources would require six rendering passes on a single-texture graphics system. In a global lighting situation, where the number of distinct light sources can be regarded as striving towards infinity, the technique breaks down.

As an alternative to the BRDF itself, the full lighting computation can be approximated. It can be shown, that the lighting computation of an isotropic BRDF (a class of materials where a planar surface can be rotated around its normal without a change of appearance) has several functional properties in common with a BRDF. Therefore the lighting computation can be approximated in a similar fashion. However the full lighting function lacks certain symmetries that BRDFs usually have, for example a radial symmetry around some axis. This problem can be solved by choosing new function parameter sets.

A visually satisfying approximation has significant advantages over the BRDF approximation, as it can capture realistic lighting environments and limits the amount of necessary real-time rendering effort even for only few light sources. Between two and four textures have proven to be sufficient for acceptable approximations of most lighting computations.

The approximation technique can also be regarded as a generalized approach to several environment mapping prefiltering methods. The incoming light function in a

global illumination scenario can be represented with an environment map. Computing the surface lighting with some simple BRDFs (for example the specular part of the Phong model) produces a result that looks like a blurred version of the environment map. Therefore these techniques are called prefiltering methods, and can be shown to be a special case for the general approximation introduced here.

Several practical problems of the BRDF approximation process have occurred while implementing the original algorithm. One of these problems is the high dynamic range of some BRDFs, especially if the described material has high specular peaks. An improvement proposed in this thesis is the additional use of the alpha channel of a texture to dynamically scale the range of the texture.

Chapter 2 introduces the terminology and notation used throughout the thesis. Then previously known approximation and prefiltering techniques will be explained (chapter 3). The homomorphic factorization of McCool et al. is described in further detail. The mentioned small improvements to this method are explained in chapter 4. Then the new method for the approximation of the lighting computation will be discussed (chapter 5). Details about our implementation of the algorithm are given in chapter 6. Finally chapter 7 concludes the thesis and suggest future enhancements.

2 Fundamentals

In this chapter the basic symbols, names and concept used in the context of this thesis are defined. After a brief description of the used vector notation and spherical coordinates, BRDF-based lighting is introduced. Readers familiar with the concept can safely skip this section, only the used function notation and symbol definition might be of interest. Additional sections about sparse matrices and iterative solvers for linear equation systems give a brief introduction to the topics used later for BRDF separations.

2.1 Vector Notation

Vectors are written in this thesis with a hat over the variable name: $\hat{a}=\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$

For better readability vectors in the text will be written as comma separated row vectors like (a_1, a_2, a_3) .

Usually vectors are three component vectors, except vectors specifying directions in spherical coordinates (see below) and vectors explicitly marked as general n-component vectors.

The dot product of two vectors is written as: $\hat{a}\cdot\hat{b}=\sum_{i=1}^{n} a_i b_i$

The cross product of two vectors $\hat{a}\times\hat{b}$ results in a vector orthogonal to both operands.

A general vector norm is written as $\|\hat{a}\|$. A specific norm contains an index, like:

$$\begin{aligned} \|\hat{a}\|_1 &= |a_1|+|a_2|+\ldots+|a_n| \\ \|\hat{a}\|_2 &= \sqrt{a_1^2+a_2^2+\ldots+a_n^2} \\ \|\hat{a}\|_\infty &= \max_i |a_i| \end{aligned} \tag{1}$$

Some vectors in this thesis represent a color consisting of three components for red, green and blue. They can be written as:

$$\hat{c}=\begin{pmatrix} c_r \\ c_g \\ c_b \end{pmatrix} \tag{2}$$

For color vectors it is sometimes practical to use a different vector multiplication than the dot or cross product. It is basically a component-wise multiplication, written just like a multiplication of scalar values:

$$\hat{a}\,\hat{c}=\begin{pmatrix} a_r c_r \\ a_g c_g \\ a_b c_b \end{pmatrix} \quad (3)$$

2.2 Spherical Coordinates

A vector can be represented in several forms. In the computer graphics area the commonly used representation is in cartesian coordinates. However for the specification of a direction in 3D space, spherical coordinates are easier to handle. The vector is specified with two angles towards a reference frame and the magnitude of the vector. For the specification of a direction the magnitude is not important and can be ignored.

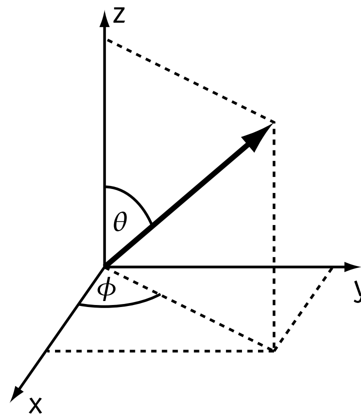


Figure 1: Spherical coordinate reference frame

The two angles are usually called theta θ and phi ϕ, and represent the angle of the vector towards the z direction and of a vector projected into the x/y plane towards the x direction respectively.

In this thesis vectors in spherical coordinates are written with the Greek letter omega like $\hat{\omega}$.

Some vectors will have a special reference frame, that is used for the computation of the angles. These vectors will then be written as a function like $\hat{\omega}(\hat{x}, \hat{a}, \hat{b})$. The vector $\hat{x}$ will be computed by measuring the angles relative to the coordinate frame defined by $\hat{a}, \hat{b}, \hat{a}\times\hat{b}$.

2.3 Bidirectional Reflectance Distribution Function (BRDF)

The BRDF and its lighting model represent a very general physically based lighting concept, that can be used to simulate any natural and artificial reflective material. Reflective materials are also called opaque or non-translucent. Their attributes are dominated by light reflection and absorption on the surface of the material.

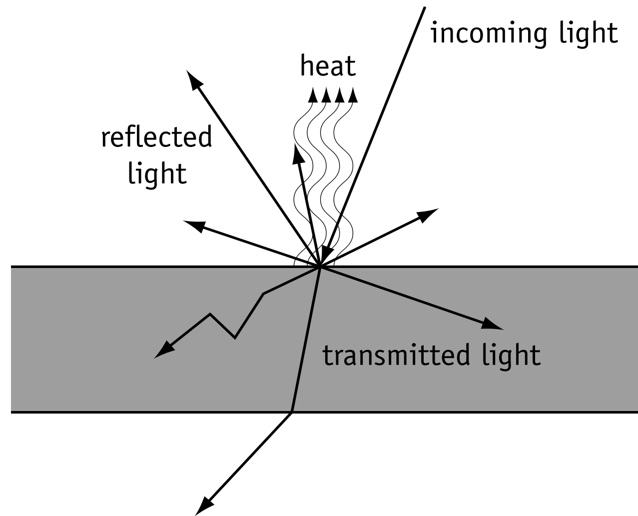


Figure 2: Light surface interaction

According to the physical law of conservation of energy (light is a form of energy), all light energy incident at a point of a surface, needs to leave that point some way. Light on a surface can be transformed in three ways:

- Absorption: The light energy is converted to heat
- Reflection: The light reflects on the surface to the general direction it came from
- Transmission: The light traverses through the surface and interacts in some way with the material below the surface

The first two ways of light-surface interaction are represented in a BRDF, the third one can be represented with similar functions, but will not be discussed any further in this thesis.

Absorption is handled by a BRDF in an implicit way, by assuming that all light that is not reflected has been absorbed.

2.3.1 BRDF Parameters

The BRDF can be generally written as a function with four parameters:

$$BRDF_{\lambda}(\hat{\omega}_i, \hat{\omega}_o, \hat{x}) \quad (4)$$

where λ is the wavelength of the light, $\hat{\omega}_i$ and $\hat{\omega}_o$ are the directions of incoming and outgoing light in spherical coordinates and $\hat{x}$ indicates the position on the surface.

The BRDF depends on the wavelength of the light λ. Different wavelengths usually show different amounts of absorption. A surface looking red reflects light of a red wavelength into the viewer's eye, but absorbs light of other wavelengths like blue.

The BRDF is also dependent on a position $\hat{x}$ on the surface. This positional variance is caused by the heterogeneous properties of materials. A material like wood consists of different densities in the rings and between the rings, which causes varying appearances. This property can be thought of as generalization of the common texture concept, where a color image defines the color of each point on the surface.

The other two parameters, $\hat{\omega}_i$ and $\hat{\omega}_o$, are the directions of incoming and outgoing light in spherical coordinates. Both the incoming and the outgoing light direction are vectors starting at the surface point under consideration. Of all the possible outgoing light directions, the most important one is the one towards the viewer, although the other directions might be considered in light models using multiple surface reflection.

2.3.2 Simplification of Parameters

A BRDF in its complete form has seven scalar valued degrees of freedom, considering that the three vectors have at least two components. To simplify the BRDF for the methods discussed later, some of the BRDF parameters will only be discussed briefly in this section.

The wavelength dependency of the BRDF can usually be replaced by regarding the BRDF for the three components of the red-green-blue (RGB) color representation commonly used in computer graphics. Only for BRDFs with light interference contributions (like the rainbow-colored reflections of an oil-water surface or of a compact disc) a more complicated calculation is necessary. Then the BRDF needs to be computed for a number of wavelengths that are then weighed by CIE color matching functions and integrated (see [CIE1986] and [Kautz1999]). In the context of this thesis, the wavelength dependency of a BRDF is not considered any further and all formulas with BRDF values can be applied to the three RGB components separately.

The positional variance $\hat{x}$ of a BDRF can sometimes be approximated with a color texture as mentioned above. Then the color in a texture can be used as a modulation of a position invariant (also called shift invariant) BRDF. If the texture is a function called t, this can be written as:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o)\, t(\hat{x}) \qquad (5)$$

To approximate a more complex position variant BRDF, it might be split up into several position invariant BRDFs, that are applied separately to parts of a geometry, e.g. with splitting the geometry or using alpha blending techniques. From here on, this thesis will concentrate on position invariant BRDFs.

Finally there are materials where the two light direction parameters of the BRDF do not need to be considered in full. The so called isotropic materials show a radial

symmetry around the surface normal. This can be thought of as having a fixed light and viewer position and then rotating the material around its normal. In this situation the value of an isotropic BRDF does not change. Materials having a structure predominantly going in one direction, for example woven cloth or brushed metal, do not have isotropic BRDFs. These are called anisotropic BRDFs.

To show the isotropic property of a BRDF mathematically, the two directions of a BRDF have to be split into the two angles of their spherical coordinates:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = BRDF(\theta_i, \phi_i, \theta_o, \phi_o) \tag{6}$$

An isotropic BDRF does not change if both ϕ angles change equally (by any δ), and therefore these parameters can be combined:

$$BRDF(\theta_i, \phi_i, \theta_o, \phi_o) = BRDF(\theta_i, \phi_i + \delta, \theta_o, \phi_o + \delta) = BRDF'(\theta_i, \theta_o, \phi_i - \phi_o) \tag{7}$$

The BRDF has been reduced to three parameters in this equation. Both three parameter isotropic BRDFs and four parameter anisotropic BRDFs will occur throughout this thesis.

2.3.3 Definition of the BRDF

So far the actual definition of the BRDF has not been discussed. It requires a small excursion into physics.

If the amount of light reflected in direction $\hat{\omega}_o$ is called $L(\hat{\omega}_o)$ and the amount of light energy hitting the surface from direction $\hat{\omega}_i$ is called $E(\hat{\omega}_i)$ then the BRDF can be written as:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = \frac{L(\hat{\omega}_o)}{E(\hat{\omega}_i)} \tag{8}$$

The incoming light energy $E(\hat{\omega}_i)$ is not identical to the amount of light having left a light source that lies in direction $\hat{\omega}_i$.

First of all the light arriving at a surface point can not be measured as coming from exactly one direction. Instead it has to be described as the amount of light passing a (possibly infinitesimal) small surface area. This leads to the concept of differential solid angles (see [Wynn2000]). The differential solid angle $d\hat{\omega}$ can be thought of as the area of a small surface element on a unit sphere:

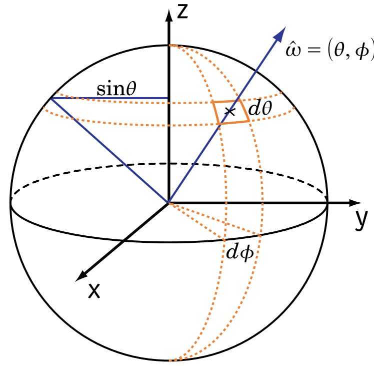


Figure 3: Differential solid angle as a small area on the unit sphere

For a direction in spherical coordinates (θ, ϕ) and small differential angular changes $d\theta$ and $d\phi$ the area of $d\hat{\omega}$ can be calculated like a rectangle:

$$d\hat{\omega} = width \cdot height = (d\phi \sin\theta)(d\theta) = d\theta \, d\phi \sin\theta \tag{9}$$

Besides weighing the amount of incoming light with the differential solid angle, it must also be considered that the amount of energy arriving at the surface spreads out depending on the angle of impact. This is also known from the diffuse part of the Phong lighting model. This effect can be formulated by weighing the energy with $\cos\theta = \hat{n} \cdot \hat{\omega}_i$ where $\hat{n}$ is the normal of the surface.

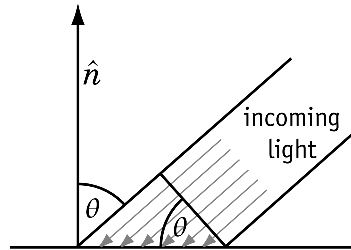


Figure 4: Light spreads out on a surface by cosine of incident angle

All taken into account this leads to the following equations, where $L(\hat{\omega}_i)$ is the amount of light emitted by a (theoretical) point light source in direction $\hat{\omega}_i$:

$$\begin{aligned} E(\hat{\omega}_i) &= L(\hat{\omega}_i) \cos\theta_i \, d\hat{\omega}_i \\ BRDF(\hat{\omega}_i, \hat{\omega}_o) &= \frac{L(\hat{\omega}_o)}{E(\hat{\omega}_i)} = \frac{L(\hat{\omega}_o)}{L(\hat{\omega}_i) \cos\theta_i \, d\hat{\omega}_i} \end{aligned} \tag{10}$$

Note that although the relation of $L(\hat{\omega}_o)$ to $L(\hat{\omega}_i)$ is in the range [0, 1], the BRDF itself is not restricted to this interval. Since the cosine is less or equal one, the division result can be greater than one.

2.3.4 BRDF-based Lighting Computation

By simply shifting around the definition formula of the BRDF, we can compute the amount of light reflected in the viewer's direction $\hat{\omega}_o$ from one point light source lying in direction $\hat{\omega}_i$:

$$L(\hat{\omega}_o) = BRDF(\hat{\omega}_i, \hat{\omega}_o) L(\hat{\omega}_i) \cos\theta_i d\hat{\omega}_i \tag{11}$$

In the real world there are no point light sources and instead light coming from all directions of the hemisphere surrounding the surface contribute to its appearance.

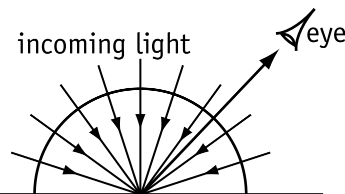


Figure 5: Incoming light hemisphere

This can be achieved by integrating over all incoming light directions:

$$L(\hat{\omega}_o) = \int_{\Omega} BRDF(\hat{\omega}_i, \hat{\omega}_o) L(\hat{\omega}_i) \cos\theta_i d\hat{\omega}_i \tag{12}$$

The omega Ω in the equation represents the whole hemisphere of incoming directions.

In the discrete case, the integral can be replaced by a summation and the $d\hat{\omega}_i$ can be set to one, indicating that all incoming directions are weighed equally:

$$L(\hat{\omega}_o) = \sum_{\Omega} BRDF(\hat{\omega}_i, \hat{\omega}_o) L(\hat{\omega}_i) \cos\theta_i \tag{13}$$

Note that here Ω does not necessarily represent all directions of the hemisphere, only directions whose light intensity are known and are above zero need to be considered.

2.3.5 Data Sources for BRDFs

Knowing how to compute BRDF-based lighting is worth nothing, if the BRDF data, the function values of the BRDF, for a specific material is not known.

In general there are two sources for BRDF data. One is the field of analytical models that can be formulated as a BRDF. By specifying the function parameters for a BRDF, the analytical model can calculate the BRDF value. The Phong model for example can be represented as a BRDF and evaluated for a given parameter set.

Another source for BRDFs is physically measured data for materials. With devices like a gonioreflectometers BRDF data samples can be measured. The sampling is sometimes not

very dense, and some viewer-light combinations cannot be measured. So in practice usually some sort of filtering or interpolation is necessary.

2.4 Sparse Matrices

Later in this thesis numerical techniques with sometimes huge matrices will be used.

It is a quite common case that a big matrix contains a lot of zero values. A matrix showing this property is called a sparse matrix. Storing such sparse matrices completely, that is as one big array of values, is both computationally and under memory considerations inefficient.

The components of a matrix that are zero do not contribute to the result of most mathematical operations, like matrix multiplications or matrix-vector products. To store the matrix in a compressed form is obvious.

Depending on the layout of the non-zero values in the matrix, several compression methods can be chosen. The compressed row storage method stores for each row only the column positions and values of the non-zero components. The compressed column is analogue for columns.

Matrices predominantly filled around the main diagonal, can be stored with all values up to a certain distance around the diagonal. Similar compression as above can be applied as well.

