Automatic BRDF Factorization

A Thesis presented

by

Forrester Hardenbergh Cole

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 8, 2002

Abstract

A method is presented to automatically generate factored approximations for arbitrary Bi-directional Reflectance Distribution Functions (BRDFs). The method extends previous work in BRDF factorization to include a search over a space of 4D to 2D projections. By using a search over projections, approximation error is improved and the necessity for user intervention is reduced. The user can also have more confidence that the approximation produced is close to optimal. The method improves upon or matches the approximation error of previous work in every case tested.

Contents

Chapter 1

Introduction

A basic problem in graphics is how to model the movement of light as it bounces from light sources through objects and finally to your eye. An integral part of this problem is how to represent the interaction of light with various materials. Every physical material reflects light in a characteristic way, which is what allows us to discern the difference between, for example, paper and plastic. This reflection information can be captured in a special function called a *Bi-directional Reflectance Distribution Function* or BRDF. A BRDF is used during rendering to determine the appearance of a material under varying viewing and lighting conditions.

In the past, real time graphics applications have used very unrealistic reflection approximations, the theoretically and empirically unsuitable Phong model being the most sophisticated. The use of general BRDFs, which can approximate the behavior of real materials to a high degree of accuracy, can drastically increase the visual realism of a scene. Unfortunately, current graphics hardware is not designed to facilitate rendering with full BRDFs. A full BRDF may be formulated as either a closed-form function or a lookup table. Current graphics hardware can not evaluate arbitrary functions during rendering, so a closed form representation is unworkable. Hardware acceleration of BRDFs has thus focussed on the lookup table formulation, but that formulation has two significant drawbacks: a BRDF lookup table is typically very large, and the format of the data is not well suited to use in hardware.

Work on BRDF factorization has attempted to alleviate these difficulties. By approximating a BRDF table with a set of factors, we can significantly reduce storage requirements and at the same time create a data format more suited to current hardware. Given a large enough number of factors, any BRDF can be approximated to arbitrary accuracy. To make a factorization useful, however, the BRDF must be approximated reasonably by a small number of factors. For a shortened series of factors, we would like to be sure that our approximation is as good as possible – that is, prove that the first k factors constructed by the algorithm are the optimal k

factors for that BRDF. Optimality can not be proven for all factorization schemes.

McCool *et al.*[13] have recently presented a new scheme that solves some practical difficulties of previous methods. Unfortunately, McCool's method does not allow for a proof of optimality. The scheme is based on a combination of projection functions and texture functions. The texture functions are found automatically, but the projection functions are arbitrary and decided by hand. It is impossible to prove, therefore, that a given set of k projections represents the optimal set of k projections. Also, McCool does not provide an algorithm for adapting the set of projections to an arbitrary BRDF. Prior analysis of the structure of the BRDF is necessary to pick effective projections. In this thesis we extend McCool's algorithm by creating a method that will automatically search for optimal projections. This algorithm aims to accomplish several goals: to reduce the approximation error of the McCool style factorization, to provide more confidence that the factorization found is optimal or close to optimal, and to reduce the amount of user input required to create an effective BRDF factorization.

In this chapter, we present background information on BRDFs and previous work in factorization. In Chapter 2, we present a high level overview of our algorithm and its capabilities and restrictions, including the definition of the projection search space. Chapter 3 contains more detailed information on the implementation of the search algorithm. Lastly, in Chapter 4 we present our results and conclusion, including a comparison between our algorithm and previous work.

1.1 Definition and Use of a BRDF

For an opaque, non-emitting object, the color and appearance of its surface depends on the quality of the light reflected from it. The quality of this light depends on several things: the position of the observer and the position of light sources relative to the surface, the material of the surface, and the intensity and color of the light shining on the surface. The dependence on position of viewer and light can be easily observed. Hold a book directly up to a light, and its cover appears bright; turn the book at an angle, and the cover becomes dimmer. Or examine the iridescent sheen of a peacock feather, which changes with the movement of your head.

Radiance is a general term for the power of light travelling in a certain direction. The power of light reflecting from a surface point $\mathbf{x}$ towards the eye is the radiance from $\mathbf{x}$ in the outgoing direction $\hat{w_o}$, denoted $L_o(\mathbf{x} \rightarrow \hat{w_o})$. The radiance striking $\mathbf{x}$ from an incoming direction $\hat{w_i}$ is denoted $L_i(\hat{w_i} \rightarrow \mathbf{x})$. For a given incoming radiance $L_i(\hat{w_i} \rightarrow \mathbf{x})$, the power of the light per unit area of the surface will vary depending on the orientation of the surface. A surface tilted away from the incoming direction will receive less power than one directly facing the surface. We call the power of the light

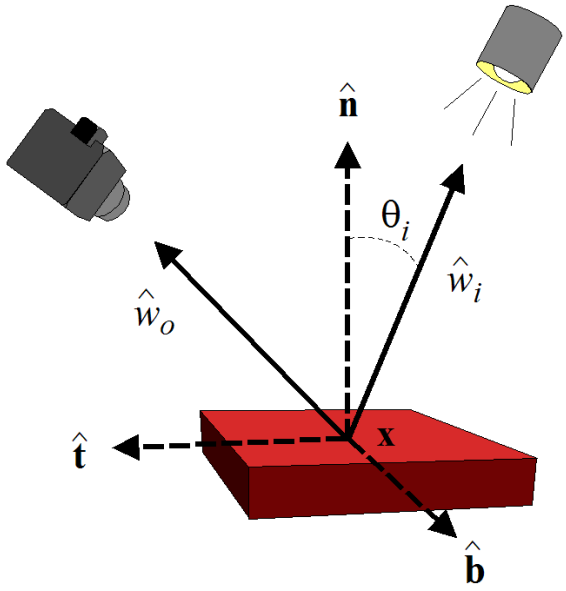

Figure 1.1: The geometry of the reflection equation. Light hits the surface at $\mathbf{x}$ from the incoming direction $\hat{w}_i$, and leaves the surface along the outgoing direction $\hat{w}_o$ (towards the camera).

per unit area of the surface from an incoming direction $\hat{w}_i$ the *irradiance* in direction $\hat{w}_i$. Usually, the irradiance of the surface directly determines how bright it appears. To get a measurement of the total light striking a surface, we integrate the irradiance over all incoming directions.

The relationship between light hitting and light reflecting from a surface can be formalized by the following equation, called the reflection equation:

$$L_o(\mathbf{x} \rightarrow \hat{w}_o) = \int_\Omega L_i(\hat{w}_i \rightarrow \mathbf{x}) f(\mathbf{x}, \hat{w}_i, \hat{w}_o) \cos\theta_i d\hat{w}_i \tag{1.1}$$

The equation expresses the total radiance leaving point $\mathbf{x}$ along direction $\hat{w}_o$ ($L_o(\mathbf{x} \rightarrow \hat{w}_o)$) as the integral of the light hitting $\mathbf{x}$ ($L_i(\hat{w}_i \rightarrow \mathbf{x})$) from all possible incoming directions $\hat{w}_i$ over the hemisphere Ω. The geometry of the reflection equation is shown in Figure 1.1. The integral has three components: the incoming radiance term L, the cosine geometry term, and the reflection term f.

We need to convert the incoming radiance measurement to an irradiance measurement before integrating. The irradiance in direction $\hat{w}_i$ is the radiance $L_i(\hat{w}_i \rightarrow \mathbf{x})$

multiplied by the cosine term $\cos\theta_i$. θ_i is the incident angle, the angle between $\hat{w}_i$ and the surface normal $\hat{\mathbf{n}}$. The more the surface tilts away from the source (as θ_i goes to $\pi/2$), the more the surface appears foreshortened from the light's position, and the lower the irradiance on the surface. With some simple geometry it is apparent that this effect is determined by the cosine of the incident angle θ_i. The irradiance is maximum when the light is shining directly down on the surface. It goes to zero as the incident angle goes to $\pi/2$, as the light increasingly grazes the surface. Note that $\hat{w}_i$ will always lie in the hemisphere centered on $\hat{\mathbf{n}}$, so the cosine will never be negative. This is simply because we assume that light cannot reach the surface from behind it.

The reflection term f is the Bi-directional Reflectance Distribution Function or BRDF. The BRDF relates the incoming irradiance to the outgoing radiance for each pair of directions $\hat{w}_i$ and $\hat{w}_o$. The value of the BRDF for an incoming direction $\hat{w}_i$ and an outgoing direction $\hat{w}_o$ is the ratio of the outgoing radiance to the incoming irradiance for those two directions. It is a scalar value in the range $[0, \infty)$. The range of the BRDF extends to infinity because it is used inside an integral, and may be, for example, a delta function. Since we place the cosine geometry term outside of the BRDF function f, the BRDF depends solely on the physical properties of the material of the surface.

The directions $\hat{w}_i$ and $\hat{w}_o$ are assumed to be specified in a frame aligned with the surface. In the standard surface frame the normal $\hat{\mathbf{n}}$ lies on the z-axis, the surface tangent vector $\hat{\mathbf{t}}$ lies on the x-axis, and the surface binormal $\hat{\mathbf{b}}$ lies on the y-axis. With the exception of some extremely simple functions, all BRDFs depend on the orientation of $\hat{w}_i$ and $\hat{w}_o$ with respect to $\hat{\mathbf{n}}$. Only a few BRDFs, however, depend on the orientation of $\hat{w}_i$ and $\hat{w}_o$ with respect to $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$. That is, one may rotate two given direction vectors $\hat{w}_i$ and $\hat{w}_o$ around $\hat{\mathbf{n}}$ and the value of the BRDF will not change. A BRDF that does not depend on $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$ is called *isotropic*, while a BRDF that does depend on $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$ is *anisotropic*. Most materials have isotropic BRDFs, for example paper, most cloth, and human skin. Materials with anisotropic BRDFs usually have some complex microstructure, such as the grooves in brushed metal. Imagine rotating a brushed metal plate on a table, while keeping your eye stationary. The apparent shininess of the plate changes considerably, which means that the reflection depends on the position of $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$.

The $\mathbf{x}$ term passed to the BRDF function f is designed to preserve the generality of the reflection equation by accounting for possible variation in the BRDF based on position. The simplicity of the reflection equation tends to hide the fact that reflection of light is an enormously complex phenomenon, which cannot be fully described for a point $\mathbf{x}$ without regard to the surrounding structure of the surface. Since it only has the two direction parameters, a BRDF can represent only the distribution of light reflecting from a single point on a locally flat surface. However, many materials

are opaque and smooth on a large scale, and these materials can be modelled very successfully with a single BRDF. Depending on the accuracy needed, even complex objects such as a grass lawn can be considered smooth on a large scale. In practice we will have one BRDF for each material, and the position on the surface will be irrelevant. For readability we suppress the $\mathbf{x}$ term, writing our BRDF as $f(\hat{w_i}, \hat{w_o})$.

The amount of light reflected by almost all materials varies with the wavelength (color) of the light. At its simplest, a BRDF holds the reflectance properties of one material for one specific wavelength of light. Sophisticated BRDFs are made up of many channels, each holding the reflectance value of the material at a certain wavelength of light. Rendering with a wide range of reflectance values is called full-spectrum rendering, and generates the most convincing imagery. Full-spectrum rendering is slow, however, so it is conventional to use the hack of combining every wavelength near red into one value, every wavelength near green into one, and every wavelength near blue into one. It turns out that handling only RGB (Red-Green-Blue) works fairly well, since most real world light sources emit a broad spectrum, and most real world materials reflect a broad spectrum. Only in special cases, such as with simulated fluorescent lighting, is the cheat very noticeable. We use RGB BRDF functions, though we will generally refer only to a single value. Handling RGB values is essentially handling three BRDFs in parallel.