Matrices with randomly distributed values can be stored with a pair of coordinates and the associated value of all non-zero components.

2.5 Iterative Solvers

Some of the techniques presented in this thesis deal with numerical algorithms, especially iterative solvers.

Iterative solvers can be used to compute a solution for linear equation systems. That is a system consisting of equations in the form

$$a_1 x_1 + a_2 x_2 + a_3 x_3 + \ldots + a_n x_n = b \tag{14}$$

Where x_i are the variables to compute, a_i and b are constants. A number of these equations assembled in the rows of a matrix A and two vectors $\hat{x}$ and $\hat{b}$ lead to the equation system in matrix form:

$$A \hat{x} = \hat{b} \tag{15}$$

The equation system might be over or under constrained, might have infinite, one or no solution.

Several iterative algorithms exist that try to compute a possible solution to the equation system as good as possible with a finite number of steps (iterations). These algorithms usually cannot guarantee to find an exact or even the best possible solution, since the algorithms can only consider solutions somehow related to a previous result. Sometimes there are cases where an algorithm breaks down completely, maybe due to the fact that there is no (at least almost exact) solution to the equation system or because some other constraint is not fulfilled.

Most iterative solver algorithms have two stop criteria. One is the maximum number of iterations to be done. The other is a tolerance threshold: If a certain attribute of the equation system, for example $\|A\hat{x}-\hat{b}\|$, goes below the tolerance threshold, the algorithm stops.

A classical iterative method for solving linear equation systems is the Conjugate Gradient (CG) method [Hestenes1952]. It has been improved over the years and several variations exists that have a high efficiency at solving linear equation systems with a symmetric (Hermitian positive) coefficient matrix.

The algorithm used in this thesis does not have this restriction. The Quasi-Minimal Residual (QMR) method [Freund1991] can compute equation systems with both unsymmetric and symmetric matrices. However the matrix still needs to be a square matrix.

To compute any kind of matrix with a method only allowing square or symmetric matrices, the equation system can be wrapped in another equation system:

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} \hat{r} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} \hat{b} \\ 0 \end{bmatrix} \quad (16)$$

where I is a diagonal matrix with ones on the diagonal, A^T is A transposed and $\hat{r}$ is a vector that will be sent to zero by the solver. This can be seen more clearly after transforming the matrices to these two equations:

$$\begin{aligned} \hat{r}+A\hat{x}&=\hat{b} \\ A^T\hat{r}&=0 \end{aligned} \quad (17)$$

Often implementations of the iterative methods do not require an explicit storage of the matrix, instead they only need computation of the matrix-vector product. Therefore it is not necessary to store the new matrix in equation 16 explicitly, the matrix-vector product can probably be computed faster by using the knowledge about the special symmetry.

Another important issue for iterative methods is preconditioning. The number of iterations necessary to reach a result of a certain precision can be greatly reduced by using an algorithm with preconditioners. A preconditioner approximates in some sense the coefficient matrix A, but is faster to compute. The QMR algorithm uses two preconditioners, a left and a right one. Their matrix product shows the mentioned approximation characteristics. Several techniques for finding preconditioners exist, most of them are closely related to a certain matrix form, for example matrices predominantly filled around the main diagonal. Examples of preconditioners are the Incomplete LU, the SSOR or the Incomplete Cholesky preconditioner.

Several implementations of iterative solvers and matrix data types are freely available to the public. In the process of writing this thesis the IML++/SparseLib++ library of the NIST [NIST1996] and the ITL/MTL package of the University of Notre Dame [UND2000] have been tested.

While ITL is supposed to be faster than IML++, it is harder to add specialized matrix computations (like the matrix-vector product mentioned above) due to a complex deeply nested C++ template structure. That aside ITL seems to be the more advanced system, with a higher problem abstraction and greater flexibility that does not compromise efficiency.

3 Prior Work

This chapter will briefly look at BRDF approximations in general, explain Kautz's separation methods and McCool's homomorphic factorization in detail and finally present several environment map prefiltering techniques.

3.1 BRDF Approximation

A number of techniques have been developed over the years to approximate BRDFs. A lot of them only apply to one BRDF or a very restricted class of BRDFs. Even the Phong lighting computation could be regarded as a (perfect) approximation of a Phong BRDF. Due to the trivial nature of this, it will be not considered any further.

A number of general approximation techniques have been developed for non real-time rendering. According to Kautz and McCool [Kautz1999II] these include approximation with spherical wavelets [Schröder1995], with Zernike polynomials [Koenderink1996], the summation of generalized Phong cosine lobes [Lafortune1997] and spherical harmonics [Cabral1987]. Fournier was the first whose separation technique [Fournier1995] uses a sum of separable functions, that can be implemented via texture mapping and is therefore suitable for real-time rendering. Since the separation technique was intended for radiosity systems, it was not considered to implement it in real-time applications. This was introduced by Heidrich and Seidel [Heidrich1998], who analytically separated the anisotropic BRDF by Banks [Banks1994].

3.1.1 BRDF Separation

A more general separation suitable for a wide variety of BRDFs was developed by Kautz and McCool [Kautz1999]. In general the separation approximates the BRDF as a combination of several lower-dimensional functions:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = \sum_{j=1}^{J} u_j(\pi_u(\hat{\omega}_i, \hat{\omega}_o)) v_j(\pi_v(\hat{\omega}_i, \hat{\omega}_o)) \quad (18)$$

The functions u_j and v_j can then be pre-computed into two-dimensional textures as a look-up method for the function values. The functions $\pi_u(\hat{\omega}_i, \hat{\omega}_o)$ and $\pi_v(\hat{\omega}_i, \hat{\omega}_o)$ are projections which map the two BRDF directions to one new direction. In the simplest case, the u- and v-projection directly map to the incident and view direction respectively.

The approximation error can be controlled with the number of product terms J. The more terms are used, the more textures have to be combined in the real-time

reconstruction of the BRDF. Usually a small number of terms, with good projections even one might be sufficient for a number of BRDFs.

Reconstruction During Rendering

To reconstruct a separated BRDF, the textures are applied via multitexturing or with multiple rendering passes to a geometry. The texture coordinates for this have to be computed in real-time after every change of a light source or viewer position. The texture coordinate generation algorithm depends on the used projection functions π and the used method of texture mapping.

Since the textures are indexed by a direction on a hemisphere, several mappings commonly known from environment mapping techniques can be used. Hemispherical and parabolic [Heidrich1998II] mapping use the inner circle of standard 2D textures. A given direction is mapped to a texture position by projecting the intersection point of the direction vector with a sphere or a paraboloid respectively onto the flat texture. Cubic texture mapping can also be used, but requires additional hardware support. With cubic texture mapping six square images fill the six faces of a cube and a given direction vector starting from the cube center is intersected with these cube faces.

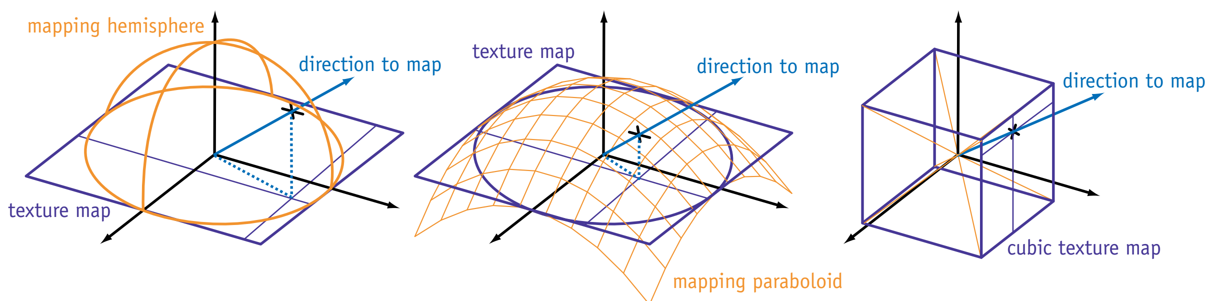


Figure 6: Texture mapping types (hemispherical, parabolic, cubic)

If the rendering hardware cannot compute the full product and sum in one pass i.e. completely in the texture combiner hardware, then the product of two textures has to be added via compositing in the frame buffer. If the hardware has no multitexture support at all, an accumulation buffer for storing the result of the product has to be used.

Additionally to the reconstruction of a BRDF from textures, the lighting computation (see equation 13) needs a multiplication by $\cos\theta_i$. Since this is equal to the diffuse part of the standard Phong lighting, the cosine term can be computed on a per-vertex basis, and then the Gouraud shaded diffuse color can be multiplied with the textures.

Precision tends to be a problem for reconstruction of some BRDF, especially with high specular peaks. Since textures and frame buffers can only store a range between $[0,1]$ but BRDFs can have values greater than one, some form of scaling to enhance the range needs to be done. A scaling by powers of two can be achieved by repeatedly adding data

in an accumulation buffer to itself. A better method uses the fixed scale factors in the texture pipeline of most recent hardware (for example via the OpenGL texture environment combiner extension), but these scaling factors apply only to the currently rendered texture combination. Using too high scaling factors at the end of a reconstruction process might impact visual quality, e.g. a factor of 8 reduces an 8 bit color component to only 5 bit data.

Singular Value Decomposition (SVD)

The most important issue with the BRDF separation method, is how to determine the approximation functions/textures. This has to be done only once for a specific BRDF and the result can be stored for later usage in rendering.

One method to decompose a BRDF is with a Singular Value Decomposition [Kautz1999]. For a SVD a matrix needs to be filled with regularly sampled values of the BRDF, varying one direction parameter over the columns the other one over the rows of the matrix. The SVD algorithm will then compute the approximation into matrices, that will be later used as the textures.

The problem with the SVD is that it has high computation and memory requirements. The matrix with BRDF samples needs to be quite big (for example $64 \times 64 \times 64 \times 64$ samples) and therefore not only needs a lot of memory but it is also slow to process. Another problem is that the SVD always minimizes absolute root mean square (RMS) error. Often this is not the visually most important criteria, because it tends to approximate specular highlights with greater precision than the base color. Finally a problem of the SVD is that the result might contain negative values, which poses a problem for most texturing hardware, that cannot do signed arithmetic.

Normalized Decomposition (ND)

Another decomposition method is the Normalized Decomposition [Kautz1999]. It is more a statistical method than a numerically accurate algorithm. Therefore it is much faster but usually less accurate than the SVD.

The ND starts with the same BRDF sample matrix as the SVD, where the rows and columns have one of the two BRDF parameters fixed. For a BRDF approximation with only a single product term, that is where J in equation 18 is one, the algorithm first computes a vector norm for each row of the matrix. The vector norms form the components of a new vector $\hat{u}$, which contains the values of one of the two approximation functions and will later fill one of the two textures. If we call each row of the matrix $\hat{m}_k$ and the norm is $\|\hat{m}_k\|$, then the values of the second approximation function are computed as

$$\hat{v} = \frac{1}{n} \sum_{k=1}^{n} \frac{\hat{m}_k}{\|\hat{m}_k\|} \tag{19}$$

The vector $\hat{v}$ is then used to fill the second texture. Basically any vector norm can be used for the computation, so the one with most visually pleasing result can be chosen.

If the error of a one term approximation is too high, the same algorithm can be applied to the difference of the original BRDF to the approximation. The resulting two textures can be considered as a second product term. These additional terms have the problem that they can contain negative values, and are therefore as problematic as the terms of the SVD.

Note that although the ND uses the same huge matrix as the SVD, its memory needs are far smaller. The matrix does not need to be stored explicitly, because the computation can be done on a row by row basis.

Reparametrization

So far the projection functions π_u and π_v in equation 18 have not been discussed. They are crucial for a good separation result. The most basic function set directly maps the incident and view directions to the functions:

$$\begin{aligned} \pi_u(\hat{\omega}_i, \hat{\omega}_o) &= \hat{\omega}_i \\ \pi_v(\hat{\omega}_i, \hat{\omega}_o) &= \hat{\omega}_o \end{aligned} \tag{20}$$

Most BRDFs are not very well separable in those two directions. Better separability can be achieved with halfangle-difference parametrizations. One such parametrization was introduced by Rusinkiewicz [Rusinkiewicz1998]. A more efficiently computable parametrization by Kautz uses Gram-Schmidt normalization.

It uses the halfangle vector $\hat{h}$ between the incident and view direction, and a difference vector $\hat{d}$. The halfangle vector is computed with the direction vectors in cartesian coordinates:

$$\hat{h} = \frac{\hat{\omega}_i + \hat{\omega}_o}{\|\hat{\omega}_i + \hat{\omega}_o\|} \tag{21}$$

The difference vector is somewhat less intuitive to compute. It can be thought of as the direction $\hat{\omega}_i$ in a new coordinate reference frame. The coordinate frame is built by the halfangle vector, a projection of the surface tangent and a vector orthogonal to both. Note that the surface tangent is a vector orthogonal to the surface normal, and has to be

known during the rendering of a geometry with a BRDF material in addition to the usual vertex data. If the tangent is called $\hat{t}$, the difference vector $\hat{d}$ can be computed as:

$$
\begin{aligned}
\hat{t''} &= \hat{t} - (\hat{t} \cdot \hat{h})\hat{h} \\
\hat{t'} &= \frac{\hat{t''}}{\|\hat{t''}\|} \\
\hat{b} &= \hat{h} \times \hat{t'} \\
\hat{d} &= \begin{pmatrix} \hat{\omega}_i \cdot \hat{t'} \\ \hat{\omega}_i \cdot \hat{b} \\ \hat{\omega}_i \cdot \hat{h} \end{pmatrix}
\end{aligned}
\tag{22}
$$

Using this parametrization the parameter projection functions become

$$
\begin{aligned}
\pi_u(\hat{\omega}_{i,} \hat{\omega}_o) &= \hat{h} \\
\pi_v(\hat{\omega}_{i,} \hat{\omega}_o) &= \hat{d}
\end{aligned}
\tag{23}
$$

3.1.2 Homomorphic Factorization (HF)

Recently a new separation method has been proposed by McCool, Ang and Ahmad [McCool2001]. Similar to the SVD it tries to find an optimal solution with a numerical algorithm, but it offers greater flexibility and avoids some drawbacks of the SVD approach. The amount of necessary computation effort lies somewhere between the SVD and the ND.

Basic Approach

The Homomorphic Factorization approximates a BRDF purely with a product of a number of two-dimensional functions. This can be written as:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = \prod_{j=1}^{J} p_j(\pi_j(\hat{\omega}_i, \hat{\omega}_o)) \tag{24}$$

Where p_j are the approximation functions to be stored in 2D textures, π_j are projections for the direction parameters as in equation 18 and J is the number of factors to be used.

Instead of solving this equation the HF is based on taking the logarithm on both sides of the equation. Writing $\log a$ as $\bar{a}$, this can be written as:

$$\overline{BRDF}(\hat{\omega}_i, \hat{\omega}_o) = \sum_{j=1}^{J} \bar{p}_j(\pi_j(\hat{\omega}_i, \hat{\omega}_o)) \tag{25}$$

Solving this equation has some significant advantages. First of all this equation is a linear problem, and it will be shown that it can be solved with a linear equation system. Another advantage is the fact that no negative texture values will occur, because exponentiating the approximation solution will only result in positive numbers. And finally if the linear problem is solved by minimizing the absolute RMS error, the solution of the original problem will have minimized relative RMS error. “This is perceptually desirable, since the eye seems to be roughly sensitive to ratios of intensity, not absolute intensity.” [McCool2001]

Taking the logarithm of the BRDF values also has some disadvantages. Since the BRDF can be zero, the logarithm is not defined for every value. This can be avoided by adding a small bias to the BRDF values. Additionally it is suggested to divide the BRDF values by the average value of all BRDF samples. This will move the logarithm of values near the average close to zero. Since most numerical algorithms send unmapped values in the solution to zero, they will be mapped to the average BRDF value. Naming the small bias ϵ and the average BRDF value a, the logarithmic transformation is:

$$\bar{BRDF}(\hat{\omega}_i, \hat{\omega}_o) = \log \frac{BRDF(\hat{\omega}_i, \hat{\omega}_o) + \epsilon a}{a} \tag{26}$$

The inverse transformation is:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = a\, e^{\bar{BRDF}(\hat{\omega}_i, \hat{\omega}_o)} - \epsilon a \tag{27}$$

The subtraction of ϵa possibly leads to negative values, but in practice the subtraction can be completely neglected, if ϵ is so small (e.g. 10^{-5}) that it is well below the measurement error of physical BRDFs or the error of the approximation. For the rest of this discussion we will ignore the bias and just assume that only the logarithm has been used. Also note that the multiplication by a is applied to BRDF values, for the transformation of the J number of approximation functions a multiplication by $\sqrt[J]{a}$ is necessary.

Parametrization

One of the advantages of the HF method is its generality regarding the used parametrization. Besides choosing a different projection function π for each approximation term, it also allows to use one function for more than one product term.

In the original paper a parametrization with two textures, but with three product terms is used. Using the halfangle vector as defined in equation 21 this parametrization can be written as:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = p(\hat{\omega}_i) q(\hat{h}) p(\hat{\omega}_o) \tag{28}$$

The symmetry by the double usage of one of the textures automatically makes the approximation fulfill the Helmholtz reciprocity criteria, an important criteria that physical BRDF have to fulfill. It basically states that if the direction the light is traveling with is reversed, the BRDF stays unchanged. This can be written as:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = BRDF(\hat{\omega}_o, \hat{\omega}_i) \tag{29}$$

Later in this thesis (chapter 4.1, page 33 and chapter 5.4, page 45) some other parametrization used in conjunction with the HF method will be discussed. For now we will use this parametrization as an example for explaining the algorithm. The generalization of the method is straight forward.

Linear Equation System

The equation we have to approximate for the above parametrization can be written as follows:

$$B\bar{R}DF(\hat{\omega}_i, \hat{\omega}_o) = \bar{p}(\hat{\omega}_i) + \bar{q}(\hat{h}) + \bar{p}(\hat{\omega}_o) \tag{30}$$

We will compute the approximation with a linear equation system, that can be generally written as

$$\hat{b} = A \hat{x} \tag{31}$$

where $\hat{x}$ is a column vector with the values to compute, $\hat{b}$ is a column vector with the constants of the equation, and A is a coefficient matrix.

To approximate the BRDF we will put a number of BRDF samples or rather the logarithmic transformation of the samples into the vector $\hat{b}$. The samples are not required to have any particular structure.

The vector $\hat{x}$ will receive the values of the textures we compute. The texels are unpacked, just as they probably lie in linear memory anyway, and the two textures are put one above the other. We can write this as

$$\hat{x} = \begin{bmatrix} \bar{p} \\ \bar{q} \end{bmatrix} \tag{32}$$

Now the coefficient matrix A defines the mapping, where the BRDF values lie in the textures. Each row of the matrix associates one BRDF value (one component in $\hat{b}$) with a number of texels (in $\hat{x}$), that will later be used to approximate that one BRDF value.

For mapping the BRDF value into the textures we have to do a computation similar to the one that will be done during the BRDF reconstruction to compute the texture coordinates. First the projection function π has to be computed for the two directions of the BRDF value, and then that value has to be mapped with the used texture mapping function (for example parabolic mapping) to a texel position.

Usually the texel position will not be exactly on the integer coordinates of a texel. In the simplest case the texel closest to the exact position could be chosen. This is basically a nearest neighbor algorithm used in image manipulation techniques. To achieve far better results bilinear interpolation should be used. If we call a function that chooses the closest integer that is less or equal to a real number $\lfloor a \rfloor$, then the bilinear weights for an exact position (u, v) are computed as

$$\begin{pmatrix} U \\ V \end{pmatrix} = \begin{pmatrix} \lfloor u \rfloor \\ \lfloor v \rfloor \end{pmatrix}$$
$$\begin{pmatrix} \alpha_u \\ \alpha_v \end{pmatrix} = \begin{pmatrix} u - U \\ v - V \end{pmatrix} \tag{33}$$
$$\begin{pmatrix} \beta_u \\ \beta_v \end{pmatrix} = \begin{pmatrix} 1 - \alpha_u \\ 1 - \alpha_v \end{pmatrix}$$

With these bilinear weights the four texels surrounding the exact position will be summed. For a texture $t(U, V)$ with integer coordinates, we name this interpolated texture function $\bar{t}(u, v)$:

$$\bar{t}(u, v) = \beta_u \beta_v t(U, V) + \alpha_u \beta_v t(U+1, V) + \beta_u \alpha_v t(U, V+1) + \alpha_u \alpha_v t(U+1, V+1) \tag{34}$$

Back to the coefficient matrix, we want to fill. The four products of the bilinear weights in the above equation have to be written at the column of the matrix, that corresponds to the texel in $\hat{x}$ it will be multiplied with.

This has to be done for all three factors of the approximation for each row, so a maximum of 12 columns in one row are filled. Note that due to the double usage of the first texture, some texels might be mapped twice in one row. Both weights should be added in that case. The remaining coefficients in the matrix are set to zero.

If one row of the matrix is now multiplied with the column vector $\hat{x}$, the computation in equation 30 is done with a lookup-function (the texture) and sub-pixel precision (due to bilinear interpolation).

Since all coefficients for the first texture lie on the left side of the matrix and the coefficients of the second texture lie on the right side, we can split the matrix in two and write the equation system of equation 31 with all the data as:

$$[B\bar{R}DF]=[A_p A_q]\begin{bmatrix}\bar{p}\\ \bar{q}\end{bmatrix} \quad (35)$$

Smoothness Constraints

Setting up a solver for the above equation system, will probably not lead to satisfying results. Usually this equation system is under-constrained. Some texels will not be constrained due to the texture mapping used. With parabolic mapping for example texels outside the inner circle of the texture will not be mapped to any direction. This might not be critical since the texels will not be used in rendering anyway, but they might at least have some influence on border texels when the rendering uses a texel interpolation. Another reason for the equation system to be under-constrained is that there might not be enough BRDF samples for all texels to be mapped. For analytical BRDF models the BRDF samples could be chosen to exactly map to the texels, this will be discussed in greater detail in chapter 4.2, page 34. But for measured BRDF data this sampling cannot be applied, only a possibly quite costly resampling of the BRDF might be done.

The HF methods suggests instead to add a LaPlace operator to the equation system. This allows both to bridge over gaps in the data and to control the smoothness of the approximation result. Ideally the LaPlace operator would be applied to the BRDF data itself, but for less computation effort it can be also applied to the textures instead.

The LaPlace operator creates a relationship between one texel and its four directly adjacent texels (in border situations only three or two texels):

0	-1	0
-1	**4**	-1
0	-1	0

Figure 7: LaPlace operator

Note that the weight of the center texel, the hotspot, is decreased to two or three, if only two or three adjacent texels exist due to a border situation.

The LaPlace operator is added to the equation system in the form of equations that set the result of the operator to zero. For non-border texels of a texture $t(U,V)$ this can be written as:

$$4t(U,V)-t(U-1,V)-t(U+1,V)-t(U,V-1)-t(U,V+1)=0 \quad (36)$$

The coefficients of the texels in the equation can be inserted into the matrix in the same way as above. If the matrices L_p and L_q contain the application of the LaPlace operator to the textures $\bar{p}$ and $\bar{q}$, we can create a new equation system based on equation 35:

$$\begin{bmatrix} \bar{BRDF} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} A_p & A_q \\ \lambda L_p & 0 \\ 0 & \lambda L_q \end{bmatrix} \begin{bmatrix} \bar{p} \\ \bar{q} \end{bmatrix} \quad (37)$$

The lambda λ in this equation is a scalar value used to control the weight of the LaPlace operator and thereby the smoothness of the solution. A λ of zero leads to the original equation system, only with additional useless constraints. Using a larger λ, the LaPlace equations will have more influence during the iterative computation of the solution, and also the error of the solution will be greater.

Starting Solution

To compute the above equation system efficiently the result vector should be filled with values as close as possible to the desired result. Just filling the result vectors with zero is clearly less than optimal.

A better starting solution can be found by averaging the projected BRDF values. In detail this works for each texture as follows.

1. Fill the texture t with zero
2. Create and fill with zero a matrix W with the same dimensions as the texture
3. For all available BRDF samples do:
    4. Compute the projected texel position (u,v) of the BRDF sample
    5. Add the BRDF value to the texture at the four texels around (u,v) , multiplied by the bilinear weight as in the coefficient matrix
    6. Add the bilinear weight at the four texels around (u,v) to W
7. For all texels (U,V) in the texture do:
    8. If $W(U,V)>0$, set $t(U,V)$ to $\frac{t(U,V)}{W(U,V)J}$ with J being the number of approximation factors (e.g. three for the standard parametrization)
    9. If $W(U,V)=0$, search for texels around (U,V) with $W(U,V)>0$ and compute average over their values. Searching in the eight directions (cross and diagonal) should be enough. Each found texel $(U+x,V+y)$ should be weighed by $\frac{1}{\|(x,y)\|}$ when computing the average.

A starting solution computed with this algorithm might even be visually similar to the final result, if the BRDF is highly separable with the used parametrization.

Another way to find a good starting solution is to use the result of a previous separation run with a smaller texture size. If the algorithm is set up with textures of twice the size of a previous run the results can be easily linearly interpolated to find a new starting solution. To gain significant computation advantages with this multi-resolution method, it is advisable to use a lower sampling frequency for the lower resolution textures.

Additionally if the algorithm is set up with a different parametrization than the one proposed in the HF paper, for example with the Gram-Schmidt Halfangle-Difference parametrization (see chapter 3.1.1, page 21), then the results of the Normalized Decomposition algorithm could be used as starting solution for the HF method. The solver is basically used to improve the result of the ND method, since it does not guarantee to find an (at least locally) optimal solution.

Iterative Computation

Having set up the necessary matrix and vectors, the equation system can be solved with an iterative method. For the HF method the Quasi-Minimal Residual (QMR) method has been used. See chapter 2.5 Iterative Solvers (page 15) for a brief introduction to iterative methods.

Note that the coefficient matrix is quite large, with M number of BRDF samples and N texels per textures the matrix (without the LaPlace operator) has a size of $M \times 2N$. But since it is quite sparse, only a maximum of 12 values per row are not zero, it can easily be represented as a compressed row matrix. If the matrix is still to big to fit into memory, it might not even need to be stored explicitly. The iterative solver only needs to do a matrix-vector multiplication with the data. Therefore it could be computed on the fly.

Iterative solvers usually do not work with three component color values. Therefore the solver has to be set up for three separate passes, one for each color component. For each pass both the BRDF value vector and the texture data have to be switched to a different color channel, the coefficient matrix can be reused.

3.1.3 Evaluation of Separation Methods

The three general BRDF separation techniques described above, have their individual advantages and disadvantages.

Both SVD and ND require a regularly sampled BRDF and are therefore less suitable for measured BRDFs than the HF. It can work with measured data without a costly resampling. The interpolation required for resampling is done by the HF (implicitly) inside the approximation (with the LaPlace operator), possibly leading to less error.

The advantages of the ND lie in the possibility to create quite large textures (128 × 128 and above) with acceptable computation efforts. Especially very sharp peaks in BRDFs tend to fill only a few texels on small textures and are poor to interpolate without looking blocky.

The biggest advantage of the HF is the flexibility and thereby the possibility to apply completely different parametrizations for different BRDFs and reconstruction hardware. The computation effort of the HF method is usually not small, but it can be controlled in a wide range, by choosing the number of BRDF samples and texture size. Therefore it is possible to first compute a rough factorization before creating high quality textures.

3.2 Environment Mapping

Another approach to the topic of realistic simulation of lighting and materials is via environment mapping.

3.2.1 Mirror-like Environment Maps

A traditional environment map contains an image with the surrounding environment of an object. A photograph (made with a high focal length) of the surroundings mirrored on a sphere with a perfectly reflecting mirror surface can be used as spherical environment map. The texture mapping is similar to the hemispherical mapping as in figure 6, page 19. Both parabolic and cubic texture mapping are also techniques for environment mapping.

Using an environment map as texture of an object makes the object look like chrome or other almost perfect mirrors. Note that applying the texture to an object assumes that the environment is far away, because the reflections ignore the local difference of the reflection from one position to another position on the surface of the object.

Several techniques are available to use environment maps to simulate other materials than perfect mirrors. A very simple one is to filter the map with a constant color, allowing the approximation of colored mirrors, for example with a golden color.

3.2.2 Glossy Environment Maps

To simulate reflections of non-mirror like materials a more complicated filtering can be applied to an environment map. These methods are based on the assumption that the pixels of the environment map represent the amount of light coming from a certain direction towards the object. Each pixel can be thought of as a directional light source with a color.

These light sources can be used for a lighting computation (for example with a BRDF) and the result of the lighting computation can be stored again in an environment map. A paper by Kautz, Vázquez, Heidrich and Seidel [Kautz2000] gives an excellent survey

about these methods. Therefore we will only give a brief summary of their notation and the conclusions.

The BRDF lighting computation as in equation 12 uses parameters relative to the local surface frame, that is the coordinate system defined by the normal, tangent and a vector orthogonal to both. Environment maps used as light sources however use direction vectors in world coordinates. To describe BRDF based lighting with vectors in world space, we can write:

$$L(\hat{v},\hat{n},\hat{t})=\int_{\Omega} BRDF(\hat{\omega}(\hat{l},\hat{n},\hat{t}),\hat{\omega}(\hat{v},\hat{n},\hat{t}))\,L_i(\hat{l})(\hat{n}\cdot\hat{l})\,d\hat{l} \quad (38)$$

Here $\hat{v}$ is the viewing direction, $\hat{l}$ is the direction of incoming light, $\hat{n}$ and $\hat{t}$ the surface normal and tangent, all in world space. The function $\hat{\omega}(\hat{l},\hat{n},\hat{t})$ computes the spherical coordinates for the light direction relative to the coordinate frame defined by $\{\hat{n},\hat{t},\hat{n}\times\hat{t}\}$. This is what we called $\hat{\omega}_i$ until now. $\hat{\omega}_o$ is analogue for the viewing direction. The light sources based on the environment map will be used to define the function $L_i(\hat{l})$.

Heidrich and Seidel proposed to use the Phong BRDF to compute environment maps [Heidrich1999]. Using a diffuse BRDF, that is a BRDF with a constant color value k_d , a lighting equation that drops the dependency on the viewing direction and tangent can be written as:

$$L(\hat{n})=\int_{\Omega} k_d\,L_i(\hat{l})(\hat{n}\cdot\hat{l})\,d\hat{l} \quad (39)$$

Since the lighting computation above only varies with the normal of the surface, the function can be pre-computed into an environment map, applied to an object with texture coordinates generated from the normal.

The specular part of a Phong BRDF can be used for a lighting computation that will lead to a function depending on the normal and the viewer direction. It can be reparametrized to use the reflected viewer direction $\hat{r}_v$, which can be computed as:

$$\hat{r}_v=2(\hat{n}\cdot\hat{v})\hat{n}-\hat{v} \quad (40)$$

With the reflected viewer direction the lighting computation only depends on one parameter:

$$L(\hat{r}_v)=\int_{\Omega} k_s(\hat{r}_v\cdot\hat{l})^N\,L_i(\hat{l})\,d\hat{l} \quad (41)$$

Here k_s is the specular color and N controls the shininess of the material. Note that the resulting environment map from this computation looks like a blurred version of the original environment map, because the function curve of the specular Phong BRDF is similar to the filter kernels of blurring algorithms used for image manipulation.

The two environment maps generated for diffuse and specular parts of the Phong model can be added together in real-time rendering to emulate the full Phong model.

To approximate other BRDFs than the Phong BRDF two techniques exist. One by Kautz and McCool [Kautz2000II] requires not only an isotropic BRDF, it also has to be radially symmetric about the reflected viewer direction.

The basic idea is to filter the environment map similar to the blurring of the specular Phong method. Instead of using a filter based on the function curve of the Phong specular BRDF, they use a more general filter kernel, that is also symmetric around the reflected viewer direction. Approximating the BRDF with a two-dimensional function $p(\hat{l}\cdot\hat{r}_{v,}\hat{n}\cdot\hat{r}_v)$, the lighting computation can be approximated with

$$L(\hat{r}_{v,}\hat{n}\cdot\hat{r}_v)=(\hat{n}\cdot\hat{r}_v)\int_\Omega p(\hat{l}\cdot\hat{r}_{v,}\hat{n}\cdot\hat{r}_v)L_i(\hat{l})d\hat{l} \tag{42}$$

This is a three-dimensional function, requiring 3D textures. Another approximation they suggest leads to 2D textures, that will be scaled by a constant factor during rendering. This uses a BRDF approximated by $F(\hat{n}\cdot\hat{r}_v)p(\hat{l}\cdot\hat{r}_v)$ and a lighting computation of

$$L(\hat{r}_{v,}\hat{n}\cdot\hat{r}_v)=(\hat{n}\cdot\hat{r}_v)F(\hat{n}\cdot\hat{r}_v)\int_\Omega p(\hat{l}\cdot\hat{r}_v)L_i(\hat{l})d\hat{l} \tag{43}$$

Another method for lighting with isotropic and radially symmetric BRDFs was proposed by Cabral et al. [Cabral1999]. They actually use a four dimensional lighting computation similar to the above approximation:

$$L(\hat{v},\hat{n})=\int_\Omega p(\hat{l}\cdot\hat{r}_{v,}\hat{n}\cdot\hat{r}_v)L_i(\hat{l})(\hat{n}\cdot\hat{l})d\hat{l} \tag{44}$$

The important change is that the function is only sparsely sampled in $\hat{v}$ and therefore has only a few exact images for certain viewer directions. To generate the environment maps for other directions, image warping is used. Therefore high end graphics hardware is required to achieve interactive frame rates with this method.

Only one technique to approximate anisotropic BRDFs in global illumination situations has been developed so far. Generally this computation leads to a five dimensional function, but Kautz et al. [Kautz2000] simplified the Banks anisotropic BRDF, so that the

lighting computation only depends on three parameters and can be stored in a 3D texture.

4 Improvements to the Homomorphic Factorization

As mentioned earlier, for writing this thesis the HF technique has been implemented and some improvements have been added. A general notation of parametrizations is introduced and methods to select BRDF samples are discussed. Then two algorithms to improve the visual appearance of reconstructed textures are presented.

4.1 Flexible Parametrizations

The HF method is a very flexible factorization technique, it is not necessary to restrict the parametrization to the one used to present the method. McCool et al. discuss several parametrizations for the technique.