Two informal terms are commonly applied to describe the reflection properties of real world materials: diffuse reflection and specular reflection. Diffuse reflection refers to the dull reflections of most objects, what we might commonly refer to as their color. Diffuse reflection does not depend on the viewing direction. A tennis ball, for instance, possesses almost solely diffuse reflectance. It looks about the same green from any direction, under constant lighting conditions. Specular reflectance refers to the mirror-like reflections of shiny surfaces. Specular reflectance varies depending on the viewing direction, and peaks when the direction of observation equals the direction of reflection. When you try to signal someone with a mirror, you want to tilt the mirror so that their observation direction matches the reflection direction of the sunlight. Most materials have a mixture of both diffuse and specular reflection properties. The shiny hood of a yellow car is an example. The car looks yellow no matter where you stand, which is to say the car reflects yellow light diffusely. At the same time, the glint of the sun off the car's hood is only visible from a certain angle, which is to say that the hood has a bright specular reflection. The BRDF of the car has a relatively constant and low value for all combinations of directions where the outgoing direction is not close to the reflection direction. Where the outgoing direction is close to the reflection direction, the value of the BRDF would jump because of the increased amount of light passing in the reflection direction. This is a feature of shiny BRDFs called the specular peak, which will cause difficulties later on.

It is important to note that the division of reflection into diffuse and specular components is arbitrary and not based in the physical reality of light reflection. It is simply a very effective way to describe the reflection properties of most real world materials. As we will see, many BRDFs can be approximated well by attempting to fit the diffuse reflection component and the specular reflection component separately.

1.1.1 Theoretical and Measured BRDFs

A BRDF $f(\hat{w_i}, \hat{w_o})$ can be one of two types: a closed-form function or a table of measured data. Closed-form functions are generally derived from some physical theory of how light reflects from a surface. These functions can vary from the very simple to the relatively complex to the fiendishly complex. The commonly used Blinn-Phong [2] (a modification of the original Phong formulation [15]) equation is the simplest theoretical model in common usage. It is a hack and has very little basis in theory or empirical evidence, but it can passably model some materials (plastics in particular). It expresses the amount of light reflected by a surface as a three term sum:

$$f(\hat{w_i}, \hat{w_o}) = k_A + k_D(\hat{w_i} \cdot \hat{\mathbf{n}}) + k_S(\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{k_n} \tag{1.2}$$

Where $\hat{\mathbf{n}}$ is the surface normal and $\hat{\mathbf{h}}$ is the half-angle vector ($\text{norm}(\hat{w_i} + \hat{w_o})$), which lies halfway between the incident and observing directions. Both dot products are clamped to zero or above. The ambient, diffuse, specular, and specular exponent constants (k_A, k_D, k_S, k_n respectively) determine the appearance of the BRDF, and are set depending on the material being modelled.

There are many other, more theoretically and empirically reasonable models. Some of the more commonly mentioned include an early general model of reasonable complexity by Torrance and Sparrow [19], a complex model for anisotropic reflection by Poulin and Fournier [16], and more recent general models by Ward [20] and Lafortune *et al.* [11].

It is relatively easy to come up with a theoretical model for simple materials such as plastic and smooth metal, but more complex materials such as leather and velvet present a challenge. This is largely because organic materials have a very complex microstructure which is hard to simulate theoretically. In these cases actually measuring the light reflected by the material itself can be superior to creating a hideously complex closed form function to approximate it. A BRDF created in this way is called a measured BRDF.

A common method of creating a measured BRDF is to make a sphere of the material to be measured and then move a single light and camera around it, attempting to register a color for a large set of different directions. The result is a table of values, parameterized by $\hat{w_i}$ (the direction to the light) and $\hat{w_o}$ (the direction to the cam-

era). This table can then be used as a lookup during rendering. Measured BRDFs are generally difficult to use well, because of noise and other practical issues with their capture. The number of samples captured also varies widely. The data from the CUReT (Columbia-Utrecht) database contains approximately 200 samples, with both $\hat{w_i}$ and $\hat{w_o}$ spread across half of a hemisphere. Data from the Cornell graphics laboratory is much denser, with approximately 1500 samples.

1.1.2 BRDFs in a Hardware Accelerated Environment

An arbitrary BRDF, theoretical or measured, is a natural representation for a software renderer (raytracer or otherwise). In software, we can trivially find the incoming and outgoing directions from the rendered surface and can easily calculate a general function for each sample. For a hardware accelerated rasterizer such as those found in commercial graphics hardware, applying a BRDF is somewhat more difficult. Arbitrary function evaluation in hardware is only recently emerging [12] and is still quite limited. Because of its simplicity, most graphics hardware has the ability to evaluate the Phong lighting equations. Phong, however, is not satisfactory for most materials. Graphics hardware does have sophisticated lookup table support, however, in the form of texture mapping. The ability to perform fast texture lookups with bilinear filtering is standard on all modern hardware. Because of the availability of this hardware, work on hardware acceleration of more sophisticated BRDFs has focused on packing the functions into texture maps [9, 10].

Use of a texture map representation requires a BRDF in the form of a discrete function with a certain set of samples. This is an obvious representation for a measured BRDF, but theoretical models can also be used by pre-evaluating them at a range of directions to generate a set of samples. We will examine both measured and theoretical BRDFs, but we treat them both as discrete functions. The disadvantages of representing a BRDF as a discrete function are the same as for conventional texture functions. Reconstruction is a problem, as is under-sampling. Both of these difficulties, if not properly handled, can lead to visual artifacts.

1.1.3 Direction Parameterization

So far we have only referred to $\hat{w_i}$ and $\hat{w_o}$ as directions in 3-space, both lying in the hemisphere centered on a surface normal vector $\hat{\mathbf{n}}$. We will denote this hemisphere Ω, and define $\hat{w_i}, \hat{w_o} \in \Omega$. We need a compact and simple method to represent $\hat{w_i}$ and $\hat{w_o}$ as values, i.e. some map $M : \Omega \to T$, where $T \subset \mathbb{R}^n$. A direction in 3-space can be parameterized in several ways, the most obvious being as a normalized 3D vector. Manipulation of vectors is extremely easy. The vector representation is inefficient, however, since representing a direction in 3-space requires only two values.

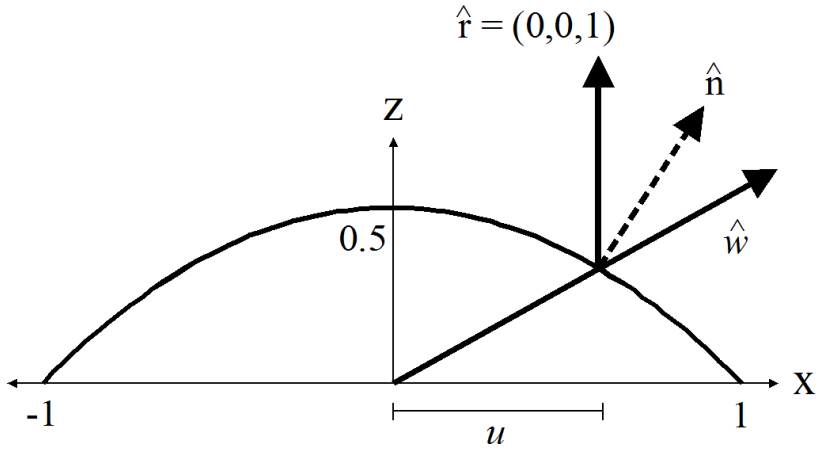

Figure 1.2: A 2D parabolic parameterization. Here $\hat{\mathbf{n}}$ is the normal, $\hat{\mathbf{r}}$ is the reflection vector, and u is one dimension of the parameterization of $\hat{w}$.

Another commonly used parameterization is spherical (elevation/azimuth) coordinates. In spherical coordinates we represent the azimuth off of an arbitrary north direction as $\theta \in [0, 2\pi)$, and elevation off of an imaginary horizon by $\phi \in [-\pi/2, \pi/2]$. While spherical coordinates are intuitive and require only two values, they have several large drawbacks. First, the mapping from spherical coordinates to directions is not one-to-one. The direction parallel to $\hat{\mathbf{n}}$ (that is, straight up) maps to $\phi = \pi/2$, for any value of θ. Second, the directions are not very evenly sampled by a naive implementation. If we evenly sample θ and ϕ, directions close to $\hat{\mathbf{n}}$ are disproportionately represented. As the elevation goes to $\pi/2$, the solid angle covered by each sample decreases towards zero.

Heidrich and Seidel [8] introduced a parameterization scheme that solves these problems, called a parabolic parameterization. The parabolic parameterization is a map $M : \Omega \to T$, where T is the unit disk. The mapping is based on the paraboloid

$$z = \frac{1}{2} - \frac{1}{2}(x^2 + y^2), x^2 + y^2 \leq 1$$

We set the positive z direction to lie along the surface normal $\hat{\mathbf{n}}$. This particular paraboloid has two convenient properties. First, a point $(x, y, z)^{\mathrm{T}}$ on the paraboloid has a (non-normalized) normal vector with the coordinates $(x, y, 1)^{\mathrm{T}}$. Second, the paraboloid reflects all rays aimed at the origin back parallel to the z-axis. Figure 1.2 shows this relationship for a 2D parameterization. Given a direction $\hat{w}$ specified as a normalized vector, the parabolic coordinates (u, v) of $\hat{w}$ are defined as $u = x$ and

$v = y$, where (x, y, z) is the point where $\hat{w}$ intersects the parabola. This point can be found by finding the normal vector at that point, then using the above relationship between the normal vector and its associated point. We know the normal lies between the direction and its reflection, so we find $\mathbf{n}$ by adding the direction $\hat{w}$ and the (constant) reflection vector $(0, 0, 1)^{\mathrm{T}}$. Secondly, we divide out the z value of $\mathbf{n}$, leaving us with (u, v) coordinates inside the unit disk. Explicitly,

$$\mathbf{n} = \begin{pmatrix} u \cdot k \\ v \cdot k \\ k \end{pmatrix} = \begin{pmatrix} \hat{w}_{\mathrm{x}} \\ \hat{w}_{\mathrm{y}} \\ \hat{w}_{\mathrm{z}} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

We can extract (u, v) easily in software. In a hardware environment we may take advantage of the divide operation in perspective correct texture hardware.

The parabolic parameterization addresses the two deficiencies of the spherical parameterization mentioned above. The mapping from directions to (u, v) points is homeomorphic, unlike the spherical mapping, and the sampling distribution is improved. For the paraboloid that Heidrich and Seidel suggest, the maximum solid angle covered by a sample is never greater than four times the minimum solid angle.

Through this paper we will use parabolic coordinates exclusively, though we will usually refer only to a direction $\hat{w}$ rather than the coordinates (u, v).

1.1.4 BRDF Data Structures

Measured BRDF data is generally collected and stored as an unsorted list of reflectance values and associated direction angles. Each entry in the list contains a color (RGB or full-spectrum) and a set of spherical coordinates for $\hat{w}_i$ and $\hat{w}_o$. For rendering, however, we must be able to make lookups of arbitrary directions quickly, and if no BRDF sample is available in the particular direction we desire, interpolate between the closest samples. For this purpose BRDFs are stored as discrete functions, represented by a table indexed by the incoming and outgoing directions. A BRDF in this representation is a 4D lookup table. As we will see, however, for hardware rendering a BRDF table usually takes the form of a set of 2D textures.

The process of placing an arbitrary measured BRDF in an evenly spaced table is difficult to perform well, however. BRDF data is often not sampled evenly and may have large holes at certain angles. Moving the data into a table generally involves a filtering step which can both erase detail and introduce aliasing. In general it is better to keep the BRDF data in the original list format as far into any algorithm as possible.