We have tested the method with the incident view (IV) parametrization (see equation 20), the Gram-Schmidt halfangle difference (GSHD) parametrization (see equation 23) and the original HF parametrization. Although the first two were introduced for the SVD/ND approach by Kautz, there is no theoretical reason for not using them with the HF method. The HF might even be used to improve the results of a ND, by using the ND to compute the starting solution for the HF. However a big advantage of the HF is lost with this, as the ND requires a regularly sampled BRDF, and therefore a costly resampling step might be required.

A Cook-Torrance BRDF [Cook1981] with a parameter setup for simulating gold, showed almost equal visual quality of the GSHD and the HF parametrization. In this situation the GSHD separation can be used at least on systems with few texture units, as it only requires two texture lookups. For multiple light sources this advantage is less important, because the HF parametrization needs only two texture lookups per additional light source after the application of the first one.

For other BRDFs, like the Poulin-Fournier BRDF [Poulin1990], the GSHD parametrization produces only mediocre results. The IV parametrization did not produce any useful results for either BRDF.

To be able to easily try out different parametrizations, which was especially important for the global illumination approximation in chapter 5 (page 43), our implementation uses the following generalized parametrization:

A parametrization has J factors and T textures. The textures are called t_k with $k \in [1, T]$. Each factor j is associated with a texture index k by a function $k(j)$, and it has a direction projection function π_j. We can write this similar to equation 24:

$$BRDF(\hat{\omega}_i, \hat{\omega}_o) = \prod_{j=1}^{J} t_{k(j)}(\pi_j(\hat{\omega}_i, \hat{\omega}_o)) \qquad (45)$$

The original HF parametrization uses $J=3, T=2, k(1)=1, k(2)=2, k(3)=1$. The textures formerly called p and q are now called $t_1=p, t_2=q$.

4.2 Sample Selection

Another important issue concerning the quality of the factorization result, is the selection of BRDF samples to use for the computation.

For measured BRDFs the choice of samples is self-evident. Usually the number of samples is quite limited and all of them can be used for the factorization. Measurements for isotropic BRDFs are often done with one restricted angle (like ϕ_i) as they only need three degrees of freedom. For a better distribution of samples, the measurements can be replicated multiple times with an offset added to the ϕ angles.

For analytically modeled BRDFs the situation is different. The samples can be taken for basically any combination of $\hat{\omega}_i$ and $\hat{\omega}_o$. The concept of enumerating all the combinations of directions to sample a BRDF can be called a “sample iterator”.

The simplest idea to iterate over sampling directions is by repeatedly adding a constant angular step to the two angles of a direction in spherical coordinates. This results basically in a latitude/longitude grid, just as on a globe. Different angular steps for the two angles should be chosen, as the θ angle only has a range of $[0,\frac{\pi}{2}]$ on a hemisphere, while the ϕ angle spans $[0,2\pi[$. Iterating over all directions of the latitude/longitude grid can be called a “direction iterator”.

To use this concept for a sample iterator we need two direction iterators, one will iterate over all directions of $\hat{\omega}_i$ and the second will iterate at any step of $\hat{\omega}_i$ over all directions of $\hat{\omega}_o$. This leads to a number of samples that is the product of the steps of each direction iterator, which in turn is a product of the latitude and longitude steps of the iterator. Calling the latitude steps A and the longitude steps O the total number of samples is $A \cdot O \cdot A \cdot O$.

Using direction iterators with constant angular steps is far from optimal. One problem is the vast differences in sampling density. At the pole of the hemisphere, iterating over ϕ always leads to the same direction, as the longitudes converge there. At the equator of the hemisphere the samples will have their maximum distance. Another problem is that the chosen texture mapping (whether parabolic or hemispherical) will also show significant differences in the density of the projected sample positions on the texture.

To overcome at least the second problem and significantly reduce the first one as well, a different direction iterator can be used. It is based on the reverse projection of the texels in a texture to the corresponding directions. This iterator will basically iterate over

all texels of an imaginary texture, discard all texels that cannot be mapped to a direction (for example outside the inner circle of a parabolic map), and then compute the corresponding direction to that texel. Using parabolic mapping this can be written for texture coordinates (x, y) that are in a coordinate system relative to the center of the texture as:

$$\hat{d} = \| \begin{pmatrix} x \\ y \\ \frac{1}{2} - \frac{1}{2}(x^2 + y^2) \end{pmatrix} \| \quad (46)$$

The direction can then be converted to spherical coordinates. Two of these directions together define a sample iterator for an incident view parametrization.

This sample iterator is also suitable for the original HF parametrization. While iterating over all $\hat{\omega}_i$ and $\hat{\omega}_o$, all texels in the second texture indexed by $\hat{h}$ will be touched as well: When $\hat{\omega}_i$ and $\hat{\omega}_o$ are equal, the half vector will be equal to both.

This iterator is not ideal for the GSHD parametrization, but can be easily altered for this purpose. $\hat{h}$ and $\hat{d}$ can iterate in the same way over all mapped texel directions. For each pair of $\hat{h}$ and $\hat{d}$ an incident view pair can be computed.

Some additional sample and direction iterators will be introduced for the lighting computation factorization in chapter 5.6 (page 49).

4.3 Texture Precision

Tests with a Cook-Torrance BRDF [Cook1981] showed that the factorization itself produced only a small numerical error, but having converted the exact approximation data to 8-bit per channel textures showed a significant error. In numbers: the average relative error between the BRDF and its reconstruction from floating-point values is 0.094, the error of the BRDF to a reconstruction from the textures is 0.240. The RMS error for the same differences is 0.294 compared to 0.862. These measurements show the importance of the conversion step to integer textures.

In the original method it is suggested to normalize the floating-point textures, i.e. to scale them with the reciprocal of their maximum value, before converting them to integers. To correct for the normalization, a constant color value is used, that will be rendered as the vertex color. The normalization is problematic for BRDFs with high specular peaks, as the lower BRDF values lose precision. To circumvent the problem it is suggested to clamp the peak of the BRDF before separation and add the clamping difference as an additional texture in the reconstruction.

The experiments with the Cook-Torrance BRDF, which has a high specular peak, have lead to a different approach that does not need a fourth texture and should also enhance the visual quality of BRDFs without specular peaks. The clamping of the specular peak before the separation should be done anyway, because it can disturb the approximation algorithm. However adding another texture is not always visually important.

Additionally to the normalization of the floating-point textures, we suggest a prescaling of the textures with interactively determined factors optimized for visual quality.

The rendering of one point of a BRDF material can be written as:

$$c(\hat{n}\cdot\hat{l})\,p_i\,q_h\,p_o\,z \tag{47}$$

where c is the normalization correction color, $\hat{n}\cdot\hat{l}$ is the diffuse lighting term, p_i , q_h and p_o are the three textures evaluated at the directions $\hat{\omega}_i$, $\hat{\omega}_o$ and $\hat{h}$, z is an additional constant scale factor. The computation is done in the specified order. c will be implemented as the diffuse material color, maybe multiplied with the color of a non-white light source and a light source attenuation factor. The dot product will be computed with diffuse lighting per vertex. Assuming a hardware system with enough multitexture stages, the three textures will be multiplied together. In the first stage the texture will be multiplied with the result of the per-vertex lighting. And finally the factor z is used to scale the result to a greater range.

Note that all the previous steps have only a range of $[0,1]$, all intermediate results will be clamped to this range. With each factor the result can only get smaller or stay the same, so only in one combination at all the result could be one, i.e. white. This is not realistic. The biggest problem is that usually c will be greater than one, and the clamping reduces the overall intensity of the material. To correct this, the original correction color is split into a product of the used correction color c and the scale factor z. The latter one can be implemented as a scale factor in the texture pipeline that most multitexture capable hardware offers (for example via the texture environment combiner OpenGL extension, that has become part of OpenGL in version 1.3). Then z can only have the value 1, 2 or 4. Larger factors can be implemented by adding another factor similar to z after the second or even the first texture stage, with the risk of loosing intensity due to clamping of the intermediate result. Using the register combiner OpenGL extension (NVIDIA only) or DirectX 8 pixel shaders, factors like 8, 16 or above can be achieved by adding the texture stage result with itself multiple times inside the texturing unit. At least for factors greater than 8 precision becomes a problem. With a factor of 16 only

4 bits of a 8 bit color channel will survive the computation. This will lead to contouring artifacts. Later an alpha blending technique to enhance precision will be introduced.

4.3.1 Prescaling of Textures

Back to the prescaling algorithm: It uses scale factors to both scale the correction color and the textures. The scale factors can be chosen interactively to assure maximum visual quality instead of a numerical measurement. The two extremes between the scale factors have to provide a good compromise are to clamp the highlights of the BRDF or to loose the accuracy of the base color. It is hard to find some measurement that can describe an optimal compromise, therefore the interactive evaluation is proposed.

Calling the prescale factors s_i for $i \in \{0,1,2\}$, the rendering computation can be written

$$(s_0 c)(\hat{n} \cdot \hat{l})(s_1 p_i)(s_2 q_h)(s_1 p_o) z \quad (48)$$

To not falsify the result, the scale factors should not change the result of the computation (assuming there are no precision restrictions of the rendering hardware). This leads to the requirement $s_0 s_1^2 s_2 = 1$. To allow an interactive selection of the scale factors fulfilling this requirement, another set of scale factor weights w_i can be used.

Since our implementation works with a number factorization parametrizations, not only the original HF parametrization, the algorithm for computing s_i out of w_i is not that simple.

Assume the generalized parametrization described above, with J factors and T textures. The textures are called t_k with $k \in [1, T]$. The index i of s_i and w_i is in the range $[0, T]$, so there is one more scale factor than there are textures. A function $f(k)$ describes the number of factors that use the texture t_k. The original HF parametrization uses $f(1)=2, f(2)=1$.

To compute a set of s_i from a given set of w_i, the following steps are necessary:

1. Set $a = w_0 \prod_{i=1}^{T} f(i) w_i$
2. Set $s_0 = \frac{w_0}{\sqrt[T+1]{a}}$
3. For each $i \in [1, T]$ set $s_i = \frac{w_i}{\sqrt[(T+1) f(i)]{a}}$

The w_i can now be set in some kind of graphical user interface to interactively compute the prescaling and evaluate the effect. For convenience, it is better to set $\log(w_i)$ through user input, as a logarithmic scale is more intuitive.

4.3.2 Alpha Blending Contouring Prevention

As mentioned in the introduction to this section, using a large scale factor z (in equation 47) will lead to contouring. This is caused by a lack of precision in the lower bit ranges of the frame buffer, and is also known from using 16 bit frame buffers where each color component has only 5 or 6 bit precision. Already a scale factor z of 4 shows some contouring artifacts, higher factors display significantly stronger contouring. Factors greater than 16 can be generally considered as unsuitable for frame buffers with 8 bit components, even with the compensation described here.

The general idea to reduce the effect of contouring is to use the alpha channel of a texture as additional computation factor, that can scale down the color value *after* it has been bit shifted through z to reduce the artifacts.

For precision enhancement an alpha channel can be added to the textures and the color channels are scaled appropriately, so that the new textures multiplied with the alpha channel are equal to the old textures. This can be written, based on equation 47 as:

$$c(\hat{n}\cdot\hat{l})\, p_i\, q_h\, p_o\, z = c(\hat{n}\cdot\hat{l})(\bar{p}_i\, \alpha_{p_i})(\bar{q}_h\, \alpha_{q_h})(\bar{p}_o\, \alpha_{p_o})\, z \qquad (49)$$

Now to get an improvement with this, apply a product of the alpha values at the end of the pipeline

$$c(\hat{n}\cdot\hat{l})\, \bar{p}_i\, \bar{q}_h\, \bar{p}_o\, z\, (\alpha_{p_i}\alpha_{q_h}\alpha_{p_o}) \qquad (50)$$

During rendering the alpha channels are multiplied parallel to the color textures, and then the alpha product is applied to the color value with alpha blending. Alternatively on hardware with programmable pixel pipeline the multiplication with the alpha channel can be done in register combiners (NVIDIA OpenGL) or pixel shaders (DirectX 8). Then the technique might not have any performance loss at all, except through the small memory transfer of the alpha channel of course.

Alpha Channel Computation

For now we will use the simplification that all color values are treated as scalar values, not three component vectors.

As mentioned earlier the main purpose of the final scaling by z is to increase the range of the computed color, for example to be able to reach the color white for highlights. The alpha factors applied after z have to make sure, that they do not scale a highlight down again to some gray value. To do this we have to look at the clamping that the texturing unit does.

Basically any intermediate computation result will be clamped to $[0,1]$. Since most of the values in the above equation are already in this range no product of them can cause clamping only the result after the application of z might be clamped. Using a function called $clamp(a)$ that will clamp its parameter to the range $[0,1]$, we can write equation 50 as:

$$clamp(c(\hat{n}\cdot\hat{l})\bar{p}_i\bar{q}_h\bar{p}_o z)(\alpha_{p_i}\alpha_{q_h}\alpha_{p_o}) \tag{51}$$

To check whether the usage of the alpha channel falsifies the results, we can check the following equation:

$$clamp(c(\hat{n}\cdot\hat{l})(\bar{p}_i\alpha_{p_i})(\bar{q}_h\alpha_{q_h})(\bar{p}_o\alpha_{p_o})z)\stackrel{?}{=}clamp(c(\hat{n}\cdot\hat{l})\bar{p}_i\bar{q}_h\bar{p}_o z)(\alpha_{p_i}\alpha_{q_h}\alpha_{p_o}) \tag{52}$$

Only if the clamping on both sides does not alter the value, the equation is fulfilled. Consider the right hand side: the clamping will not have any effect, if:

$$c(\hat{n}\cdot\hat{l})\bar{p}_i\bar{q}_h\bar{p}_o z\leq 1 \tag{53}$$

Since c and z are positive constants, we can write

$$(\hat{n}\cdot\hat{l})\bar{p}_i\bar{q}_h\bar{p}_o\leq\frac{1}{cz} \tag{54}$$

Now using the knowledge about the range $[0,1]$ of all four factors on the left side, we know that the condition will be fulfilled, if

$$(\hat{n}\cdot\hat{l})\leq\frac{1}{cz}\ \vee\ \bar{p}_i\leq\frac{1}{cz}\ \vee\ \bar{q}_h\leq\frac{1}{cz}\ \vee\ \bar{p}_o\leq\frac{1}{cz} \tag{55}$$

The first of the four relations is not interesting to us, but the other three are important. If at least one of them is fulfilled, then on the right hand side of equation 52 no clamping will occur. There are of course other situations without clamping, but we do not know enough about these to be able to exploit them. For the left hand side of equation 52, we can write the same conditions for the product with the alpha value, or since the product is equal to the original textures, we can write

$$p_i \leq \frac{1}{c\,z} \;\vee\; q_h \leq \frac{1}{c\,z} \;\vee\; p_o \leq \frac{1}{c\,z} \tag{56}$$

With the conditions of equations 55 and 56 we can set up simple rules, to compute the alpha value α_f for any factor $f \in \{p_{i,} q_{h,} p_o\}$

1. If $f \leq \frac{1}{c\,z}$, set $\bar{f} = \frac{1}{c\,z}$ and $\alpha_f = f\,c\,z$
2. Otherwise set $\bar{f} = f$ and $\alpha_f = 1$

Rule 1 tries to set $\bar{f}$ to the maximum value without causing clamping, so that as much down-scaling as possible can be done with the alpha channel, because the alpha scaling result will not suffer from the same precision problems as the color channel.

Rule 2 is a case, where we cannot be certain that no clamping will occur. So we set the alpha channel to one and have basically no advantages over the calculation without the alpha. In practice it is quite common, at least for the problematic BRDFs with high specular peaks, that at least one of the three factors is below the limit of rule 1.

To implement the two rules, we can connect them to one simpler one:

$$\begin{aligned} \alpha_f &= clamp(f\,c\,z) \\ \bar{f} &= \frac{f}{\alpha_f} \end{aligned} \tag{57}$$

The advantage of this computation is not obvious, as the above if-statement has only moved into the clamping function, but it is useful for dropping the one assumption we set up at the beginning, that colors are scalar values.

Remember that both f and c are actually three component vectors. We can only have one alpha value for all three components. Equation 57 can basically compute three alpha values for the three color components. We can then choose the most critical one, the one with the highest chance to invalidate one of the less-than conditions above. This is obviously the greatest of the three alpha values. With $\hat{f}$ and $\hat{c}$ as color vectors now, we can write the alpha computation as:

$$\begin{aligned} \hat{a} &= \hat{f}\,\hat{c}\,z \\ \alpha_f &= clamp(max(a_{r,}\,a_{g,}\,a_b)) \\ \hat{\bar{f}} &= \frac{1}{\alpha_f}\hat{f} \end{aligned} \tag{58}$$

Note that the vectors here are multiplied as separate components, not with a dot product.

Evaluation

The method can only be considered useful, if it really improves the results, and not under some special circumstances will lead to worse results than without using the alpha channel.

It is hard to prove this exactly, as it is more a problem of computation precision of certain hardware than of mathematics. We can at least look at the possible precision under certain conditions. However the used hardware might be not reach this precision, for example alpha blending for a 16 bit frame buffer is probably done with only 5 bit precision for each color channel.