1.1.5 Formal BRDF Definition

The domain of a BRDF is the cross of two hemispheres, $\Omega \times \Omega$ or Ω^2. Using a parabolic parameterization we define a map $H : \Omega^2 \rightarrow T^2$, where $T^2 \subset \mathbb{R}^4$. A member of T^2 is a four-tuple of the form $(s, t, u, v)^{\mathrm{T}}$, where (s, t) are the parabolic coordinates of $\hat{w_i}$ and (u, v) are the parabolic coordinates of $\hat{w_o}$. We find that the BRDF function $f(\hat{w_i}, \hat{w_o})$ can be restated as $f(\hat{w_i}, \hat{w_o}) = f'(H(\hat{w_i}, \hat{w_o}))$, where $f'(H(\hat{w_i}, \hat{w_o})) : T^2 \rightarrow \mathbb{R}$. The range of the BRDF is $[0, \infty)$. For the sake of generality, we will continue to use the notation $f(\hat{w_i}, \hat{w_o})$ to represent the BRDF function.

1.2 Factorization Concept

A discrete BRDF is naturally stored in a 4D lookup table. While graphics hardware has long had 2D texture mapping capability, no hardware currently in use has the 4D texture mapping capability necessary implement this natural representation. Even if 4D texture mapping were implemented, the tables required to hold a measured BRDF would be quite large. At 4 bytes per sample (single-precision floating point), and a modest sampling rate of approximately 20 samples per radian (or a parabolic parameterization of 64x64 samples), a BRDF requires 64MB of memory. When a modern graphics accelerator board has only 64MB of total texture memory, this is a large amount of space. Conventional texture compression algorithms can reduce the space requirement, but not by more than an order of magnitude [1].

Both these difficulties can be addressed, however, by factoring a BRDF into several smaller functions. A factorization algorithm seeks to find a series of functions $f_1 \ldots f_n$ of lower dimensionality than the BRDF $f(\hat{w_i}, \hat{w_o})$, such than when combined they approximate $f(\hat{w_i}, \hat{w_o})$. Given an large enough number of factors, any discrete BRDF can be approximated to any degree of accuracy. Though theoretically the factors of our BRDF could be three, two, or one dimensional, we will be concerned here with 2D factors. Two-dimensional factors are more convenient than 3D factors, and of course require far fewer terms than 1D factors. Factoring $f(\hat{w_i}, \hat{w_o})$ into 2D functions has the additional benefit of allowing the factors to be stored in conventional 2D texture maps.

Fournier [5] introduced a method for factoring BRDFs for radiosity calculations. He introduced singular value decomposition as his method of approximation. Kautz and McCool [10] used a similar technique to accelerate hardware rendering of BRDFs. Most recently, McCool *et al.* [13] introduced a new approximation method that aims to address some of the practical shortcomings of the singular value decomposition.

The results of BRDF factorization can be surprisingly good. Theoretically the structure of a BRDF can be highly complex, but in practice most BRDFs are rather

simple. The average BRDF is highly separable - that is, it can be broken into a few terms, each of which is dependent on only a few of the parameters of the BRDF. This property naturally lends itself to a factorization approach. Only one or two 2D factors, depending on the approximation algorithm, can approximate many BRDFs to a high degree of accuracy [10]. Compared to a naive 4D representation, factorization can reduce the memory footprint of a BRDF by a factor of 100, enough to allow several BRDFs to be easily stored in the texture memory of a graphics board [13].

An example may be helpful to show the separability of a BRDF. For simplicity we again examine the Blinn-Phong model, though as mentioned above it is a hack.

$$f(\hat{w_i}, \hat{w_o}) = k_A + k_D(\hat{w_i} \cdot \hat{\mathbf{n}}) + k_S(\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{k_n}$$

Where again, $\hat{\mathbf{n}}$ is the surface normal, $\hat{\mathbf{h}}$ is the half-angle vector ($\text{norm}(\hat{w_i} + \hat{w_o})$), and the dot products are clamped to zero or above. k_A, k_D, k_S, k_n are constants. Though the whole function is 4D, each term in the Blinn-Phong model depends only on two free variables. The first term (ambient light) is simply a constant based on the lighting conditions, so it can be separated easily. If we assume that $\hat{\mathbf{n}}$ is constant (usually set to the z-axis), then the second term depends only on $\hat{w_i}$ and constants. It can thus also be separated, and represented perfectly with only two dimensions. The third term likewise depends only on $\hat{\mathbf{h}}$ and constants, so it too can be separated into a two-dimensional factor.

Blinn-Phong is an exceptionally simple function, and most sophisticated BRDFs take some more work to factor. The basic concept remains the same, however.

1.2.1 BRDF Reparameterization

The BRDF parameters are always the two directions $\hat{w_i}$ and $\hat{w_o}$, relative to the surface normal $\hat{\mathbf{n}}$. There is no reason why the table of BRDF data must be directly parameterized by these directions, however. Separability of the BRDF into factors can be markedly improved by reparameterizing the BRDF. Reparameterization is performed by applying some map $M : \Omega^2 \to \Omega^2$ before doing a BRDF lookup. In other words, instead of storing f we store f', where $f(\hat{w_i}, \hat{w_o}) = f'(M(\hat{w_i}, \hat{w_o}))$. Note that direction parameterization and BRDF parameterization are different and independent problems. One may interchange spherical and parabolic coordinates with differing BRDF parameterizations.

Rusinkiewicz [18] suggests an alternative parameterization for BRDFs to improve separability. His reparameterization is defined as

$$Z(\hat{w_i}, \hat{w_o}) = (\hat{\mathbf{h}}, \hat{\mathbf{d}}) \tag{1.3}$$

where $\hat{\mathbf{h}} = \text{norm}(\hat{w_i}, \hat{w_o})$ (the half-angle vector) and $\hat{\mathbf{d}}$ is the "difference vector", which represents the difference between $\hat{\mathbf{h}}$ and $\hat{w_i}$. $\hat{\mathbf{d}}$ obtained by transforming $\hat{w_i}$

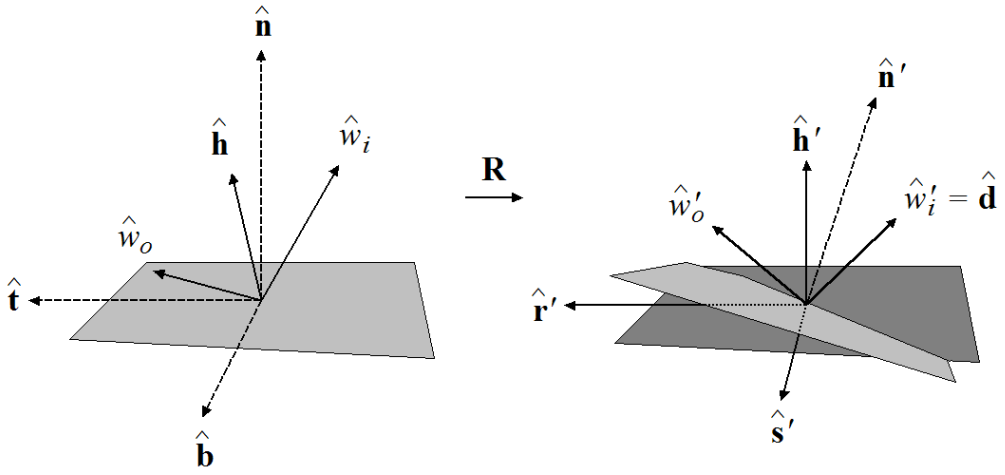

Figure 1.3: The Rusinkiewicz reparameterization, showing the transformation $\mathbf{R}$. On the left is the standard surface frame, on the right is the transformed surface frame.

into a space where $\hat{\mathbf{h}}$ lies on the z-axis. Kautz and McCool [10] note that the original transformation suggested by Rusinkiewicz is numerically unstable when $\hat{\mathbf{n}} \approx \hat{\mathbf{h}}$. They suggest a slightly different parameterization that is more stable, but may be less effective for anisotropic BRDFs. Given $\hat{\mathbf{n}}$, $\hat{\mathbf{t}}$, $\hat{\mathbf{b}}$, $\hat{w}_i$, and $\hat{w}_o$ all as normalized 3D vectors specified in the surface frame, the exact transformation is:

$$\begin{gathered}\hat{\mathbf{h}} = \text{norm}(\hat{w}_i + \hat{w}_o) \quad \hat{\mathbf{r}} = \text{norm}(\hat{\mathbf{t}} - (\hat{\mathbf{t}} \cdot \hat{\mathbf{h}})\hat{\mathbf{h}}) \quad \hat{\mathbf{s}} = \hat{\mathbf{h}} \times \hat{\mathbf{r}} \\ \mathbf{R} = \begin{pmatrix} \hat{\mathbf{r}} & \hat{\mathbf{s}} & \hat{\mathbf{h}} \end{pmatrix}^{-1} \\ \hat{\mathbf{d}} = \mathbf{R}\hat{w}_i\end{gathered} \tag{1.4}$$

The purpose of the Rusinkiewicz parameterization is to align prominent features of the BRDF (such as the specular peak) with coordinate axes. For instance, the power of the specular reflection is not directly dependent on the position of the incoming or outgoing vectors. The specular power is, however, usually directly dependent on the position of the half-angle vector. Specular reflection is normally highest when the half-angle vector is parallel to the surface normal, i.e. when the angle of incidence is very close to the angle of reflection. Using the Rusinkiewicz parameterization it is often possible to separate the specular reflection from diffuse reflection more easily than in a standard parameterization.

Though very useful in many cases, the Rusinkiewicz parameterization does not increase separability of all BRDFs. See Chapter 4.

1.3 Approximation Forms

In this section we examine two of the mathematical methods that have been previously proposed for factoring BRDFs. Unlike some other forms [9], both methods presented here assume that the BRDF is given as a discrete set of samples, not as a closed-form function. To take advantage of these methods with theoretical BRDFs, the function must be sampled prior to factorization.

1.3.1 Singular Value Decomposition

The method of singular value decomposition [17] was one of the first proposed for factoring BRDFs. There is a theorem that states that a $m \times n$ (where $m \geq n$) matrix M may be represented as $\mathsf{M} = \mathsf{USV}^{\mathrm{T}}$, where U is a $m \times n$ matrix, S is an $n \times n$ matrix, and V is an $n \times n$ matrix. The columns of U and V are orthonormal vectors. That is, for two columns $\mathbf{u}_m$, $\mathbf{u}_n$ of U, $(\mathbf{u}_m \cdot \mathbf{u}_n) = 0$ for $m \neq n$, 1 for $m = n$. S is a diagonal matrix of values w_k, $0 \leq k \leq n$, where $w_k \geq 0$. This representation is the singular value decomposition (SVD) of M, and can always be computed. The decomposition may be expressed as a sum:

$$\mathsf{M} = \sum_{k=1}^{n} w_k \mathbf{u}_k \mathbf{v}_k^{\mathrm{T}} \tag{1.5}$$

where $\mathbf{u}_k$ and $\mathbf{v}_k$ are columns of their respective matrices. The product $\mathbf{u}_k \mathbf{v}_k^{\mathrm{T}}$ is the outer product, so it yields a matrix of the same shape as M. The most interesting property of the SVD is that it yields provably optimal approximations to M, under the root mean square error metric. Truncating the above sum at some $k < n$ will give the optimal k term approximation for M.

Using the SVD to approximate a BRDF requires stuffing the BRDF $f(\hat{w}_i, \hat{w}_o)$ into the matrix M using some regular parameterization. The columns and rows of M must correspond to constant parameter values. For a normal $(\hat{w}_i, \hat{w}_o)$ parameterization, a column would correspond to a constant $\hat{w}_i$ while a row would correspond to a constant $\hat{w}_o$, or vice versa. For a Rusinkiewicz parameterization, columns and rows would correspond to constant $\hat{\mathbf{h}}$ and $\hat{\mathbf{d}}$. The SVD of the matrix is then computed, and the terms of the SVD sum ($w_k \mathbf{u}_k \mathbf{v}_k^{\mathrm{T}}$) represent the 2D factors of the BRDF, stored as texture maps. Reconstruction of a BRDF sample M_{ij} is as simple as adding the associated samples in each term of the approximation. Any desired degree of accuracy can be achieved.

The SVD approximation has several practical drawbacks, however. The first is the requirement that the rows and columns of M correspond to constant parameter values. Measured BRDF data is not always captured at constant parameter values. Each BRDF sample may have a unique $\hat{w_i}$ or $\hat{w_o}$ which does not reappear in any other sample. As a result, restricting the BRDF to an even grid while keeping the M dense may require a resampling operation. As previously mentioned, this is not always easy or desirable.

Second, the approximation terms $w_k \mathbf{u}_k \mathbf{v}_k^{\mathrm{T}}$ may contain negative numbers. Negative numbers are not handled well or at all by most texture mapping hardware, so ideally they should be avoided altogether.

1.3.2 Homeomorphic Factorization

Several of the problems with an SVD approximation can be avoided by using a modified approximation due to McCool [13]:

$$f(\hat{w_i}, \hat{w_o}) = \prod_{j=1}^{J} p_j(\pi_j(\hat{w_i}, \hat{w_o})) \tag{1.6}$$

McCool approximates the function f as a pure product of factors, rather than the SVD's sum of factors. The factors p_j are two-dimensional functions, again to be stored in texture maps. The functions $\pi_j(\hat{w_i}, \hat{w_o})$ are projections from the 4D domain of the BRDF to the 2D domain of the factors.

McCool does not solve for p_j directly. He takes the logarithm of both sides of Equation 1.6 to get an approximation for the logarithm of f (following McCool, we let $\tilde{g}(x) = \log(g(x))$ for compactness):

$$\tilde{f}(\hat{w_i}, \hat{w_o}) = \sum_{j=1}^{J} \tilde{p}_j(\pi_j(\hat{w_i}, \hat{w_o})) \tag{1.7}$$

McCool solves for $\tilde{p}_j$ in the transformed logarithmic space. Reconstruction of p_j is achieved by exponentiating $\tilde{p}_j$. The homeomorphism $\log(p_j) \leftrightarrow p_j$, for positive values of p_j, gives this approximation method its name, however McCool unfortunately uses the term homomorphism instead of homeomorphism.[1] Moving the approximation problem to the log space reduces the data fitting problem to a linear fitting problem, which is much easier to solve than the original product formulation.

[1] A *homeomorphism* is a mapping that is one-to-one and continuous in both directions [21]. This seems to be what McCool is interested in. A *homomorphism* is a term from group theory which refers to a mapping from one group to another that preserves the group operation [21]. Whether we consider $\mathbb{R}$ a group under addition or multiplication, the logarithmic transformation is not a homomorphism.

This approximation can also be easily reparameterized using the Rusinkiewicz parameterization. If $\tilde{f}'(\hat{\mathbf{h}}, \hat{\mathbf{d}})$ is the reparameterized function $\tilde{f}(\hat{w_i}, \hat{w_o})$, then the approximation is of the form:

$$\tilde{f}'(Z(\hat{w_i}, \hat{w_o})) = \sum_{j=1}^{J} \tilde{p}_j(\pi_j(Z(\hat{w_i}, \hat{w_o})) \tag{1.8}$$

where Z is the Rusinkiewicz reparameterization map. For clarity, we may write $\pi_j(\hat{\mathbf{h}}, \hat{\mathbf{d}})$ instead of $\pi_j(Z(\hat{w_i}, \hat{w_o})$.

The homeomorphic factorization solves two difficulties of the SVD. First, since p_j is obtained through exponentiation, the values of p_j will necessarily be positive. No negative values will ever reach the texture hardware. Second, the original parameterization of f can be much more flexible. Since the method is not based directly on matrix decomposition, f does not have to be resampled to fit inside a matrix. The BRDF samples can simply be stored in a large list, as long as there exists a mapping from an element of the list to its associated directions $\hat{w_i}$ and $\hat{w_o}$.

There are a few minor disadvantages of the method. If the BRDF can have the value of 0 anywhere, a small bias must be introduced to avoid negative infinity in the logarithm. The transformation to log space McCool uses is

$$\tilde{f} = \log(\frac{f + \varepsilon A}{A}) \tag{1.9}$$

and the inverse transformation is

$$f = A\exp(\tilde{f}) - \varepsilon A \tag{1.10}$$

where A is the average value of the BRDF and ε is a small fixed value, around 10^{-5}. Since the size of ε effectively determines the range of $\tilde{f}$, it can have a significant impact on the effectiveness of the approximation. In practice numerical imprecision can yield a result where $A\exp(\tilde{f}) < \varepsilon A$, so to ensure that f is positive only we omit the $-\varepsilon A$ term.

The major disadvantage of this approximation, however, is that its optimality is not guaranteed. Like the SVD approximation, given enough terms the homeomorphic approximation will give f to any degree of accuracy. Unlike the SVD, however, we can not prove that the first k terms of the homeomorphic factorization represent the best possible k term approximation of $f(\hat{w_i}, \hat{w_o})$. The speed at which the approximation converges is determined by the suitability of the projection functions π_j to the particular BRDF chosen. Some extra intelligence must be applied to determine the most effective projection functions. It is this problem that our algorithm attempts to solve.

Chapter 2

Algorithm

In the previous chapter we introduced the homeomorphic factorization scheme for BRDFs. While homeomorphic factorization can be very efficient, it relies on the suitability of projection functions for its effectiveness. In this chapter we present our algorithm, which is an extension of the homeomorphic factorization scheme. We perform a search over projection functions to find the most suitable projections for any given BRDF.

2.1 Importance of Projections

Under the SVD, the separability of the average BRDF can only be fully exploited if the proper parameterization is used [10]. The SVD can approximate certain matrices M very easily, and others only with difficultly. To get the most out of the approximation algorithm, the BRDF $f(\hat{w_i}, \hat{w_o})$ must be parameterized so that its representation in the matrix M can be easily approximated by the SVD algorithm. Kautz and McCool found that most effective parameterization for separability varied from BRDF to BRDF. Some BRDFs were easily approximated using the conventional $(\hat{w_i}, \hat{w_o})$ parameterization, while others were only successfully approximated by more exotic parameterizations, such as the Rusinkiewicz parameterization mentioned in the last chapter.

In the homeomorphic method, unlike SVD, the BRDF can simply be represented by a list of samples. By using this method we are not compelled to change the parameterization of $f(\hat{w_i}, \hat{w_o})$ from the standard $(\hat{w_i}, \hat{w_o})$ method, although we can if we so choose. The reason we can use the standard parameterization is that the problem of choosing a good parameterization has simply been shifted to the problem of choosing good projections. Flexible projection functions can do much the same work as reparameterizing the BRDF.

We can apply some theoretical analysis to choosing projections. We know that a 4D BRDF has more information than we can easily store in a short series of 2D textures. Usually, however, there are a small number of salient features in each BRDF. For example, one may have a red diffuse color and a moderately shiny surface. Given just those two statements, it is easy to approximate that BRDF for any set of directions. We hope to design projections that will capture such salient features and allow them to be stored in our 2D factors.

2.1.1 Initial Choices

We begin by looking at the projections that McCool initially chooses. Though the homeomorphic approximation can use an arbitrary number of terms, in practice McCool only uses three. The projections that he chooses are

$$\pi_1(\hat{w}_i, \hat{w}_o) = \hat{w}_i$$
$$\pi_2(\hat{w}_i, \hat{w}_o) = \hat{\mathbf{h}}$$
$$\pi_3(\hat{w}_i, \hat{w}_o) = \hat{w}_o$$

where $\hat{\mathbf{h}}$ is the vector midway between $\hat{w}_i$ and $\hat{w}_o$, called the half-angle vector. It is defined as

$$\hat{\mathbf{h}} = \text{norm}(\hat{w}_i + \hat{w}_o)$$

These projections are designed to capture the diffuse and specular components of a BRDF in separate terms. As we have seen, many BRDFs have a more or less constant diffuse color and a very bright but localized specular peak. The amount of specular reflectance can often be computed from just the half-angle vector $\hat{\mathbf{h}}$. This is because when the half angle vector is near the normal of the surface, the outgoing direction is near the reflection direction. Since the strength of the specular reflection often depends largely on how near the outgoing direction is to the reflection direction, the specular reflection may be nicely captured by the half-angle projection.

The diffuse color of the BRDF (if it varies at all) can often depend on the angle of the incident light, which is captured by the first projection.

Lastly, McCool chooses his projections to enforce reciprocity of the BRDF approximation. A BRDF satisfies reciprocity if and only if

$$\forall \hat{w}_i, \hat{w}_o, f(\hat{w}_i, \hat{w}_o) = f(\hat{w}_o, \hat{w}_i)$$

Reciprocity is an important characteristic of BRDFs that model physical materials. If a BRDF satisfies reciprocity, then it satisfies the physical law of reflecting wave phenomenon that reversing the direction of the wave does not affect the power or direction of reflection. It also ensures that the amounts of energy entering and leaving

the material are equal. Enforcing reciprocity in a homeomorphic factorization requires adjusting both the projections and texture functions. See Section 3.2.3 for more detail.

McCool's projections work well for a wide range of BRDFs. However he notes that there may be superior BRDFs that would allow greater accuracy or fewer terms for the approximation. While he offers a few more for consideration, there is little reason to believe that his hand-picked projections represent the best possible projections for every BRDF. It is our goal to create an algorithm that will attempt to find the best projections for any given BRDF.

2.2 Our Approach

We start by suggesting that the best projections for a given BRDF may not be intuitive or easy for a human to find. Even if the basic shape of good projections can be known a priori, the small nuances of BRDF data suggest that adapting good projections slightly may help reduce approximation error. The way we propose to find these superior, tailored projections is by an automatic search through a space of projections.

Our algorithm has an inner and outer loop. The inner loop is essentially the homeomorphic approximation algorithm of McCool, slightly generalized. The inner loop takes a set of projections $\pi_1 \dots \pi_J$ and finds the set of texture factors $p_1 \dots p_J$ that, when combined with the projections, best approximates the BRDF. The loop consists of solving a linear system by an iterative method, explained in the next chapter.

Since the inner loop is deterministic, we can associate an approximation error with each set of projections. The outer loop attempts to find the set of projections with the smallest approximation error. There are at least several ways to go about this task. The first and potentially the most accurate is to pick a number of terms J and then simultaneously search for J projections, executing the inner loop once for each set of J we wish to test. This method is rather unwieldy, however, as the number of dimensions in our search space increase with each projection. The method that we have implemented is a faster and simpler, but potentially less accurate, greedy method. The basic algorithm is shown by the following pseudocode:

```
ApproximateBRDF
    for j from 1 to J
        make guess for projection j
        do
            find best homeomorphic approx. for projections 1 .. j
            refine guess for projection j
```

```
        while (projection j can be improved)
        save projection j
    endfor
```

In other words, we search for one projection at a time, building up the J projections incrementally. Our outer loop starts on π_1, and attempts to find the best one term approximation of the BRDF. Once that is found we move on to π_2. The algorithm attempts to find the best two term approximation it can by keeping π_1 constant and only varying π_2. For π_i, the algorithm attempts to find the best i term approximation it can while varying π_i and keeping $\pi_1 \dots \pi_{i-1}$ constant. Thus the outer "loop" actually consists of two nested loops, one over the the projections from $\pi_1 \dots \pi_J$ and another over each projection guess π_j. This greedy algorithm actually works rather well in practice, perhaps because the features of a BRDF can often be broken down easily by importance. Commonly, we find that the diffuse component is roughly approximated first, followed by a rough specular component, and then followed by refinements.