Nevertheless consider the following computation: We compute equation 50 with the simplification $c=\hat{n}\cdot\hat{l}=\bar{q}_h=\bar{p}_o=\alpha_{q_h}=\alpha_{p_o}=1$, so we only have to consider p_i . We also think of the colors as scalar values again. The whole computation now becomes

$$\bar{p}_i z \alpha_{p_i} \tag{59}$$

We choose $z=8$ and assume that $p_i \leq \frac{1}{8}$, so that the alpha channel can have a value other than one. To be able to look at the precision, we will assume, we have 8 bit fixed-point arithmetic hardware. This means that a value of 1 will be represented with $255_{10}=11111111_2$, the index indicates the base of the number system. For conversion in or from the fixed-point arithmetic, the values have to be multiplied or divided respectively by 255. In the following illustration the 8 bits are drawn as boxes, with the lowest bit on the right hand side of the box.

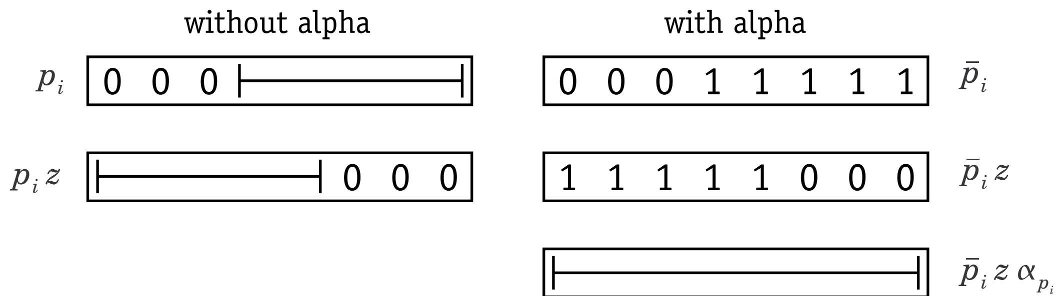


Figure 8: Alpha computation precision

First look at the left hand side of the illustration. Since we know that $p_i \leq \frac{1}{8}$, it can only fill the lower 5 bits. A value of $\frac{1}{8}$ is represented with $(255\cdot\frac{1}{8})_{10} \approx 11111_2$. The multiplication with z will shift the result 3 bits upward. Note that if we did not restrict the range of p_i , some bits might have been shifted beyond the highest bit, and clamping

would have to be done. It can be seen that the result contains only 5 data bits. The lack of the other three causes the contouring we are trying to prevent.

Now look at the computation with the alpha channel. $\bar{p}_i$ has been set according to the algorithm to the highest value, it is allowed to take: $\frac{1}{8}$. This is shifted with z and then multiplied with the alpha value. Depending on the alpha value the whole range of the 8 bits can be used. The smallest possible $\alpha_{p_i}=(255\cdot\frac{1}{255})_{10}=1_2$, will cause a result of 1_2 or $\frac{1}{255}$. The greatest value $\alpha_{p_i}=(255\cdot 1)_{10}=11111111_2$ will not alter the result of $\bar{p}_i z$ and then no precision advantage over the the other method has been achieved. Finally a value somewhere between the extremes, maybe $\alpha_{p_i}=(255\cdot\frac{1}{4})_{10}\approx 111111_2$ will result in 111101_2, showing that the three least significant bits are used.

This little hardware simulation game is far from a proof, but can be used to test the validity of the approach.

Finally a visual example is given with the following two images. They were generated with a separated Cook-Torrance BRDF. A z of 8 has been used to show the effect.

Figure 9: Contouring of separated gold BRDF (left: without alpha, right: with alpha)

The error measurements for the two images, show the effect of the alpha channel method as well. The average relative error is without the alpha method 0.382, with alpha channel 0.269. The differences for the RMS error are less drastic: 0.865 to 0.863.

As a conclusion, it can be said, that if a BRDF has a high dynamic range and the computation effort for doing the alpha blending step is acceptable, using the alpha channel will produce noticeably higher visual quality.

5 Factorization of Lighting Computation

The algorithm of the Homomorphic Factorization method (see chapter 3.1.2 Homomorphic Factorization (HF), page 22) has been adapted to approximate the lighting of a number of light sources or a whole global illumination scenario.

This chapter begins with the motivation for developing this method. After the general approach, the necessary changes to the HF method and problems will be introduced. In particular possible parametrizations, changes in texture mapping and in the sampling of functions are discussed. Then some factorization results are presented with an error analysis. Finally the method will be compared to similar techniques, for the most part these are environment map prefiltering techniques.

5.1 Motivation

BRDF-based lighting allows simulation of arbitrary materials with high visual quality. Approximating the BRDF of a material for use in real-time rendering can recreate BRDF-based lighting for a very limited number of light sources. The textures used for approximating the BRDF need to applied once for every single light source.

A simple example: An approximation with two textures in a scene with only four point or directional light sources requires eight texture applications to each model. This impacts rendering performance through the texture application, the multiple rendering passes needed and the dynamic texture coordinate computations for each texture.

In an ideal setting, the whole scene is rendered with BRDF-lit models. The amount of rendering computation required for the BRDF textures quickly becomes the bottleneck of the rendering system.

Most of the objects in a scene do not move, and the dynamic computation of the BRDF for every light source is not necessary. However the lighting is not fixed for motionless objects. The direction of the viewer towards an object plays a significant role in the computation of reflections and thereby the overall appearance of an object.

It is therefore desirable to fully approximate the lighting of any number of light sources with a fixed number of textures. The method introduced here allows a visually sufficient approximation of the lighting computation with two to four textures, depending on the complexity of the used BRDF.

The method has no theoretical restriction for the number of light sources used. Therefore applying it just to accelerate the computation of a small number of point or directional light sources does not exploit its full potential. It can be used to approximate a global illumination scenario, where a certain amount of light comes from every direction in space. A possible description for this light situation is an environment map (see chapter 3.2.2 Glossy Environment Maps, page 29).

Figure 10: Golden teapot with complex lighting

As an example of the technique, an image created with a factorization of a gold material and lighting based on the environment map “Loch” by Jeff Heath is shown.

5.2 Approach

The idea behind the application of the HF method to the lighting computation, is based on the functional similarities between the lighting computation and the BRDF. Most importantly they both can be written as depending on two directional parameters and therefore having four degrees of freedom.

To show this, we start with the lighting computation in world space (introduced in equation 38, page 30):

$$L(\hat{v},\hat{n},\hat{t})=\int_{\Omega} BRDF(\hat{\omega}(\hat{l},\hat{n},\hat{t}),\hat{\omega}(\hat{v},\hat{n},\hat{t}))L_i(\hat{l})(\hat{n}\cdot\hat{l})d\hat{l} \quad (60)$$

where $\hat{v}$ and $\hat{l}$ are the viewing and incoming light direction, $\hat{n}$ and $\hat{t}$ are the surface normal and tangent, all in world space.

First we restrict the BRDF to be an isotropic BRDF, and therefore can drop the dependency on the tangent. For the local surface frame of the BRDF we still need a tangent, but it can be any vector orthogonal to $\hat{n}$. It only has to be assured that the same vector is used for both BRDF parameters. Therefore we will introduce a function $\bar{t}(\hat{n})$, that will compute a valid surface tangent orthogonal to $\hat{n}$. The lighting computation becomes:

$$L(\hat{v},\hat{n})=\int_{\Omega} BRDF(\hat{\omega}(\hat{l},\hat{n},\bar{t}(\hat{n})),\hat{\omega}(\hat{v},\hat{n},\bar{t}(\hat{n})))L_i(\hat{l})(\hat{n}\cdot\hat{l})d\hat{l} \quad (61)$$

Both $\hat{v}$ and $\hat{n}$ are directional vectors that can be represented in spherical coordinates, just as the two BRDF parameters. So the function $L(\hat{v},\hat{n})$ is suitable for the factorization

method in place of the BRDF. For the rest of this thesis the term "lighting computation" (LC) refers to this function.

5.3 Differences to BRDF Factorization

The BRDF and the LC have the most important aspect for the factorization in common, the number and type of parameters, but some other characteristics are different.

Most importantly the two parameters are not directions on a hemisphere but on a whole sphere. The viewing direction is a parameter for both the BRDF and the LC, but for the BRDF it is specified relative to the local surface frame, for the LC it is any direction in world space. This causes a small problem: There are parameter combinations of $\hat{v}$ and $\hat{n}$, where the viewing direction looks at the back face of the surface. In that case the function can be sent to zero, or the back face could be computed with a reversed normal.

It is possible to use a viewing direction relative to the local surface frame for the LC as well and thereby avoid the back facing problem, however this is not recommended due to some practical problems. One is the dependence between the two parameters. Since the normal is a crucial part in defining the local surface frame, the viewing direction would be dependent on the normal. Another problem is that the normal would still be a direction on a whole sphere, while the viewing direction would be on a hemisphere. Therefore different texture mappings (discussed below) are suitable for each parameter.

A difference between the BRDF and the LC that can be used to our advantage is the range of the computed values. Although the LC can have values greater than one, the result of the LC will be directly used for display, not weighed by a $\hat{n}\cdot\hat{l}$ factor like the BRDF. Therefore the LC range can be clamped to $[0,1]$ before factorization, and problematic approximation range issues as with highly specular BRDFs are unlikely.

Finally the most problematic difference between the BRDF and the LC is the lower grade of symmetries in the function. From the knowledge about material properties, like shadowing or microfacet distribution, assumptions about at least partial symmetries in the BRDF (for example around the half vector) can be deduced. The structure of the LC is more complicated: The incoming light function $L_i(\hat{l})$ does not have fixed properties, and therefore assumptions about the overall symmetry of the LC are harder to find. The parametrization of the approximation has to try to compensate this.

5.4 Parametrization

A good parametrization is crucial for a high quality factorization. Two simple parametrizations can be deduced from environment mapping prefiltering techniques. Based on this a new parametrization will be defined, and some additions are made.

In chapter 3.2.2 Glossy Environment Maps (page 29) the techniques to prefilter an environment map with the diffuse and specular part of the Phong BRDF is described (equations 39 and 41). Both methods can be written as a specialization of the factorization method. The factorization can be set up with one texture and one approximation factor. Of course it does not need to compute the equation system for only one factor. The filling of the texture with the starting solution as described in chapter 3.1.2 (page 27) already creates an optimal results.

The used parametrizations are interesting for further consideration. For the diffuse Phong BRDF the normal vector is used as parameter. The computation of a texture t_1 from the LC can simply be written as:

$$L(\hat{v}, \hat{n}) = t_1(\hat{n}) \tag{62}$$

The specular Phong computation is similar and only depends on the used the reflected viewer direction:

$$L(\hat{v}, \hat{n}) = t_1(\hat{r}_v) = t_1(2(\hat{n} \cdot \hat{v})\hat{n} - \hat{v}) \tag{63}$$

The idea to combine both parametrizations leads to the first “real” (with more than one texture) factorization parametrization:

$$L(\hat{v}, \hat{n}) = t_1(\hat{n}) t_2(\hat{r}_v) \tag{64}$$

This normal-reflection (NR) parametrization is surprisingly effective for a number of BRDFs. Not only the Phong BRDF is approximated quite good, also the Cook-Torrance BRDF gives acceptable results.

A small improvement can be made by adding a third texture with the viewing direction, leading to the normal-reflection-view (NRV) parametrization:

$$L(\hat{v}, \hat{n}) = t_1(\hat{n}) t_2(\hat{r}_v) t_3(\hat{v}) \tag{65}$$

Usually the viewing direction is as unsuitable to be good parameter here as it is in the BRDF incident view parametrization, but the mere fact that the approximation uses one more factor than the previous one, ought to lead to a slightly smaller error.

Another possible parametrization tries to improve the NR parametrization (equation 64) with a modified version of the Gram-Schmidt halfangle-difference (GSHD) parametrization (equation 23, page 22).

The idea behind this parametrization is that often the incoming light comes predominantly from one direction, for example in outdoor environments from the

direction of the sun. This direction is then used in place of $\hat{\omega}_i$ in the computation of the halfangle and difference vectors.

The dominant light direction has to be explicitly specified or it can be computed with a weighed average of the incoming light function. Assume a number of light sources K evenly distributed on a whole sphere, with a direction $\hat{l}_k$ and a light intensity of e_k . The dominant light direction $\hat{i}$ can be computed as:

$$\hat{i}=\sum_{k=1}^{K} e_k \hat{l}_k \qquad (66)$$

The halfvector $\hat{h}'$ in world space can now be computed as:

$$\hat{h}'=\frac{\hat{i}+\hat{v}}{\|\hat{i}+\hat{v}\|} \qquad (67)$$

The vector needs to be transformed into the local surface frame. We need the normal and a tangent, that can be computed with a function as above.

$$\hat{b}=\hat{n}\times\bar{t}(\hat{n})$$
$$\hat{h}=\begin{pmatrix}\hat{h}'\cdot\bar{t}(\hat{n})\\ \hat{h}'\cdot\hat{b}\\ \hat{h}'\cdot\hat{n}\end{pmatrix} \qquad (68)$$

The tangent computation from the normal usually causes a singularity, assuming that it is done with a cross product to some third direction. In the BRDF computation above this did not matter, because any tangent can be taken for an isotropic BRDF evaluation. A way to minimize the effect of the singularity is to map it at a visually less important position, for example to surfaces with a normal orthogonal to the viewer's direction.

The difference vector $\hat{d}$ is the light direction, here the dominant light direction $\hat{i}$, in a new coordinate reference frame. The coordinate frame is built by the halfangle vector, a projection of the surface tangent and a vector orthogonal to both. These vectors can all be computed in world space. The mapping to the new coordinate frame makes an additional mapping to the local surface frame superfluous:

$$\hat{t}'' = \bar{t}(\hat{n}) - (\bar{t}(\hat{n}) \cdot \hat{h}')\hat{h}'$$
$$\hat{t}' = \frac{\hat{t}''}{\|\hat{t}''\|}$$
$$\hat{b}' = \hat{h}' \times \hat{t}' \tag{69}$$
$$\hat{d} = \begin{pmatrix} \hat{i} \cdot \hat{t}' \\ \hat{i} \cdot \hat{b}' \\ \hat{i} \cdot \hat{h}' \end{pmatrix}$$

With the halfangle and difference vectors and the previously used normal and reflected viewer direction, the normal-reflection-halfvector-difference (NRHD) parametrization can be written as:

$$L(\hat{v}, \hat{n}) = t_1(\hat{n}) t_2(\hat{r}_v) t_3(\hat{h}) t_4(\hat{d}) \tag{70}$$

The validity of this parametrization can be shown by using it for the approximation of the lighting computation for a single light source, which results in visually similar textures to the approximation of a BRDF with GSHD parametrization and an approximation of the $\hat{n} \cdot \hat{l}$ term in the normal texture.

The results of this four factor approximation in complex lighting situations show about the same statistical error as the other parametrizations, but the visual artifacts resulting from the tangent singularity might negatively impact the appearance.

5.5 Texture Mapping

The textures created with a factorization of the LC need to map a direction to certain texels. Unlike the parameters of a BRDF factorization the direction lies anywhere on a whole sphere, not just a hemisphere.

Therefore using one parabolic or hemispherical map for each factor is not possible. There are basically three texture mapping types that can map directions on a whole sphere: one spherical map, two parabolic maps, or a cube map with six faces.

Spherical maps are problematic, because they have a greatly varying sampling density and they have a singularity, because one direction is mapped to all positions on the inner circle of the map.

Parabolic or cube maps both have a good sample distribution and can both be used for real-time rendering, cube maps should be preferred if supported by the rendering hardware. For the factorization algorithm both have one difficulty. The original algorithm was not intended to handle several (two for parabolic or six for cubic) separate maps for each texture.

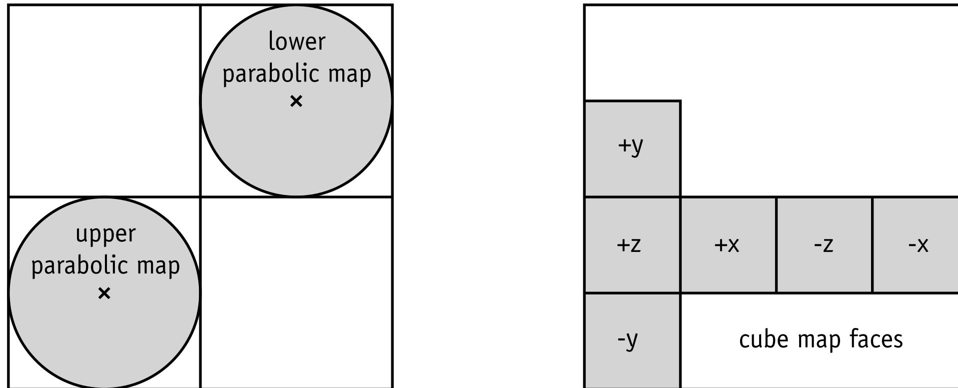


Figure 11: Layout of multiple maps for one texture (left: parabolic, right: cube map)

In our implementation the maps are simply put together into one square texture. Of course some areas of this texture are not used, but so is the area outside the inner circle of parabolic maps anyway. After the factorization these maps can be split into separate textures for rendering.

The LaPlace operator could be modified to smooth the border texels of one map with the texels of another map at similar directions. Our implementation uses double parabolic maps and this additional smoothness constraint did not seem necessary.

5.6 Function Sampling

As described in chapter 4.2 Sample Selection, a good method for selecting samples improves the factorization result and performance.

Similar to the sample iterator described for the sampling of analytical BRDFs the samples for the LC factorization can be obtained by using two direction iterators that use reverse-projected texture positions in the map with the two parabolic submaps. For both NR and NRV parametrizations, one direction iterator can set the normal and the other the view direction – allowing a simple mapping to the LC function parameters.

For the NRHD parametrization the above sample iterator is acceptable as well, a few additional samples, taken regularly over the whole sphere might give a small improvement in the result.

Some normal/view combinations will be useless and may have negative effects on the computation. If the view direction is below the surface plane defined by the normal, the LC will not have a valid result. Our implementation uses a sample iterator filter, that excludes invalid combinations.

Another issue with sampling the LC is how to define an incoming light function $L_i(\hat{l})$ with an environment map. Basically a number of light sources can be defined with the texels of the environment map. Using cubic environment maps all six faces of the cube can be used to select texels with an equal distribution in both directions of each face.

The resolution of cubic environment maps used for rendering is often to high. Taking one light function sample for each texel of a $256 \times 256 \times 6$ will lead to about 400,000 light samples to be weighed by a BRDF for each LC evaluation. Just taking about 20 or 32 light samples in each cube map direction should be enough for non-mirror like materials.

In order to replace the integral in the LC (as in equation 61) by a sum, the sampling needs to be done with equally distributed integration steps (over the vector $\hat{l}$). Directly taking cube map samples does not lead to an exactly equal distribution. Weighing the samples in a similar way to the differential solid angle (see chapter 2.3.3 Definition of the BRDF) would be optimal. However the sampling density over a cube map does not differ significantly, and this step can be neglected.

5.7 Results

The method has been tested with three BRDFs and several lighting environments. The standard Phong model, the modified, physically plausible Phong model [Lafortune1994], and the Cook-Torrance model [Cook1981] have been used as BRDF. Among the lighting environments used are the following environment maps: The “Loch” scene by Jeff Heath (for comparisons: it is also used in [Kautz2000]), an “Arizona Desert” sky, an environment map with some “painted” light sources (area lights) and several intensity gradients.

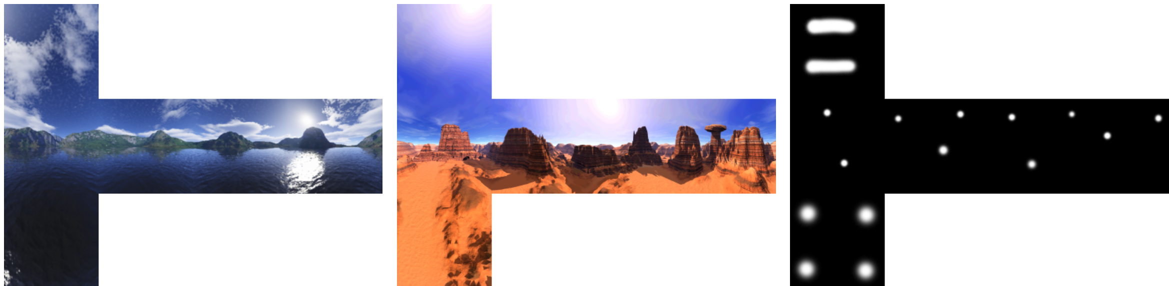


Figure 12: Environment maps (Loch, Desert, painted light sources)

Note that environment maps usually have a far smaller intensity range than real lighting situations. In the Loch scene the light energy coming from the direction of the sun ought to be several powers of ten higher than the diffuse reflections of the light on the islands. To at least partially enhance the range the environment map, the highlights (e.g. intensity above 90 per cent) have been scaled by a factor (e.g. 10). As an alternative a point light source with a high intensity might be added, e.g. at the direction of the sun in the example above.

Some of the combinations of materials, environment maps and parametrizations are shown below with a rendered result, the texture maps and error measurements. All textures have a resolution of 64×64 and contain two parabolic maps of half the

resolution with the layout as in figure 11. For rendering these textures have been converted to cube maps.

As sample iterator for the factorization the above texture based direction iterators have been used. To limit the number of samples the direction iterators only use a texture size of 48×48. This results in about 400,000 LC samples, each computing the result of 2,400 light samples from the environment maps. Having to weigh all light samples by a BRDF consumes the bigger part of calculation effort needed for the factorization. On a current 1.4 GHz. AMD Athlon processor this takes about 20 minutes with a Cook-Torrance BRDF, the factorization itself requires another 15 minutes. Note that these values are for highest quality results, for average quality results significantly less LC and light samples need to be taken.

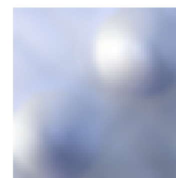
The textures are prescaled (see chapter 4.3.1 Prescaling of Textures) and the alpha channel has been used to enhance precision (see chapter 4.3.2 Alpha Blending Contouring Prevention). For display of the textures the alpha channel usage has been deactivated. Below the textures the correction color and the range enhancement scale factor is displayed.

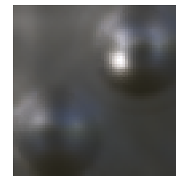
The error measurements are abbreviated as follows: Average (Avg) and maximum (Max) absolute (Abs) error and relative (Rel) error, all between the exact measurements and the reconstruction from double precision floating point values. RMS I is the root mean squares error of the same difference, while RMS II is the error of the reconstruction from 8-bit integer textures with alpha channel. All values are the arithmetic averages over the RGB components. The error has been measured with the same samples that have been used for the factorization.

For comparison of the Cook-Torrance approximations in the Loch environment figure 13 shows the LC computed for every vertex on an very fine tessellated teapot (almost per-pixel computation).

Figure 13: Teapot with exact lighting computation

Cook-Torrance, Loch environment, NR parametrization

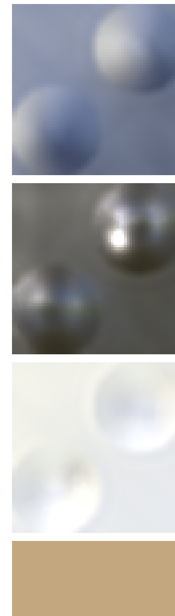




Scale factor 4

Avg Abs	0.0587
Max Abs	4.1961
Avg Rel	0.1093
Max Rel	3.5218
RMS I	0.1137
RMS II	0.1322

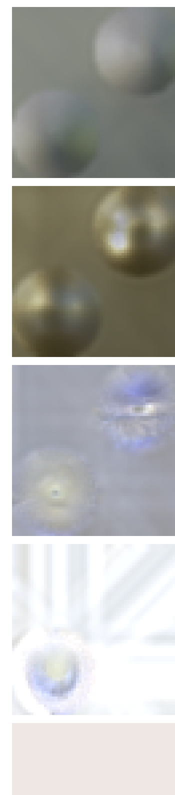
Cook-Torrance, Loch environment, NRV parametrization



Scale factor 4

Avg Abs	0.0580
Max Abs	4.2081
Avg Rel	0.1051
Max Rel	3.3035
RMS I	0.1130
RMS II	0.1379

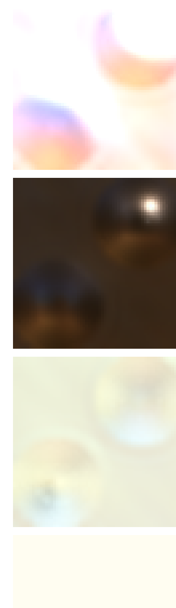
Cook-Torrance, Loch environment, NRHD parametrization



Scale factor 4

Avg Abs	0.0619
Max Abs	4.4475
Avg Rel	0.0919
Max Rel	2.1229
RMS I	0.1395
RMS II	0.1801

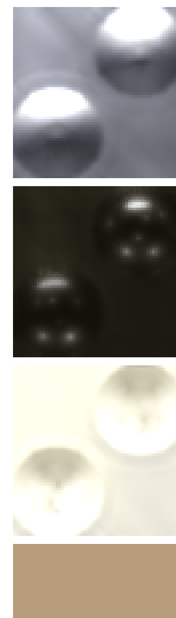
Cook-Torrance, Desert environment, NRV parametrization



Scale factor 4

Avg Abs	0.1007
Max Abs	3.7643
Avg Rel	0.1041
Max Rel	3.6739
RMS I	0.2282
RMS II	0.6423

Cook-Torrance, Painted light sources, NRV parametrization



Scale factor 4

Avg Abs	0.0740
Max Abs	4.7215
Avg Rel	0.3522
Max Rel	16.3575
RMS I	0.1857
RMS II	0.1988

Modified Phong, Loch environment, NRV parametrization

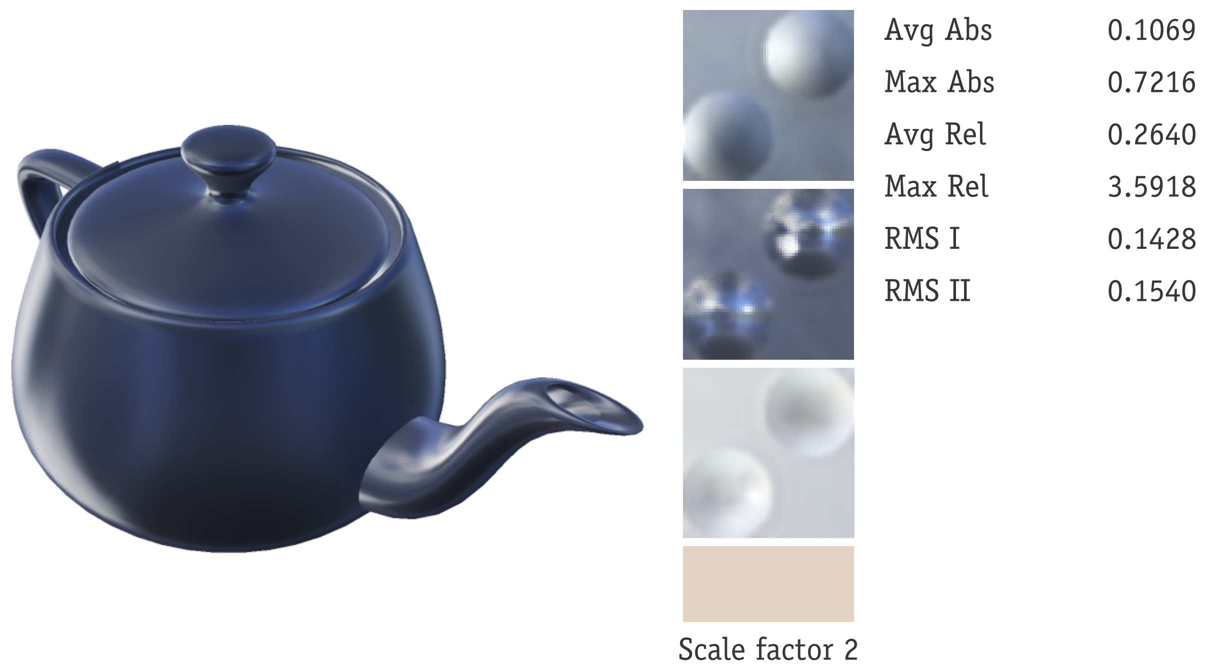


Figure 14: Results of lighting computation factorizations

Additionally the result of using the algorithm for the technique of Heidrich and Seidel [Heidrich1999] (see equation 41, page 30) is presented in figure 15. The factorization can be stopped early to only fill one texture, showing that the techniques for filtering an environment map with the Phong BRDF can be regarded a specialization of the LC factorization.

Specular part of Phong model, Loch environment, Reflection vector parametrization

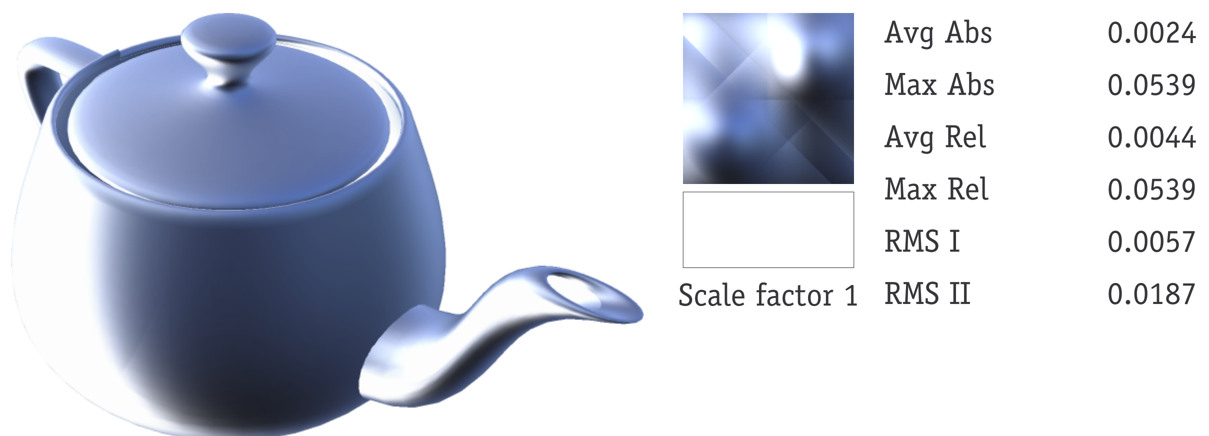


Figure 15: Result of glossy environment map with specular Phong model

5.8 Evaluation

The proposed method has been shown to work with a very limited number of textures at an acceptable visual quality. Both NR and NRV parametrizations have very simple texture coordinate computations in real-time rendering and can be used on most current graphics hardware with acceptable performance.

The NRHD parametrization can currently be more regarded as a proof of concept than a high-quality separation. It shows that the BRDF approximation can be considered a special one light source LC, but in most lighting environments the dominant light direction does not seem to be practical and the singularity poses another problem.

Compared to environment map prefiltering techniques the method is the first multiple texture factorization approach. Therefore the generality is its greatest advantage. The factorization will try fit any characteristics of the lighting environment and the material as good as possible. Unlike the method by Kautz and McCool [Kautz2000II] the restriction to radially-symmetric BRDFs does not exist, neither are 3D textures necessary.

However factorization speed is probably the biggest disadvantage. Especially compared to the hierarchical prefiltering proposed by Kautz et al. [Kautz2000], the full evaluation of the LC function and the factorization have significantly higher computation effort than the application of a 2D filter kernel. Additionally the BRDF factorization has only to be done once for each material, while the LC factorization needs to be repeated for every change in the lighting environment. However in real-life applications a low-resolution factorization with a small number of LC samples can already produce acceptable results, high-quality textures only need to be generated for final production results.

Environment maps in general have the restriction that the environment should be static and relatively far away compared to the size of the textured geometry. Often these restrictions can be neglected, at least to a certain extent. However a fast moving car in a racing game will look awkward with a static environment map. In this case the environment map might be used to simulate the general, far away surroundings like the sky and the landscape. Some additionally applied BRDF- or Phong-based lights can simulate close street lights and the lights from other cars. Alternatively a dynamically generated cubic environment map might be added to show the close environment updated every few frames. Of course this map would not have the correct reflective properties of the material, but in the combination with the factorization textures this will not be easily noticeable.

6 Implementation Details

This chapter describes details of our implementation used for factorization and display of BRDFs and LCs. The implementation is written in C++ and uses an object-oriented design. Although this chapter is primarily intended to give a brief overview of the implementation, enabling the reader to extend and alter it for special applications, it can also be used without access to its source code as the concepts may be useful for alternative implementations as well. However most production level implementations might not need the full flexibility of the chosen approach, as it is mostly optimized for experimenting with various setups of the method and only partially for speed.

First the basic concepts and interfaces are described. Then a user's guide will describe how to set up factorizations and find an optimal result. Note that there is no complete user interface for all steps of the process and parts of the setup need to be done in source code.

6.1 Basic concepts

The basic concepts introduced in this section are modeled as class interfaces or C++ templates. Some of the concepts have already been mentioned previously, e.g. the sample and direction iterator, but will be discussed from an implementation aspect here.

The implementation is basically split into a texture generation part and a real-time rendering part. However some concepts are used in both parts of the implementation and therefore differences are described where applicable.

6.1.1 Image

A very basic concept of the implementation is the abstract data type image. It is a C++ template, derived from the C++ Standard Template Library (STL) vector type. The template has to be instantiated with the data type used for the pixels of the image. This allows to use any kind of “pixels”, e.g. scalar floating-point values or RGB byte vectors.

Additionally to the interface of the vector class, the image allows access to pixels by using a two coordinate vector with the array access operator [], or with the function operator () with two integer coordinate parameters.

Another data type for images is CubeImages, also a template based type. One CubeImage object contains six images of the type Image and offers both individual access to the images as well as access to pixels with a three component vector that is used like the texture coordinates for cube mapping.

6.1.2 BRDFSampler

The BRDFSampler interface offers the evaluation of a BRDF for a given parameter set. It contains only one method for computing a color sample of the BRDF for incoming and outgoing light directions in local surface coordinates. Specific materials are derived from this interface and implement the method. This has been done for both analytical and measured BRDFs.

Analytical BRDFSamplers

The Cook-Torrance BRDF (CookBRDFSample), Poulin-Fournier BRDF (PoulinFournierBRDFSampler) and several Phong BRDFs have been implemented as classes using the BRDFSampler interface. They provide additional methods for setting the specific parameters for evaluating the BRDF.