To allow this algorithm to work, however, we must have a way of searching the space of projections. In the next section we discuss the goals and implementation of our projection search space representation.

2.3 Projection Search Space

In the general form of the homeomorphic factorization, the projections π_j can be arbitrary functions from $\mathbb{R}^4$ to $\mathbb{R}^2$. This gives excellent flexibility to match the characteristics of particular BRDFs. In addition to the simple $\hat{w_i}$ and $\hat{w_o}$ projections, for example, McCool suggests several projection functions based on specific vector calculations. These include the aforementioned $\pi(\hat{w_i}, \hat{w_o}) = \hat{\mathbf{h}}$, the half-angle projection, and the more sophisticated $\pi(\hat{w_i}, \hat{w_o}) = \hat{w_i} \cdot \hat{\mathbf{h}}$ or $\hat{w_o} \cdot \hat{\mathbf{h}}$, projections intended to capture Fresnel effects. If we are to search for optimal projections, however, we must be able to manipulate projections easily. Since there is no simple way to work with all possible functions $\pi : \mathbb{R}^4 \to \mathbb{R}^2$, we must settle for a subset of all projections for the search space of our algorithm.

Our search space of projections $\mathcal{P}$ should satisfy a few basic conditions. An element $p \in \mathcal{P}$ should be easily defined with as few values as possible. Most search algorithms are designed to operate in a vector space, so $\mathcal{P}$ should have as many qualities of a vector space as possible. In particular, there should be an easy method of quantifying the difference between $p, q \in \mathcal{P}$, and some idea of a direction from p to q. Finally, since we want to waste as little computation time as possible, we want $\mathcal{P}$ to contain only projections that could be reasonably used in a homeomorphic factorization.

2.3.1 Search Space Restriction

The most obvious candidate for $\mathcal{P}$ is the space of linear projections $p : \mathbb{R}^4 \to \mathbb{R}^2$, call it $\mathcal{P}_{\mathrm{L}}$. A linear projection $p \in \mathcal{P}_{\mathrm{L}}$ can be expressed as a 2×4 matrix. $\mathcal{P}_{\mathrm{L}}$ is a very easy space to search, however it does not satisfy one of the most important goals for our search space: most of the elements of $\mathcal{P}_{\mathrm{L}}$ are useless for the homeomorphic approximation. We need to restrict the $\mathcal{P}$ to a space smaller than $\mathcal{P}_{\mathrm{L}}$ to achieve reasonable efficiency. To see why this is we need to recall the role of a projection in the approximation algorithm.

The job of a general projection $p(\hat{w_i}, \hat{w_o})$ is to project the domain T^2 to our texture domain, which is defined as the square centered on the origin with sides of length 2. More precisely, the texture domain $D = \{(u, v) : (u, v) \in \mathbb{R}^2, u \in [-1, 1], v \in [-1, 1]\}$. Recall that a member of T^2 is a four-tuple of the form $(s, t, u, v)^{\mathrm{T}}$, where (s, t) are the parabolic coordinates of $\hat{w_i}$ and (u, v) are the parabolic coordinates of $\hat{w_o}$.

The shape of T^2 is not easily defined, as it is something between a 4-sphere and 4-cube. We use a 4-cube as a conservative bound on T^2. The image of the 4-cube under p[1] must fall completely inside the square D for any projection that is to be used in an approximation. Additionally, for the purposes of reducing approximation error, we want the image of the 4-cube to fill as much of the square as possible.

To accommodate this restriction we limit our projection space to normalized projections. Formally, if $\mathbf{x}$ and $\mathbf{y}$ are the two rows of the matrix representation, $\|\mathbf{x}\| = \|\mathbf{y}\| = 1$. We also add a scaling factor to each projection to put the image of the 4-cube exactly inside the square.

As a final restriction we limit $\mathcal{P}$ to orthogonal projections, i.e. for matrix rows $\mathbf{x}$ and $\mathbf{y}$, $\mathbf{x} \cdot \mathbf{y} = 0$. Under an orthogonal projection, the image of the 4-cube will cover the largest area possible inside the texture domain. As we will see, this restriction is also convenient for our final representation. Restriction to orthogonal projections does not limit the effectiveness of the algorithm, as any projection $p \in \mathcal{P}_{\mathrm{L}}$ can be represented by an orthogonal projection q followed by a linear transformation (see Appendix A). This means that the image of the 4-cube under p is the same as under q to within a linear transformation. Since only the points that lie in the image, not the shape of the image, are important for us, any linear projection can be duplicated by an orthogonal projection.

Our final search space $\mathcal{P}$ is the space of orthonormal linear projections from $\mathbb{R}^4$ to $\mathbb{R}^2$.

We lose some flexibility by restricting ourselves to linear projections. McCool's results seem to rely heavily on the effectiveness of the half-angle projection ($\pi(\hat{w_i}, \hat{w_o}) =$

[1]The image of a projection is also called the range of the projection. When we refer to the image of the 4-cube under p, we are referring to the set of projected points obtained by projecting each point inside the 4-cube.

$\hat{\mathbf{h}}$), which is not a linear projection, as least under the standard $(\hat{w_i}, \hat{w_o})$ parameterization and the parabolic mapping H from directions to values[7].

We can make up for some of our lost flexibility by reparameterizing the BRDF with the Rusinkiewicz parameterization $Z(\hat{w_i}, \hat{w_o}) = (\hat{\mathbf{h}}, \hat{\mathbf{d}})$. Under Z, our projections take the form $\pi(\hat{\mathbf{h}}, \hat{\mathbf{d}})$ instead of $\pi(\hat{w_i}, \hat{w_o})$. In this form, the half-angle projection is trivially linear. In Chapter 4 we discuss the effects that R has on approximation accuracy. Overall, our results indicate that the restriction to linear projections does not limit the accuracy of the approximation significantly.

Projection Representation

The space of unrestricted linear projections $\mathcal{P}_{\mathrm{L}}$ has eight dimensions. Our space $\mathcal{P}$ is a subspace of $\mathcal{P}_{\mathrm{L}}$, so intuitively it should have fewer than eight dimensions. In fact it has five, which is shown by the following argument: a member of $\mathcal{P}$ can be described by two orthonormal 4D vectors. Specifying one normalized 4D vector requires three values. Once the first vector is specified, the second vector must lie in the 3D space orthogonal to the first vector. Specifying a normalized vector in this 3D space requires two values, for a total of five for both vectors.

We want a representation for $p \in \mathcal{P}$ that comes as close as possible to this theoretical limit of five dimensions. This different representation is only for the sake of the search algorithm. When we finally apply a projection, we will always convert back to the 2×4 matrix representation. We might also use a 2×4 matrix for our search representation, but at eight values per matrix it over specifies p by three dimensions. Besides the obvious inefficiency, not all matrices represent elements of $\mathcal{P}$. If a search algorithm attempts to search over the space of 2×4 matrices ($\mathcal{P}_{\mathrm{L}}$), it must make an additional step of somehow projecting the matrix onto the manifold of $\mathcal{P}$. This projection is necessarily somewhat arbitrary, and introduces difficulties where the direction of search lies off of the manifold.

Another possible approach is to represent p directly as a vector in $\mathbb{R}^5$. This opens up the problem of how to parameterize p in five dimensions, and the problem of converting the vector into a 2×4 matrix. There does not seem to be a solution that is both intuitive and simple enough for a search algorithm to use successfully.

Instead of either of the two above approaches, we chose a representation consisting of a 4D rotation and two basis vectors. The motivation for this representation is as follows. A orthogonal projection $p \in \mathcal{P}$ can be represented as a pair of 4D orthonormal vectors, $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$. These vectors form the rows of the 2×4 matrix form of the projection. We can also think of the two vectors as defining a 2D plane in 4D space. The action of the projection p is to orthogonally project 4D points onto the surface of the plane defined by $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$. We can relate a projection p with basis vectors $\hat{\mathbf{x}}_p$ and $\hat{\mathbf{y}}_p$ to another projection q with basis vectors $\hat{\mathbf{x}}_q$ and $\hat{\mathbf{y}}_q$ by using a 4D rotation

R:

$$\hat{\mathbf{x}}_q = R\hat{\mathbf{x}}_p$$
$$\hat{\mathbf{y}}_q = R\hat{\mathbf{y}}_p$$

The space $\mathcal{P}$ is closed under the operation of rotation, because rotation preserves orthogonality and scale. Thus we know that q is a member of $\mathcal{P}$, for all $p \in \mathcal{P}$ and rotations R. Using this information, we can represent any projection in $\mathcal{P}$ as a set of arbitrary orthonormal vectors, for example $(1,0,0,0)^{\mathrm{T}}$ and $(0,1,0,0)^{\mathrm{T}}$, and a 4D rotation. It is simple to show that we can in fact generate all $q \in \mathcal{P}$: we simply solve for R in the equations above. Using this representation we can leave the basis vectors fixed and only change the rotation, and obtain every possible projection in $\mathcal{P}$.

Using the rotation representation, the number of dimensions of our search space is equal to the degrees of freedom of a 4D rotation, which is six. A 3D rotation possesses three degrees of freedom, one for each axis-aligned plane: the XY, XZ, and YZ planes. A 4D rotation similarly possesses a degree of freedom for each axis-aligned plane in 4D: the XY, XZ, XW, YZ, YW, and ZW planes.

Our rotation representation thus has six dimensions, which is one more than the theoretical optimum. The extra dimension corresponds to the rotation parallel to the plane that lies orthogonal to the plane of projection, as defined by the projection's two basis vectors. As such it has no effect on the properties of the projection. This extra dimension can then be easily ignored by a minimization search algorithm. In exchange for this slight inefficiency, we reap the benefits of an intuitive and relatively simple to parameterize representation.

Rotation Representation

There remains the problem of how to represent a 4D rotation. The conventional representation for a 4D rotation R is as a 4x4 matrix $\mathbf{R}$. A rotation matrix is a matrix that satisfies two conditions. It must be orthogonal, i.e.

$$\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$$

and it must have determinant = 1. The rotation $\mathbf{R}$ is applied to a vector $\mathbf{v}$ by simple matrix-vector multiplication. The main problem with a matrix representation is that it over specifies the rotation. A 4D rotation has only six degrees of freedom, but a 4x4 matrix has 16 values. It would be possible to deal with matrices entirely, however we can make a much cleaner implementation using quaternions.

A quaternion is a 4-tuple $\mathbf{Q} = (w,x,y,z)^{\mathrm{T}}$ that obeys special rules for manipulation. Quaternions are analogous to vectors in several ways. The quaternion norm is equivalent to the vector norm:

$$\|\mathbf{Q}\| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

A unit quaternion is analogous to a unit vector, $\|\hat{\mathbf{Q}}\| = 1$. The basic operations on quaternions are quaternion multiplication (denoted $\mathbf{Q}_1\mathbf{Q}_2$) and quaternion power (denoted $\mathbf{Q}^x$). These operations roughly parallel vector addition and vector-scalar multiplication, except that quaternion multiplication is not commutative: $\mathbf{Q}_1\mathbf{Q}_2 \neq \mathbf{Q}_2\mathbf{Q}_1$. These restrictions will become very important later when we attempt to search through a quaternion space. Quaternions also have an identity $\mathbf{I}$ under multiplication and power:

$$\mathbf{I} = (1, 0, 0, 0)^{\mathrm{T}}$$
$$\mathbf{QI} = \mathbf{IQ} = \mathbf{Q}$$
$$\mathbf{I}^x = \mathbf{I}$$

In graphics, rotations in 3D are commonly represented as unit quaternions. The quaternion multiplication operator represents the composition of two rotations. The quaternion power operator corresponds to rotation power: $\mathbf{Q}^2$ doubles the effect of $\mathbf{Q}$, spinning vectors twice as far around the axis of rotation. When considered as a rotation, a quaternion is usually represented as a scalar value and a vector[6]:

$$\hat{\mathbf{Q}} = (\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\mathbf{v})^{\mathrm{T}}$$

where $\mathbf{v} = (x, y, z)$. This formulation represents a right-handed rotation of θ radians around the axis parallel to $\mathbf{v}$. To apply the rotation to a 3D vector $\mathbf{u}$, we create a 4D vector $\mathbf{v} = (0, \mathbf{u})^{\mathrm{T}}$. The rotation is applied using the following formula:

$$\mathbf{v}' = \hat{\mathbf{Q}}\mathbf{v}\hat{\mathbf{Q}}^{-1}$$

Vector $\mathbf{v}$ is treated as a quaternion, and $\mathbf{v}' = (0, \mathbf{u}')^{\mathrm{T}}$, where $\mathbf{u}'$ is the rotated 3D vector. Since quaternion multiplication is not commutative, this operation is not trivial. The correctness of quaternion rotation is not simple to prove, and has been proven elsewhere [3], so we will not repeat it.