MeasuredBRDFSampler

The MeasuredBRDFSampler class allows sampling of measured BRDF data. Sample data can be read from ASTM files, as used in the Cornell University BRDF database.

Currently the BRDF sampler does not provide interpolation of measured data, and can only be used in combination with its own SampleIterator (see below).

6.1.3 LightingEquationSampler

The LightingEquationSampler has the same interface as the BRDFSampler, but evaluates a full lighting computation. Due to a minor inaccuracy in the naming scheme of the implementation, the term LightingEquation refers to the function called lighting computation in this thesis.

The LightingEquationSampler implements the method of the BDRFSampler interface, but reinterprets the incoming light direction as the normal in world coordinates. To compute the LC for a given parameter set, a BRDFSampler object is used and the incoming light function can be specified by setting light sources explicitly or by sampling a cubic environment map.

6.1.4 DirectionIterator

This interface represents the direction iterator concept introduced in chapter 4.2 Sample Selection (page 34). It is modeled similar to STL iterators.

Classes implementing this interface allow to select one direction vector of a sequence of vectors with a special layout. With the increment operator the next direction vector in the sequence is selected, with the indirection operator (*) the selected direction is retrieved.

Some exemplary classes implementing the interface are the SphereDirectionIterator, the CubeDirectionIterator and the ParabolicMapDirectionIterator.

SphereDirectionIterator

This iterator selects all directions of a latitude/longitude grid on a unit sphere. See page 34 for a description of this method. An equivalent iterator on a hemisphere is called HemisphereDirectionIterator.

CubeDirectionIterator

The CubeDirectionIterator iterates over the six faces of a cube with a constant step, similar to the pixel coordinates of a cube map. This is particular useful for sampling the incoming light function for the LC from a cubic environment map.

ParabolicMapDirectionIterator

This iterator selects directions by reverse projecting the coordinates in a parabolic texture map as described on page 34. The DoubleParabolicMapDirectionIterator uses the same method to iterate over one texture with two parabolic submaps.

6.1.5 SampleIterator

This interface implements the sample iterator concept introduced in chapter 4.2 Sample Selection (page 34). The state of an object with this interface stores which incoming and outgoing directions are used to sample a BRDF. By using the increment operator on the object the next set of directions can be selected. With the method getDirections the current set is retrieved.

Some classes implementing the interface are TwoDirectionSampleIterator, FilterValidEyeDirectionSampleIterator, MeasuredBRDFSampler::MeasuredSampleIterator and ReplicateIsotropicBRDFSampleIterator.

TwoDirectionSampleIterator

This template class is the most general sample iterator class and iterates over both sample directions with two DirectionIterator objects. With the first direction iterator the incoming light direction for a BRDF sample is set. The second direction iterator iterates over all outgoing light directions for each selected direction of the first iterator. Thereby the total number of samples is equal to the product of the number of steps both direction iterators can take.

Many important sample iterators can be modeled with this template. The class IncidentViewTextureSampleIterator is derived from a TwoDirectionSampleIterator using two ParabolicMapDirectionIterators. The HalfangleDifferenceTextureSampleIterator uses

the same basic concept, but reinterprets the vectors from the direction iterators as the halfangle and difference vectors used in the GSHD parametrization and computes the BRDF sample directions with them.

FilterValidEyeDirectionSampleIterator

This template based sample iterator wraps around another sample iterator and filters all invalid sample direction combinations. Normal/view direction combinations that do not have a valid LC function value (see page 49), are not iterated and simply skipped by an object of this type.

MeasuredBRDFSampler::MeasuredSampleIterator

This sample iterator works closely together with a MeasuredBRDFSampler object, and can only be accessed via the interface of the sampler. While iterating with this iterator over all measured samples, the sampler object will return the exact sample for the sample direction set.

ReplicateIsotropicBRDFSampleIterator

This iterator is intended to increase the number of samples of a measured isotropic BRDF. Usually the measured data has one fixed sampling direction, in the measured Cornell data this is ϕ_i . To get data samples over all possible directions, the measured samples need to be replicated with an increasing angular offset in the ϕ angles.

This iterator is a wrapper around the iterator of a measured BRDF and repeatedly iterates over the BRDF samples while offering a shifted set of sample directions.

6.1.6 DirectionMapper

The DirectionMapper interface describes how to map a direction vector onto a texture. Its most important method is mapDirection, which computes a two component vector with texture coordinates between zero and one from a direction vector in spherical coordinates.

Another method is primarily intended for real-time rendering and can compute the texture coordinates with less precision and can also return three-component texture coordinates as they are necessary for cube texture maps.

For some texture mapping types the corresponding DirectionMapper interface is called HemisphericalDirectionMapper, SphericalDirectionMapper and ParabolicDirectionMapper. For 2D textures with multiple submaps as displayed in figure 11 (page 49) the interfaces are called DoubleParabolicDirectionMapper and CubicDirectionMapper.

While the CubicDirectionMapper creates texture coordinates of a flattened cube map in a 2D texture, the interface CubeMapDirectionMapper generates 3D coordinates necessary for real cube maps. Therefore it only implements the real-time method mentioned above.

6.1.7 Parametrization

A class implementing the Parametrization interface represents the flexible parametrization setup as described on page 33. Using the symbols introduced there, the interface allows to determine the number of textures T and approximation factors J. Each factor is represented as an object that contains both the projection function π_j and the index k of the texture it uses.

Factor interface

The factors are based on an interface with the method mapTextureVector that implements the projection function. For a given set of two directions (BRDF or LC parameters) a vector is computed.

Derived classes like FactorWI and FactorWO simply return the first or second vector. Other classes like FactorH and FactorD compute the halfvector or the difference vector of the GSHD parametrization. For the LC approximation the classes FactorReflection, FactorDominanceH and FactorDominanceD compute the projection functions described in chapter 5.4 (page 45).

The method mapTexturePosition of the factor interface uses the vector computed from mapTextureVector in combination with a DirectionMapper object to compute the texture coordinates for a set of two directions. As with the DirectionMapper interface a real-time rendering version of this function allows for optimization with less precision.

Derived Parametrization Interfaces

The Parametrization classes not only describe the parameter setup of a factorization, but also offer the functionality for real-time rendering. The method computeTextureCoordinates takes a number of vertices and computes the correct texture coordinates. Since the vector parameters for the BRDF and LC reconstruction are computed differently, two specialized interfaces offer access to specifying the necessary data for real-time rendering.

The BRDFParametrization interface allows to set the eye and light position in model space coordinates. Classes based on the implementation of this interface in the abstract class AbstractBRDFParametrization can compute the texture coordinates for vertices with this data. A default implementation in AbstractBRDFParametrization can compute the texture coordinates for any combination of factors and DirectionMappers, but will do this quite inefficiently by calling several functions for each vertex. For often used

combinations the derived Parametrization class ought to override the default implementation with a more efficient one.

The LightingEquationParametrization interface needs both the eye position in world coordinates and a matrix that transforms object to world coordinates. The texture coordinate computation in AbstractLightingEquationParametrization computes the parameters for the LC reconstruction in world space and needs to transform the vertex normals to world coordinates. The same efficiency considerations as with AbstractBRDFParametrization apply.

Implementing Classes

The classes implementing a concrete parametrization are derived from any of the two abstract classes. They only override the number of texture values and the array (STL vector) of references to the factor objects.

The original parametrization of the HF method is represented by the class IncidentHalfangleViewBRDFParametrization, the GSHD parametrization is called HalfangleDifferenceBRDFParametrization. The LC parametrizations mentioned in this thesis are called NormalReflectionLightingEquationParametrization (NR), NormalReflectionViewLightingEquationParametrization (NRV) and NormalReflectionHalfangleDifferenceLightingEquationParametrization (NRHD).

6.1.8 BRDFSeparator

The BRDFSeparator class contains the essential parts of the factorization algorithm. It offers methods for all steps of the factorization and stores setup and intermediate data. Unlike its name and some of the methods names suggest, it can both be used for BRDF and LC factorizations.

An object of this class requires three references to BRDFSampler, SampleIterator and Parametrization objects. Other parameters for the factorization, e.g. the texture size and the weight of the LaPlace operator can be set via additional methods.

After the factorization has been set up, the whole algorithm can be executed by calling the doSeparation method. To gain greater control over the steps of the algorithm, the steps can be called as separate methods. After the factorization has been done completely once, only parts of the algorithm that adjust the texture range need to be repeated to test other texture scaling setups.

The complete algorithm has been broken down into the following steps:

1. computeBRDFSamples: Computes and stores all samples according to the SampleIterator and the BRDFSampler

2. measureAverageBRDF: Measures the average value of all samples. This value is necessary for computing the logarithmic BRDF/LC values used in the following steps
3. estimateStartingSolution: Fills the approximation textures with a starting solution for the equation system solver, works according to the algorithm in chapter 3.1.2 Starting Solution (page 27)
4. computeEquations: Creates the matrix and the vectors for the linear equation system, and solves the system with the QMR algorithm in three passes, one for each color channel
5. exponentiateTextures: Transforms the computed textures back from logarithmic space
6. measureError(false): This optional step measures the error of the approximation, using a reconstruction from the textures with double-precision floating point values
7. adjustTextureRange: Normalizes the texture range, so that the maximum texture value is one. To restore the original range later a correction color is computed
8. prescaleTextures: This optional step prescales the textures with the algorithm described in chapter 4.3.1 Prescaling of Textures (page 37)
9. clampTextureRange: Converts the floating-point textures to integer (byte) textures, possibly clamping values out of range and computing the alpha channel with the method described in chapter 4.3.2 Alpha Blending Contouring Prevention (page 38)
10. measureError(true): This optional step measures the error using a reconstruction from integer textures
11. writeMaps: Writes the integer maps to TGA image files and optionally to 16-bit raw data files
12. writeDescriptionFile: Writes a textual description file, which stores correction color and scaling values as well as the parametrization and texture mapping type used

The error measurements here are written at the earliest possible position, and can be done at any later position. Additionally anywhere after step 3 the methods writeDoubleMaps and readDoubleMaps can be used to store or retrieve the floating-point approximation textures to/from raw image files. This is especially useful for doing the texture prescaling step in an interactive application at a later time.

A final word on the flexibility of the BRDFSeparator algorithm. It can not only be used for factorizations, but also for a lot of resampling and projection tasks. In our

implementation it is used to convert parabolic or double parabolic maps to cube maps. For this purpose a special BRDFSampler object extracts a pixel from a texture map at the position computed by a DirectionMapper from the first of the two BRDF parameters. The SampleIterator uses a CubeDirectionIterator to define the first sample parameter and has only one fixed position for the second parameter (that is ignored by the BRDFSampler anyway). Performing the factorization algorithm, omitting the real equation system computation, will calculate the texture projection (essentially in the estimateStartingSolution method). For the target texture a CubicDirectionMapper is used, and the six cube map faces only need to be extracted from the result.

6.1.9 Rendering Scene Graph

The real-time rendering part of the implementation uses a scene graph to organize objects and their attributes for rendering. Most parts of the scene graph have universal applications and have been used in other projects as well. The graph is intended to work with OpenGL, but other state-driven architectures like DirectX can be adapted easily.

After a brief description of the basic structure, the additions for BRDF/LC reconstructions are explained.

Basic structure

The objects are structured in a directed acyclic graph (DAG) consisting of leafs (class DagLeaf) and nodes (class DagNode).

Geometries are derived from the DagLeaf class, e.g. DagCube or DagSphere. Transformations and other attributes of the geometries are derived from DagNode and apply to all their child nodes or leafs, e.g. DagTransform, DagColor, DagTexture or DagAnim.

In order to display the whole contents of the graph a render method, that recursively traverses all nodes and their children, is called. Each node will set its attributes before traversing its children and restore the previous state afterwards. Most classes closely wrap OpenGL features, e.g. DagColor calls glColor before rendering its child nodes.

Geometries for BRDF/LC reconstruction

A geometry that will be used with the BRDF or LC based lighting needs to have continuously updated texture coordinates. For this purpose all objects need to implement an interface for recomputing the texture coordinates. The class DagBRDFLitObject is the base class for objects with this interface. The objects derived from this class need to have vertex data consisting of position, normal, tangent, binormal and space for several texture coordinate sets. The geometry specifies the first four of these data fields once and the texture coordinates will be computed when necessary.

During rendering only some of these data sets need to be given to the graphics system. For rendering with multiple passes only one or more texture coordinate sets will be used. Alternatively, if vertex programs/shaders (NVIDIA OpenGL extension or DirectX 8) are used, the texture coordinates can be generated in hardware and instead of these the tangent and binormal are specified together with the geometry (e.g. with glVertexAttrib3fvNV).

In our implementation only one class derived from DagBRDFLitObject is used. DagBicubicPatch tessellates bicubic patch data and automatically takes tangents and binormals from the partial derivatives of the surface. Polygonal data would have to be specified directly with tangent data or the tangents could be generated as orthogonal vectors to the normal and another direction.

DagBRDFMaterial

The most important element in the BRDF/LC reconstruction is the DagBRDFMaterial class. It is derived from DagNode and applies the material to all its children, more precisely to all children offering the interface of the DagBRDFLitObject class.

The DagBRDFMaterial loads BRDF or LC data with the textual description file generated by the factorization algorithm. First the texture mapping type is recreated. If a material offers the use of a CubeMapDirectionMapper and cube map hardware is available, this is used. Otherwise a DirectionMapper for 2D texture mapping is created. Then the used Parametrization object is created, which in turn allows the DagBRDFMaterial to know how many textures to load (with DagTexture objects). If vertex program hardware is available and a vertex program to generate texture coordinates for the specific texture mapping and parametrization type is available, the program is loaded (as DagVertexProgram object).

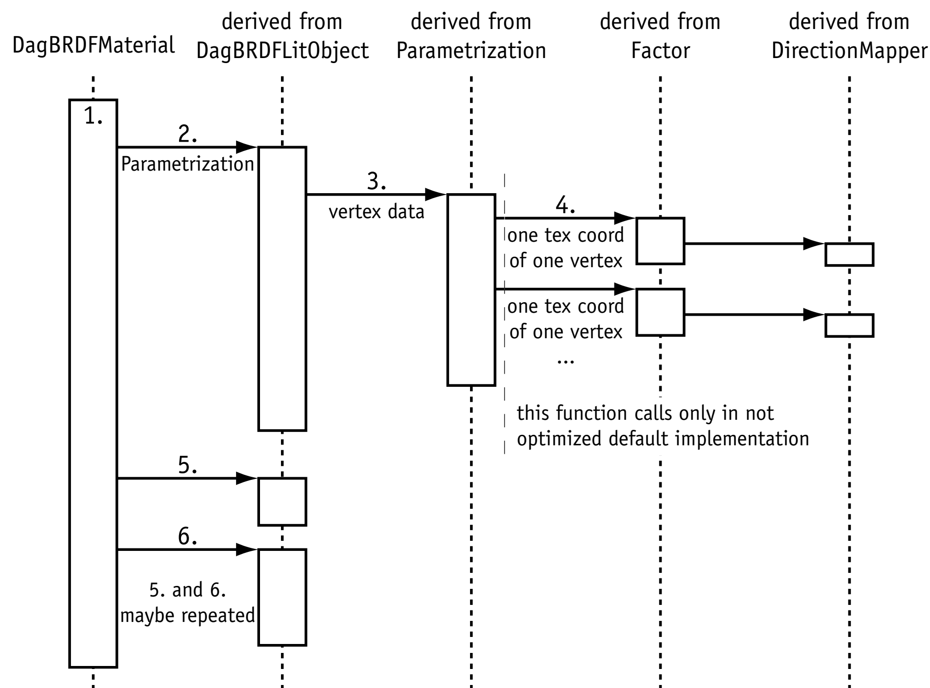


Figure 16: Rendering sequence in DagBRDFMaterial

During rendering the data flow is as follows:

1. Compute the vectors/matrix needed for texture coordinate generation (e.g. local eye/light position, see Parametrization interface, page 61)
2. Locate all children derived from DagBRDFLitObject and call their method for computing texture coordinates, giving them access to the Parametrization interface used.
3. The children call a method in the Parametrization interface once or multiple times to compute texture coordinates, giving it access to a continuous block of vertex data that the computation should be done for.
4. The Parametrization object computes the texture coordinates considering the parametrization factors and DirectionMapper type, either with the rather slow default implementation in the abstract Parametrization classes or with an optimized method for a specific DirectionMapper (suggested for real-life applications).
5. After the texture coordinate generation the DagBRDFMaterial object sets up the texturing units for application of the materials. If multiple rendering passes are necessary, the children are instructed to send only part of the texture coordinate data.

6. The render method of the children is called. Children derived from DagBRDFLitObject can use a method in the base class to send only the correct vertex data.

The complicated texture coordinate computation uses double linkage (two step dynamic linkage) in order to abstract both the vertex storage used in the geometry object and the parametrization setup. In a worst-case (performance-wise) scenario the geometry would call the computation method of the Parametrization object once for each vertex.

Using vertex programs steps 2 to 4 are not necessary. They are basically done in the vertex program, which has been specifically created for the chosen parametrization and texture mapping. In the future such a program could be created dynamically, connecting code fragments for parametrization factors and DirectionMapper functions.

Currently our implementation only allows usage of vertex programs, if rendering is done in one pass. This is no real limitation as they are only supported in hardware on recent systems with at least four texture units anyway. Using vertex programs with multiple pass rendering would require the programs to be split up to only generate part of the texture coordinate sets.

DagBRDFSamplerMaterial and DagLightingEquationMaterial

There are two other classes that can compute BRDF or LC materials. These are however not intended for real-time applications. Their only purpose is the comparison of materials with approximation textures to an exact material computation.

Both classes can use geometries with the DagBRDFLitObject interface, but will compute a color sample instead of texture coordinates at each vertex. This is basically the same procedure that is done for standard per-vertex Phong lighting.

In order to do this computation the classes derive a special class from the Parametrization interface, that is no parametrization at all. The only purpose of this class is to be called from the geometry with vertex data, that can in turn be processed to compute color values with a BRDFSampler or LightingEquationSampler object.

On a current hardware system most analytical BRDFs seem to be computable per-vertex with an acceptable performance in real-time. The LightingEquationSampler however is not computable in real-time, if a lot of light sources are used. However the material color only needs to be recomputed when the viewer or the scene changes. Computing the exact colors once and then for example adjusting the approximation textures on another geometry is a possible application of these classes.

Figure 13 (page 52) has been created with a DagLightingEquationMaterial class and a fine tesselated geometry.

6.2 User's Guide

The implementation consists of two separate applications that share much source code. In general the BRDFSeparation application can only perform the factorization algorithm, while the BRDFViewer can display the results.

However the BRDFViewer is also intended to interactively adjust the texture scaling and can in fact also perform the full factorization, but the lack of a complete GUI for the setup makes it unattractive to use the more complicated BRDFViewer for the factorization.

6.2.1 BRDFSeparation

The BRDFSeparation application is a non-interactive implementation of the factorization algorithm. It has been developed and tested on the Windows platform using Microsoft Visual C++ 6. Assuming the portability of the used vector/martrix libraries (SparseLib++/IML++) it should port easily to other operating systems and development platforms, because it does not use any operation system specific interfaces. It also is not based on any graphics library (OpenGL/DirectX). However the use of abstract data types (C++ templates) requires a recent C++ compiler.

The application does not come with a user interface. The setup for a factorization has to be done in source code, in the main procedure. Basically an object of type BRDFSeparator is created with the necessary BRDFSampler, SampleIterator and Parametrization object. Then some algorithm parameters need to be set (e.g. texture size) and the algorithm steps can be executed as described in chapter 6.1.8 BRDFSeparator (page 62).

The main procedure already contains a lot of possible setups for factorizations, that only need to be enabled or altered for specific purposes. As an example the setup of a LC factorization with the NRV parametrization using a Cook-Torrance BRDF and the Loch environment map is described.

1. Create a CookBRDFSampler object. The application also allows to dynamically load BRDFSampler objects from libraries. If the sampler object is part of a library, e.g. the ConfigurableBRDF library that is part of the implementation, the sampler can be loaded by the function loadBRDFSamplersFromLibrary.
2. Create a LightingEquationSampler that uses the BRDFSampler
3. Load the cube map into a CubeImage object
4. Set the incoming light function of the LightingEquationSampler to sample the CubeImage, eventually enhancing the dynamic range of the image (see page 50)

5. Create a TwoDirectionSampleIterator with two DoubleParabolicMapDirectionIterators
6. Set the map size of both direction iterators, e.g. to 48
7. Create a FilterValidEyeDirectionSampleIterator to wrap around the TwoDirectionSampleIterator
8. Create a NormalReflectionViewLightingEquationParametrization object with a DoubleParabolicDirectionMapper as parameter
9. With the results of steps 4, 7 and 8 create a BRDFSeparator object
10. In the BRDFSeparator set the base file name for the textures, the texture size (e.g. 64), the weight of the LaPlace operator (e.g. 1), a maximum value for clamping the LC (1 for any LC) and parameters for the linear equation system solver (e.g. a tolerance threshold of 10^{-3} and a maximum of 100 iterations).
11. Execute the factorization in the sequence described in chapter 6.1.8 BRDFSeparator (page 62). It is suggested to call the writeDoubleMaps method directly after the exponentiateTextures call. The BRDFViewer can restore these floating-point textures later and interactively adjust their range.

For different setups basically the same steps are necessary, only other objects and values will be used.

6.2.2 BRDFViewer

The BRDFViewer application is the interactive part of the implementation. It has been developed on the Windows platform using Microsoft Visual C++ 6. So far it has been tested both on Windows and Apple MacOS. The underlying OpenGL scene graph structure also has been used on Linux and SGI Irix systems, so porting the whole application ought not be difficult. The application requires working OpenGL 1.2 drivers. It uses up to four texture units if available in hardware, but already works with only one texture unit. However if the rendering requires multiple render passes, due to a lack of texture units, the rendered results might have less visual quality, in particular less contrast, caused by the clamping of intermediate values in the computation. To avoid these problems an accumulation buffer and another rendering pass might be used (see [Kautz1999]).

The BRDFViewer has a graphical user interface (GUI), but some setups require source code changes. Currently the list of possible BRDF/LC factorization textures is implemented as a constant array and needs to be extended once a new factorization has been processed (see file GluiWindow.cpp).

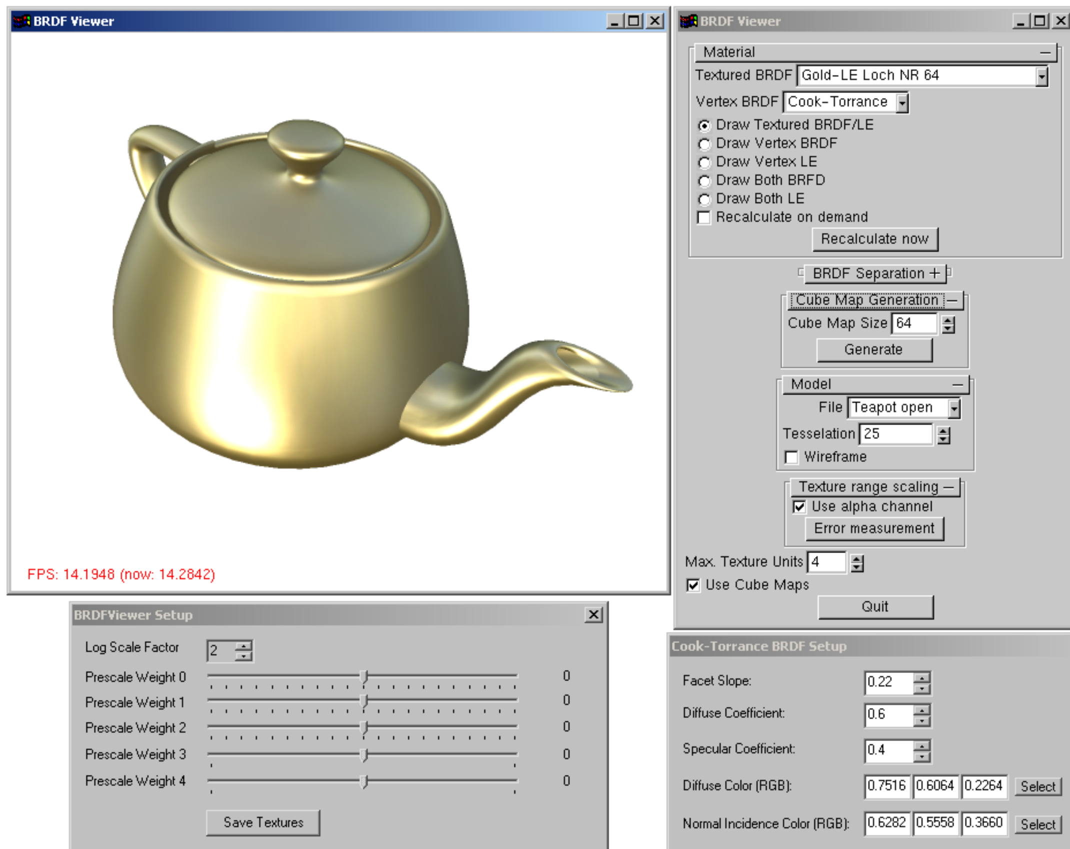


Figure 17: BRDFViewer user interface

The user interface of the BRDFViewer consists of several windows. The rendering window displays the currently selected model with the applied textures. With a three-button mouse the model can be rotated (left button pressed), zoomed (middle mouse button) and moved sideways (right button). While holding down the key 'v' on the keyboard, the mouse movement applies to the whole world instead of the model, thereby moving the light environment with the object.

The other three windows are setup dialogs. The long window on the right is used for general setups. This window uses the portable GLUI library and is also available in the MacOS version of the application, while the other two windows are currently only implemented for the Windows version.

The main setup window allows to select a material and model to display and to perform conversion operations with the textures. The material section allows to choose whether to display the model with a texture based BRDF/LC or with a per-vertex evaluated material. Both materials can also be displayed side by side. Since LC computations done per-vertex are time consuming, the “Recalculate on demand” option should be used. This option has no effect for texture based materials. The texture or vertex BRDF/LC can be chosen from drop-down menus. The chosen vertex material will

create a BRDF setup dialog (if available) as a window below the main setup window. This allows to interactively set the BRDF parameters, that can also be set manually for a factorization setup.

Below the material selection is a “Separation” section, allowing to execute a factorization algorithm. However as mentioned earlier this is not very flexible and the factorization should be done using the BRDFSeparation application.

The “Cube Map Generation” section allows to convert the textures of the currently selected material to cube maps. This option only works, if no cube maps are currently displayed. The use of cube maps can be deactivated by a check box at the bottom of the window.

The “Model” panel offers the selection of a bicubic patch model to be used for display. Also the tessellation of the patches can be controlled. It is suggested to use simple models, e.g. “Bowl” consists only of a single patch, for complicated per vertex-computations of a LC.

The texture range scaling section is split into two parts. The separate dialog window with the horizontal sliders is closely associated with this section in the main setup dialog. This separation has been done, because horizontal sliders are far more intuitive for setting up the range scaling than the controls the GLUI library offers. The range adjustment for a newly created material works as follows: First select the texture material and vertex BRDF/LC. Display them side-by-side and activate the window with horizontal sliders. First select a scale factor. This is displayed as the binary logarithm in the window, i.e. $3 \rightarrow 2^3 = 8$. Start low, e.g. with 1 or 2. Then adjust the prescale weights with the sliders, so that the texture result is more similar to the exact evaluation. These sliders also have a logarithmic scale. Once an optimal solution has been found it can be written to disk with the “Save Textures” button. With the check box “Use alpha channel” in the main setup window, the alpha channel precision enhancement can be enabled and disabled. Finally the textures can be converted to cube maps (see above). Note that the range adjustment works only on 2D textures, not cube maps. These have to be deactivated first in order to adjust the textures again.

With the “Error measurement” button the error of the current textures in relation to the selected vertex BRDF can be measured. However the used SampleIterator is important for the error measurement and it can only be changed in source code (in the method GluiWindow::initializeBRDFSeparator).

Finally the setup window allows to override the number of texture units used for rendering in order to simulate hardware with less capabilities.

7 Conclusion and Further Work

A new technique for approximating arbitrary materials in global illumination settings has been presented. The method combines the advantages of BRDF approximation methods with the approach of environment map prefiltering techniques. The results of the factorization are a limited number of 2D textures or cube maps, that can easily be applied in scenes with a high polygon count on current graphics hardware systems. Using a cubic environment map as incoming light function is straight forward and does not require special algorithms for measuring a light situation.

The advantages over a general BRDF approximation lie in the simultaneous approximation of the lighting environment around the objects, avoiding multiple reconstructions of the approximation results. Compared to environment map prefiltering methods the approach allows to approximate any material with isotropic BRDF with limited reconstruction effort, but a quality above that of single-texture filtering methods.

In the future better parametrizations for the LC factorization might be found. The NRHD parametrization should be regarded as an incentive to consider properties of the lighting environment in the factorization.

The factorization in general is not restricted to isotropic BRDFs. It might also be used with lighting computations for anisotropic BRDFs. These functions have five degrees of freedom and at least require three 2D textures or a combination of a 2D and a 3D texture for a complete approximation.

Since the factorization process is slower than most prefiltering techniques, several speed ups might be considered. A significant part of the time used for the factorization is spent by evaluating the LC, which often needs to weigh several hundred or thousand incoming light samples by BRDFs. These BRDF values could be computed with a reconstruction of the factorization results of a BRDF, which would also simplify the usage of measured BRDFs for the LC evaluation.

Furthermore it might be considered whether a full factorization is necessary when the light situation changes only partially. For example an environment predominantly lit by the sun might turn red when the sun is setting. Only a few “keyframes” of the process might be computed and the lighting can be interpolated in between. Alternatively applying simple filters, like a red coloring, might already be enough to simulate the change in the lighting environment.

Acknowledgments

The implementation of the factorization algorithm was originally based on a code written by Chris Wynn and Jan Kautz. The bicubic patch tessellation algorithm and accompanying models were also take from this code. The implementation currently uses the SparseLib++ and IML++ libraries from NIST [NIST1996] as well as the Vector Library VL by Andrew Willmott. Additionally the OpenGL Helper library GLH from NVIDIA is used.

Bibliography

Banks1994: Banks, David C., Illumination in diverse codimensions, SIGGRAPH Proceedings, 1994

Cabral1987: Cabral, Brian; Max, Nelson; Springmeyer, Rebecca, Bidirectional Reflection Functions from Surface Bump Maps, SIGGRAPH Proceedings, 1987

Cabral1999: Cabral, Brian; Olano, Marc; Nemec, Philip, Reflection Space Image Based Rendering, SIGGRAPH Proceedings, 1999

CIE1986: Commission Internationale de l' Eclairage, CIE Colorimetry Standard, Central Bureau of the CIE, 1986

Cook1981: Cook, Robert L.; Torrance, Kenneth E., A reflectance model for computer graphics, SIGGRAPH Proceedings, 1981

Fournier1995: Fournier, Alain, Separating Reflection Functions for Linear Radiosity, Eurographics Rendering Workshop, 1995

Freund1991: Freund, Roland W.; Nachtigal, Noël M., A Quasi-Minimal Residual Method for Non-Hermitian Linear Systems, Numerische Mathematik (60), 1991

Heidrich1998: Heidrich, Wolfgang; Seidel, Hans-Peter, Efficient Rendering of Anisotropic Surfaces Using Computer Graphic Hardware, Image and Multidimensional DSP Workshop, 1998

Heidrich1998II: Heidrich, Wolfgang; Seidel, Hans-Peter, View-independent Environment Maps, Eurographics/SIGGRAPH Proceedings, 1998

Heidrich1999: Heidrich, Wolfgang; Seidel, Hans-Peter, Realistic, Hardware-accelerated Shading and Lighting, SIGGRAPH Proceedings, 1999

Hestenes1952: Hestenes, Magnus R.; Stiefel, Eduard, Methods of Conjugate Gradients for Solving Linear Systems, Journal of Research of the Nat Bureau of Standards, 1952

Kautz1999: Kautz, Jan; McCool, Michael D., Hardware Rendering with Bidirectional Reflectances, Department of Computer Science, University of Waterloo, 1999

Kautz1999II: Kautz, Jan; McCool, Michael D., Interactive Rendering with Arbitrary BRDFs using Separable Approximations, Department of Computer Science, University of Waterloo, 1999

Kautz2000: Kautz, Jan; Vázquez, Pere-Pau; Heidrich, Wolfgang; Seidel, Hans-Peter, A Unified Approach to Prefiltered Environment Maps, Eurographics Rendering Workshop, 2000

Kautz2000II: Kautz, Jan; McCool, Michael D., Approximation of Glossy Reflection with Prefiltered Environment Maps, Proceedings Graphics Interface, 2000

Koenderink1996: Koenderink, Jan J.; van Doorn, Andrea J.; Stavridi, M., BRDF Expressed in Terms of Surface Scattering Modes, European Conference on Computer Vision, 1996

Lafortune1994: Lafortune, Eric P.; Willems, Yves D., Using the Modified Phong Reflectance Model for Physically Based Rendering, Department of Computer Science, K. U.

Leuven, 1994

Lafortune1997: Lafortune, Eric; Foo, Sing-Choong; Torrance, Kenneth; Greenberg, Donald, Non-linear Approximation of Reflectance Functions, SIGGRAPH Proceedings, 1997

McCool2001: McCool, Michael D.; Ang, Jason; Ahmad, Anis, Homomorphic Factorization of BRDFs for High-Performance Rendering, SIGGRAPH Proceedings, 2001

NIST1996: National Institute of Standards and Technology, Iterative Method Library V1.2, 1996, http://math.nist.gov/iml++/

Poulin1990: Poulin, Pierre; Fournier, Alain, A Model for Anisotropic Reflection, SIGGRAPH Proceedings, 1990

Rusinkiewicz1998: Rusinkiewicz, Szymon M., A New Change of Variables for Efficient BRDF Representation, Nineth Eurographics Workshop on Rendering, 1998

Schröder1995: Schröder, Peter; Sweldens, Wim, Spherical Wavelets: Efficiently Representing Functions on the Sphere, SIGGRAPH Proceedings, 1995

UND2000: University of Notre Dame, Iterative Template Library, 1997-2000, http://www.lsc.nd.edu/research/itl/

Wynn2000: Wynn, Chris, An Introduction to BRDF-Based Lighting, NVIDIA Corporation, 2000

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Lutz Latta
Wedel, 30. August 2001