As in 3D, we can represent 4D rotations with both matrices and quaternions. In fact, the application of quaternions in 3D is somewhat ad-hoc because it is really a special case of 4D rotation. A theorem going back at least to 1901 [14] claims that an arbitrary 4D rotation R may be represented by the left and right multiplication of unit quaternions, $\hat{\mathbf{S}}$ and $\hat{\mathbf{T}}$. Given a 4D vector $\mathbf{v}$, we can apply the 4D rotation R to $\mathbf{v}$ as follows:

$$\mathbf{v}' = \hat{\mathbf{S}}\mathbf{v}\hat{\mathbf{T}}$$

Here as before $\mathbf{v}$ is treated as a quaternion. The proof again is not repeated.

This is quite a useful theorem for us. The quaternion representation requires us to store and manipulate eight values, only two more than the theoretical limit of six.

We will sometimes denote a 4D rotation as a duple of unit quaternions, for example:

$$R = [\hat{\mathbf{S}}, \hat{\mathbf{T}}] = [(0.707, 0, 0.707, 0)^{\mathrm{T}}, (0, 0, 1, 0)^{\mathrm{T}}]$$

The quaternion representation also gives us a simple way to implement 4D rotation operations. The power operation is especially simpler using quaternions than matrices. Rotation operations composition and power for 4D rotations in the quaternion representation can be defined as follows. The composition operation $R' = R_1 R_2$ is defined as:

$$\hat{\mathbf{S}}' = \hat{\mathbf{S}}_1 \hat{\mathbf{S}}_2$$
$$\hat{\mathbf{T}}' = \hat{\mathbf{T}}_2 \hat{\mathbf{T}}_1$$

the power operator $R' = R^x$ is defined as:

$$\hat{\mathbf{S}}' = \hat{\mathbf{S}}^x$$
$$\hat{\mathbf{T}}' = \hat{\mathbf{T}}^x$$

and therefore the inverse $R' = R^{-1}$ is:

$$\hat{\mathbf{S}}' = \hat{\mathbf{S}}^{-1}$$
$$\hat{\mathbf{T}}' = \hat{\mathbf{T}}^{-1}$$

and finally the identity rotation I is defined as:

$$\hat{\mathbf{S}} = \mathbf{I}$$
$$\hat{\mathbf{T}} = \mathbf{I}$$

2.3.2 Conclusion

We have reduced the original search over $\mathcal{P}$ to a search over the space of two unit quaternions (call it $\mathcal{Q}^2$). $\mathcal{Q}^2$ has only six dimensions, so it is almost as concise a space as possible. It also maps entirely onto the space $\mathcal{P}$. $\mathcal{Q}^2$ is a troublesome space to search through, since quaternions are not linear. Quaternions are well behaved enough to make searching them possible, however, as we will see in the next chapter.

Chapter 3

Implementation

In this chapter we discuss the implementation of the algorithm we have broadly outlined. Our algorithm has two main sections: a outer loop consisting of a search over the projection space $\mathcal{P}$, and an inner loop consisting of the factorization computation. Our inner loop is evaluated with a modified version of McCool's homeomorphic factorization scheme, presented in Chapter 1. The outer loop is an adaptation of previous search procedures to our own projection search space.

3.1 Search Algorithm

The outer loop of our algorithm is the search procedure over all projections in $\mathcal{P}$. The task of the search algorithm is to find the set of projections $\pi_1, \ldots, \pi_J$ that minimize some approximation error function. A set of projections uniquely determines a homeomorphic approximation, which in turn has an approximation error. As a result we can assign one and only one error value to each set of projections. The error function that we use is not critical, and can be tailored to subjective criteria. The precise error computation is covered in Section 3.1.4.

Our search algorithm should satisfy some basic qualifications. First, it should fit the quaternion space $\mathcal{Q}^2$ easily. Quaternion combination is not linear, so interpolating between more than two quaternions is troublesome. Any algorithm that requires interpolation between more than two points in the space is a poor match for $\mathcal{Q}^2$, and it is advantageous if the problem can be avoided. This rules out several multi-dimensional search algorithms that require this operation. In general, algorithms dealing with simplices (in an N dimensional space, a simplex is a set of $N+1$ points) often require interpolation between the N points.

Our search algorithm should also require as few function evaluations as possible. Each function evaluation is very costly. A function evaluation represents one full eval-

uation of our inner loop, and the inner loop is a full homeomorphic approximation computation. It is advantageous in this case to allow some increased complexity in the outer loop in order to secure fewer evaluations of the inner loop. This requirement tends to make simulated annealing and its family of randomized algorithms less appealing. While they fit $\mathcal{Q}^2$ well, they usually require a large number of function evaluations to achieve a solution.

Additionally, while our search space has six dimensions, it has a relatively small number of local minima. When running our algorithm on random guesses, we usually find between one and five distinct minima per term. For these reasons, we chose a greedy algorithm that will aggressively search for minima. We can perform several random restarts to make it more likely that we reach the global minimum.

3.1.1 Direction Set Methods

There is a class of greedy multi-dimensional search algorithms that do fit a quaternion space well, called direction set methods [17]. Direction set methods are essentially hill-climbing methods, with all the associated advantages and disadvantages. The search space is assumed to have N dimensions, and a function $f : \mathbb{R}^N \to \mathbb{R}$ (or in our case, $f : \mathcal{Q}^2 \to \mathbb{R}$) to be minimized. A direction set algorithm maintains a set of directions $d_1, \ldots, d_N$ in the search space. Each direction defines a line, along which the algorithm will attempt to find minima. The only qualification on the directions is that they be linearly independent.

During execution, the algorithm picks a starting position P and a set of directions, either randomly or by some more sophisticated heuristic. The algorithm then begins cycling through the direction set. For each direction d_i, the algorithm runs a line minimization routine. The line minimization finds a minimum (either local or global, depending on implementation) along the line. The position P is then moved to the minimum, and the direction d_i+1 is checked in the same way. The algorithm continues to cycle through its directions, repeating the first direction once it finishes with the last. The algorithm terminates when none of the directions in its direction set yields any improvement in the function.

Most direction set algorithms (including Powell's method, our choice) provide for changing their direction set to adapt to the contours of the function. For example, the algorithm may find itself in a long valley that does not parallel any of the directions in its set. In this case, Powell's method will use a heuristic to attempt to pick a direction of movement directly down the valley. The following pseudocode shows the algorithm. The heuristics used are slightly more complex than shown, but the overall shape of the algorithm in the same:

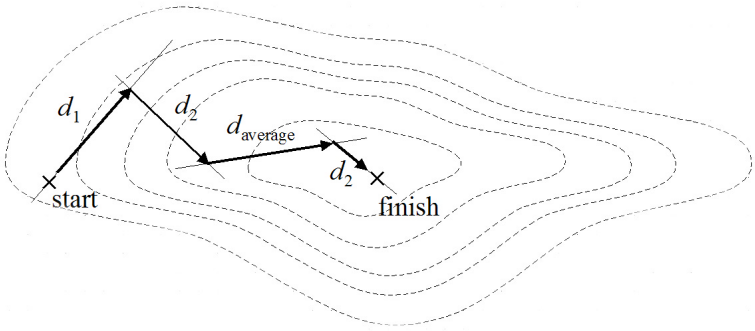

Figure 3.1: Two full iterations of Powell's method in 2D. Dotted lines are contours of the error function. d_1 and d_2 are the initial directions. d_1 is replaced by d_{average} in the second iteration.

```
DirectionSetMinimization
    pick position P;
    while currenterror < lasterror do
        // keep going until we cannot improve anymore
        save Pold = P;
        foreach direction d
                P = minimum along line through P parallel d;
        endfor;
        calculate P - Pold, the average direction moved this iteration;
        check error along average direction line;
        if better than currenterror
            put average direction in the set;
            take last used direction out;
    endwhile;
    return P;        // P is located at a (local) minimum
```

An illustration of several iterations of a 2D algorithm is shown in figure 3.1.1.

A direction set method is a natural search algorithm for a quaternion space, because the basic operation is movement along a line. Movement along a line is one of the few operations that is easy to adapt to a quaternion space. In a vector space, a line is specified as a point vector and a direction vector. The line is parameterized by a scalar value t: a point on the line is specified as $\mathbf{p}(t) = \mathbf{x} + t\mathbf{y}$.

In our case, we have 4D rotations (pairs of quaternions) instead of vectors. We represent both "points" and "directions" with rotations. A rotation can specify an

orientation in space relative to a standard basis (a "point" in $\mathcal{Q}^2$). A rotation can also represent a direction to rotate in, where movement in that direction is achieved by raising the rotation to a power. A "line" in our space is parameterized much like one in a vector space. For a scalar t, $R(t) = (D)^t R_s$, where $R(t)$ and R_s are rotations interpreted as orientations, and D is a rotation interpreted as a direction. In other words, movement along the line is equated to changing the power of D.

Equating a line with a rotation means that our error function f will be periodic with t (period 4π). We count on our line minimization routine to handle the periodicity effectively, which is not difficult. A minimum at t translates to the same quaternions as $t + 4\pi k$, for integral k, so there is no reason for t to ever leave a 4π range.

Our algorithm selects new directions according to the adapted Powell's method implementation in Numerical Recipes [17]. The concept is this: after cycling through the directions once, the algorithm finds the total direction moved from the start point. If R_s is the starting orientation, and R_f is the final orientation, the direction moved is $D = R_f(R_s)^{-1}$. The algorithm then tries moving in this new direction. If D is useful, i.e. the error function decreases in direction D, then D replaces the last direction in the direction set and the algorithm continues. After several iterations, the initial guess directions may be completely replaced.

The initial directions we give to the algorithm are six basis rotations:

$$
\begin{aligned}
d_1 &= [(0,1,0,0)^{\mathrm{T}}, (1,0,0,0)^{\mathrm{T}}] \\
d_2 &= [(0,0,1,0)^{\mathrm{T}}, (1,0,0,0)^{\mathrm{T}}] \\
d_3 &= [(0,0,0,1)^{\mathrm{T}}, (1,0,0,0)^{\mathrm{T}}] \\
d_4 &= [(1,0,0,0)^{\mathrm{T}}, (0,1,0,0)^{\mathrm{T}}] \\
d_5 &= [(1,0,0,0)^{\mathrm{T}}, (0,0,1,0)^{\mathrm{T}}] \\
d_6 &= [(1,0,0,0)^{\mathrm{T}}, (0,0,0,1)^{\mathrm{T}}]
\end{aligned}
$$

We make no claim to as to the suitability of these starting directions, other than that they span the space $\mathcal{Q}^2$. As long as the starting guesses satisfy that basic condition, Powell's method will find more appropriate directions during its execution.

We have found that Powell's method is efficient for searching $\mathcal{Q}^2$. On average, the algorithm will converge to a minimum in between two and four iterations through the six directions (see Chapter 4). It may be possible to improve this number by more intelligent selection of directions, but overall the number of function evaluations is reasonable.

3.1.2 Line Minimization

For our line minimization algorithm we could use any reasonably efficient one dimensional minimization algorithm. Powell's method only finds local minima regardless of our line minimization algorithm, so there is little reason to choose a global line minimization routine. We can again choose from the greedy hill-climbing set of algorithms for efficiency. For simplicity of implementation we use the Golden Section search [17]. Our one dimensional minimization problem is defined along the line $R(t) = (D)^t R_s$. Here t is our one dimension. We define a one dimensional error function $g(t) = f((D)^t R_s)$, where f is our original error function over all six dimensions.

Golden section search begins with a bracket of parameters (a, b, c), such that $a < b < c$, $g(b) < g(a)$, and $g(b) < g(c)$. We are thus assured that there is a minimum between a and c. The search proceeds by progressively shrinking the bracket, always maintaining the relationship between, a, b, and c. The algorithm gets its name from the amount that bracket is shrunk each step, which is the Golden ratio. Golden section search is fairly efficient, though we could probably improve performance by using a more sophisticated predicting algorithm.

3.1.3 Projection Guesses

As with any minimization problem, guessing well at the beginning can increase our method's efficiency by a large margin. Our method allows us to give the search algorithm initial guesses for each projection π_j. Some good initial guesses include those mentioned in Section 2.1, although we cannot use the half-angle projection $\pi = \hat{\mathbf{h}}$ directly unless we use the Rusinkiewicz parameterization. The most basic projections, $\pi = \hat{w}_i$ and $\pi = \hat{w}_o$, are generally useful guesses, as is $\pi = \hat{w}_i + \hat{w}_o$. In our rotation representation, $\pi = \hat{w}_i$ is represented by the basis vectors $(1, 0, 0, 0)^{\mathrm{T}}, (0, 1, 0, 0)^{\mathrm{T}}$ and the identity rotation. Similarly, $\pi = \hat{w}_0$ is represented by the basis vectors $(0, 0, 1, 0)^{\mathrm{T}}, (0, 0, 0, 1)^{\mathrm{T}}$ and the identity rotation.

Part of the purpose of searching the projection space, however, is to alleviate the need to think up good projection guesses. Thus we often simply use random guesses. A random guess is formulated by fixing our basis vectors and creating a random rotation by choosing two random unit quaternions. Since there are relatively few local minima in our error function f, this approach works fairly well. Only a small number of restarts are usually needed to find a good approximation.

3.1.4 Error Calculation

Unlike the inner loop, which minimizes error depending on the linear system solver used, the outer loop may minimize whatever error function we like, as long as it is

of the form $f : \mathcal{Q}^2 \to \mathbb{R}$. The algorithm is most effective, of course, when the outer loop attempts to minimize the same error metric as the inner loop. Our inner loop minimizes the average squared error per sample of the transformed BRDF function $\tilde{f}(\hat{w_i}, \hat{w_o})$. Because we transform the BRDF into log space before we run the inner loop, however, we are in effect minimizing the relative error for each sample. That is, the larger the value of a sample in the untransformed space, the more absolute error the linear system solver will tolerate for that sample.

The human eye seems to be more sensitive to relative error than absolute error, so subjectively relative error minimization is usually superior. In some cases, however, it appears that the average absolute error is a subjectively superior metric. We can use either or a mixture of both for our error function f. A blended function (for example, $f = |relative| + |absolute|$) can also be effective. More experimentation would be useful in determining the best error function. Generally, however, we use the average squared relative error per sample.

The error function is computed by iterating over all samples in the BRDF, computing their corresponding values in the approximation, and taking the square or absolute value of the difference.

3.2 Homeomorphic Factorization

In this section we discuss the implementation of the homeomorphic factorization scheme. Our method largely follows the algorithm laid out by McCool, except that we present the factorization algorithm in greater generality. Recall Equation 1.7, which expresses an approximation for the transformed BRDF function as a sum of transformed factors and projections:

$$\tilde{f}(\hat{w_i}, \hat{w_o}) = \sum_{j=1}^{J} \tilde{p}_j(\pi_j(\hat{w_i}, \hat{w_o}))$$

Given set of projections $\pi_1 \ldots \pi_J$ (where J is an arbitrary number of terms), the factorization algorithm determines the best fitting factor textures $\tilde{p}_1 \ldots \tilde{p}_J$. Our inner loop is deterministic, so each set of projections specifies a single set of factors $\tilde{p}_1 \ldots \tilde{p}_J$ and thus a single factorization.

The task of the inner loop can be intuitively understood as a 3D data-fitting problem. A projection π_k projects BRDF samples onto the texture domain D, which is defined as the set of points (u, v) such that $u \in [-1, 1]$ and $v \in [-1, 1]$. These projected samples can be thought of as forming a 3D point cloud, with each sample having 3D coordinates (u, v, w). Here u and v are the 2D projected position of the sample as above, and w is the transformed BRDF value of the sample. We chose

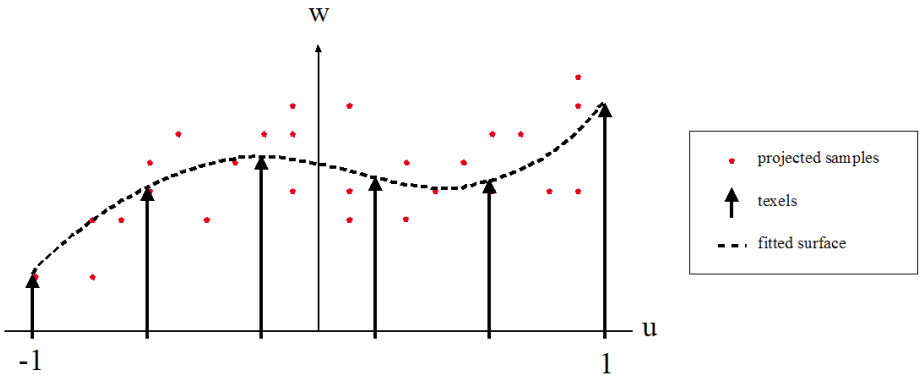

Figure 3.2: A slice of the fitting problem along the u axis. The w axis is the sample value axis. Small dots are projected sample points, while vertical lines are texels.

the point cloud analogy rather than a continuous function because more than one projected sample may share (u, v) coordinates.

Consider a single term approximation. The inner loop must find the texture function $\tilde{p}_1(u, v)$ over D that best fits the samples of the BRDF . This operation can be thought of as finding the continuous surface that best fits the 3D point cloud described above. The surface will necessarily be an approximation, since no one-to-one function can generally approximate a point cloud. The discrete texture function $\tilde{p}_k$ represents an even sampling of the continuous approximating surface. For convenience, we will use the common term texel (for "texture element") to refer to a data point of $\tilde{p}_k$. The fitting operation along one dimension (u) is visualized in figure 3.2. In this visualization, the approximation error of each sample is the vertical distance between the sample and the surface.

We can picture an approximating with J terms as solving J surface fitting problems in parallel. Each of the J projections defines a different point cloud, and a different fitting surface. The final approximated value of a sample is the sum of its corresponding values on the J surfaces. By simultaneously solving all J terms, we hope to find the set of surfaces that minimizes the total approximation error for each sample.

There are two separate problems that must be solved in this method. First is how to approximate the sample points with a surface. Second is how to approximate the surface with texels, i.e. how to resample it. In practice, however, our surface representation is the texture itself, so we in effect skip the intermediate surface and

directly relate the sample points to the texels. The inner loop algorithm solves for the best possible $\tilde{p}_k$ by setting up a linear system relating the projected BRDF samples to the texel values of $\tilde{p}_k$. The relation between samples and texels is created by a reconstruction filter, in our case a simple bilinear filter. Once the linear system is set up, the algorithm uses standard linear solving techniques to find the solution that best approximates all BRDF samples.

3.2.1 Linear System Setup

We first examine the setup of the linear system for a single term approximation. Let $\mathbf{f}$ be a vector of size N containing the transformed samples of $\tilde{f}(\hat{w}_i, \hat{w}_o) = \log f(\hat{w}_i, \hat{w}_o)$. The order of the samples is not important, as long as we maintain some mapping from the an element $\mathbf{f}[k]$ to its associated directions $\hat{w}_i$ and $\hat{w}_o$. Let $\mathbf{x}$ be a vector representing the texel values of the texture function $\tilde{p}$. The size of $\mathbf{x}$ is the total number of texels under the image of the projection π (see Section 3.2.4). The number of texels in $\mathbf{x}$ will also vary with size of the texture $\tilde{p}$, which may be user specified. Order is again unimportant, but we maintain a mapping from a texel $\mathbf{x}[l]$ to its (u, v) position in the texture domain D. Larger textures give superior approximations, but they also give much larger linear systems and thus slow down the execution of the algorithm considerably. Textures in our implementation are always square, to match the square projection image.

We now must determine how to relate a projected sample $\mathbf{f}[k]$ to its associated texels in $\mathbf{x}$. We do not provide here a full analysis of the best reconstruction method. In our implementation we use a bilinear reconstruction filter, the details of which can be found in Section 3.2.7. In this section we present the problem generally. For a general reconstruction and resampling operation, each sample will be constrained by a set of texels and weights for each texel. For a single term, we can express the approximation of a sample $\mathbf{f}[k]$ as a vector dot product between a vector of filter weights $\mathbf{w}_k$ and the texel value vector $\mathbf{x}$:

$$\mathbf{f}[k] \approx \mathbf{w}_k \cdot \mathbf{x} \tag{3.1}$$

A single weight $\mathbf{w}_k[l]$ is nonzero if the texel $\mathbf{x}[l]$ contributes to the value of the sample k, and zero otherwise. The specific configuration of the weighting vector $\mathbf{w}_k$ depends on the position of the projected sample in D and the reconstruction method used. We can collapse the weight vectors into a $N \times M$ matrix, where $M = |\mathbf{x}|$:

$$\mathsf{A} = \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_N \end{pmatrix}$$

We can then express a full single term approximation very compactly:

$$\mathbf{f} \approx \mathsf{A}\mathbf{x} \tag{3.2}$$

For a J term approximation, we create a matrix A_j and a vector $\mathbf{x}_j$ for each term. We can then express a J term approximation as:

$$\mathbf{f} \approx \sum_{j=1}^{J} \mathsf{A}_j \mathbf{x}_j \tag{3.3}$$

We can again make this expression more compact by defining a matrix B and a vector $\mathbf{y}$:

$$\mathsf{B} = \begin{pmatrix} \mathsf{A}_1 & \cdots & \mathsf{A}_J \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_J \end{pmatrix}$$

The full J term approximation can then be expressed by one matrix-vector product:

$$\mathbf{f} \approx \mathsf{B}\mathbf{y} \tag{3.4}$$

This product is a system of linear equations, which can then be fed into a standard linear system solver. The result is the vector $\mathbf{y}$ that best fits the BRDF $\mathbf{f}$ under the weights specified by B.

3.2.2 Single Term vs. Multi Term Approximation

The formulation presented above solves for all $\tilde{p}_j$ simultaneously. While this guarantees the best possible results, it also requires a very large system of equations. Speed of evaluation is an issue for all but the smallest textures, so we in practice often approximate only a subset of the $\tilde{p}_j$. We choose the n last terms of the approximation to solve simultaneously, where n is usually 1 or 2. We set up our approximation as follows

$$\mathbf{f} - \mathbf{f}_{J-n-1} \approx \mathsf{B}_{J-n,J}\mathbf{y}_{J-n,J} \tag{3.5}$$

where $\mathbf{f}_{J-n-1}$ is the approximation of $\mathbf{f}$ up to term $J-n-1$, which we assume was found previously (in an earlier iteration of the outer loop). $\mathsf{B}_{J-n,J}$ and $\mathbf{y}_{J-n,J}$ are obtained as above, except containing only $\mathsf{A}_{J-n} \ldots \mathsf{A}_J$ and $\mathbf{x}_{J-n} \ldots \mathbf{x}_J$, respectively.

We have found that significant improvement in final error can be achieved by increasing n from 1 to 2, but there are diminishing returns after that point.

3.2.3 Ensuring Reciprocity

Reciprocity is a condition on a BRDF that holds if $f(\hat{w}_i, \hat{w}_o) = f(\hat{w}_o, \hat{w}_i)$ for all directions $\hat{w}_i$ and $\hat{w}_o$. As mentioned previously, this is a necessary qualification for BRDFs based on real materials. Our algorithm as described so far does not make any explicit attempt to preserve this quality. If the BRDF is well sampled and the approximation is good, reciprocity will of course be very nearly preserved automatically. As mentioned in Chapter 2, however, we can ensure reciprocity by construction. The basic principle is to replace each term $\tilde{p}(\pi(\hat{w}_i, \hat{w}_o))$ of the approximation with two terms, $\tilde{p}(\pi(\hat{w}_i, \hat{w}_o))$ and $\tilde{p}(\pi^*(\hat{w}_i, \hat{w}_o))$. Both the replacement terms are based on one texture, $\tilde{p}$, but each has different projections, π and π^*. The projection π^* is the "conjugate" of π. Two projections are conjugate if and only if:

$$\forall \hat{w}_i, \hat{w}_o, \pi(\hat{w}_i, \hat{w}_o) = \pi^*(\hat{w}_o, \hat{w}_i)$$

In this way, we ensure that

$$\tilde{p}(\pi(\hat{w}_i, \hat{w}_o)) + \tilde{p}(\pi^*(\hat{w}_i, \hat{w}_o)) = \tilde{p}(\pi(\hat{w}_o, \hat{w}_i)) + \tilde{p}(\pi^*(\hat{w}_o, \hat{w}_i))$$

which is to say that reciprocity is preserved for those two terms. If we construct every term of the approximation in this way, we are assured that the entire approximation will preserve reciprocity. Ensuring reciprocity is very cheap, since it does not increase the number of textures and thus does not increase the size of our linear system.

McCool uses this trick in his approximation. Although he uses three projections, one of his textures is shared between two projections. The three projections are:

$$\begin{aligned}
\pi_1(\hat{w}_i, \hat{w}_o) &= \hat{w}_i \\
\pi_2(\hat{w}_i, \hat{w}_o) &= \hat{\mathbf{h}} \\
\pi_3(\hat{w}_i, \hat{w}_o) &= \hat{w}_o
\end{aligned}$$

The half-angle projection π_2 is symmetric, so it immediately preserves reciprocity and can have its own texture. The incoming and outgoing projections π_1 and π_3 are conjugate. To exploit this, McCool restricts his approximation so that π_1 and π_3 share a texture.

We set up our linear system for enforced reciprocity in the following way. We replace each $\mathsf{A}_j\mathbf{x}_j$ term in our original formulation (Equation 3.3) with a pair of terms, one each for the projection and its conjugate:

$$\mathbf{f} \approx \sum_{j=1}^{J} \mathsf{A}_j\mathbf{x}_j + \mathsf{A}_j^{'}\mathbf{x}_j \tag{3.6}$$

where A_j is the weighting matrix for π_j, and A'_j is the weighting matrix for π^*_j (we use $'$ to avoid confusion with the adjoint matrix). Since both A_j and A'_j operate on the same $\mathbf{x}_j$, we may simply add them together. The large matrix B with enforced reciprocity then takes the form:

$$\mathsf{B}' = \begin{pmatrix} \mathsf{A}_1 + \mathsf{A}'_1 & \cdots & \mathsf{A}_J + \mathsf{A}'_J \end{pmatrix}$$

The final matrix is thus no larger than without reciprocity, although in a sense we have doubled the number of terms in our approximation. Each term now has an added restriction, however. The technique may or may not improve the accuracy of the approximation, depending on the BRDF. We have found that theoretical approximations usually benefit from enforced reciprocity, while the results for measured BRDFs are more mixed.

Finally, we need to find π^* for any π in our search space. Under the standard $(\hat{w_i}, \hat{w_o})$ parameterization the argument of the projection is of the form $(s, t, u, v)^{\mathrm{T}}$, where (s, t) are the parabolic coordinates of $\hat{w_i}$, and (u, v) are the coordinates of $\hat{w_o}$. For this parameterization, finding π^* is simple. Since we know

$$\pi = \begin{pmatrix} a & b & c & d \\ e & f & g & h \end{pmatrix}$$

The conjugate projection π^* is simply

$$\pi^* = \begin{pmatrix} c & d & a & b \\ g & h & e & f \end{pmatrix}$$

For the Rusinkiewicz parameterization, (s, t) correspond to the parabolic coordinates $\hat{\mathbf{h}}$, and (u, v) to the coordinates of $\hat{\mathbf{d}}$. In this case we define:

$$\pi^* = \begin{pmatrix} a & b & -c & -d \\ e & f & -g & -h \end{pmatrix}$$

Since a switch of $\hat{w_i}$ and $\hat{w_o}$ under the Rusinkiewicz parameterization amounts to flipping $\hat{\mathbf{d}}$ around the origin while keeping $\hat{\mathbf{h}}$ constant.

3.2.4 Projection Masking

The image of the 4-cube under an arbitrary projection π will not cover a square texture exactly. The image of the 4-cube can range from a four sided shape to an eight sided shape depending on the rotation of the projection. We make the restriction that the image of the projection must lie entirely inside the texture domain, but this restriction leaves parts of the texture domain outside the image of 4-cube. If included

in the linear system, these unconstrained texels will at best slow down our linear system solver, and at worst cause the solver to fail.

We remove unconstrained texels from the linear system by associating a mask with each projection. The mask has an entry for each texel in the target texture. The value of the mask is 0 if the texel is not in the image of the projection, and a positive integer if it is. When we set up the linear system, we make sure to include only texels that have a positive mask value. The mask value itself determines the mapping between the position of texels in the vector $\mathbf{x}$ and their positions in the texture domain D. We use the map $maskmap(u, v)$ which takes a texel (u, v) position and maps it to a value between 1 and M ($|\mathbf{x}| = M$), or to zero if that texel lies outside the image of the 4-cube.

Creation of the mask is achieved by applying the projection to the sixteen corners of the 4-cube. Since the projection is linear, the convex hull of the projected corners encompasses the entire image of the projection. We apply a giftwrapping algorithm to obtain the convex hull. We then include in our mask all texels that lie inside the convex hull, and all the neighbors of those texels. The neighbors are included to allow bilinear reconstruction of points on the edge of the convex hull. We then save both the mask and the total number of texels inside the image, which for projection π_j we denote T_j.

3.2.5 Reconstruction Smoothing

The samples of a BRDF may be very sparse, or may have holes at certain angles. Under these conditions a texel may be completely unconstrained even after the masking operation. That is, for element $\mathbf{y}[l]$ the lth column of B is zero. Additionally, the BRDF samples (for measured BRDFs) may be noisy. In both these cases we would like to smooth the texture factors, to smooth over holes and smooth out noise. To achieve this, McCool uses a Laplacian smoothing operator on the textures. The operator can be added on the the bottom of the matrix B as a set of additional constraints on the texel values of $\mathbf{y}$.

3.2.6 Linear System Solution

Once we have the approximation in the matrix form, we can use one of the many publicly available linear equation solvers to obtain the vector $\mathbf{y}$. Following McCool, we chose the Quasi-Minimal Residual (QMR) algorithm implemented in the NIST's IML++ library.

The details of the operation of the QMR algorithm are not important to us, other than that it requires a square, non-singular matrix on which to operate. To satisfy the algorithm, we solve the least squares version of our equation, obtained by multiplying

both sides by the transpose of matrix B:

$$\mathsf{B}^{\mathrm{T}}\mathbf{f} = \mathsf{B}^{\mathrm{T}}\mathsf{B}\mathbf{y} \tag{3.7}$$

Assuming that there are no degenerate (all values equal 0) columns of B, $\mathsf{B}^{\mathrm{T}}\mathsf{B}$ is a square, non-singular matrix. B will have degenerate columns only in the cases where some texels are unconstrained, a situation we rely on our smoothing algorithm to avert.

3.2.7 Bilinear Weights

For completeness, we present the equations we use to determine the weighting vectors $\mathbf{w}_k$. Following McCool, we use bilinear reconstruction, one of the simplest possible reconstruction filters. For each projected sample point we assign bilinear coefficients to the four nearest texels. For sample k and projection π_j, the bilinear coefficients of the projected sample position $(u_{jk}, v_{jk}) = \pi_j(\hat{w}_i^k, \hat{w}_o^k)$ are found as follows. Note that to determine these weights we transform from the texture domain D used above into a new domain of width and height equal to the texture size, where the texels lie exactly on the integers.

$$\begin{aligned}
(U_{jk}, V_{jk}) &= (\lfloor u_{jk} \rfloor, \lfloor v_{jk} \rfloor) \\
(\alpha_{jk}^u, \alpha_{jk}^v) &= (u_{jk} - U_{jk}, v_{jk} - V_{jk}) \\
(\beta_{jk}^u, \beta_{jk}^v) &= (1 - \alpha_{jk}^u, 1 - \alpha_{jk}^v)
\end{aligned} \tag{3.8}$$

(u_{jk}, v_{jk}) are assumed to be transformed from D to the integer domain prior to applying the previous equations. The four texels surrounding (u_{jk}, v_{jk}) are at coordinates (U_{jk}, V_{jk}), $(U_{jk}+1, V_{jk})$, $(U_{jk}, V_{jk}+1)$, and $(U_{jk}+1, V_{jk}+1)$, and their bilinear coefficients are $\beta_{jk}^u\beta_{jk}^v$, $\alpha_{jk}^u\beta_{jk}^v$, $\beta_{jk}^u\alpha_{jk}^v$, and $\alpha_{jk}^u\alpha_{jk}^v$, respectively. These values make up the constraints for the BRDF value $\mathbf{f}[k]$.

We find the weighting vector $\mathbf{w}_k$ by using the projection masking map defined above. Let

$$\begin{aligned}
a_j &= maskmap(U_{jk}, V_{jk}) \\
b_j &= maskmap(U_{jk}+1, V_{jk}) \\
c_j &= maskmap(U_{jk}, V_{jk}+1) \\
d_j &= maskmap(U_{jk}+1, V_{jk}+1)
\end{aligned} \tag{3.9}$$

We then define

$$\begin{aligned}
\mathbf{w}_k[a_j] &= \beta^u_{jk}\beta^v_{jk} \\
\mathbf{w}_k[b_j] &= \alpha^u_{jk}\beta^v_{jk} \\
\mathbf{w}_k[c_j] &= \beta^u_{jk}\alpha^v_{jk} \\
\mathbf{w}_k[d_j] &= \alpha^u_{jk}\alpha^v_{jk}
\end{aligned} \tag{3.10}$$

$\mathbf{w}_k[l] = 0$ for all other l. It should never happen that any of $a \ldots d = 0$, since in our masking operation we include the neighboring texels of any texels that lie in the image of the 4-cube. If we unroll the dot product $\mathbf{f}[k] = \mathbf{w}_k\mathbf{x}$ under the bilinear reconstruction, the full linear equation for each BRDF sample then is

$$\begin{aligned}
\mathbf{f}[k] = \sum_{j=1}^{J} &\beta^u_{jk}\beta^v_{jk} \cdot \tilde{p}_j(U_{jk}, V_{jk}) + \alpha^u_{jk}\beta^v_{jk} \cdot \tilde{p}_j(U_{jk}+1, V_{jk}) \\
&+ \beta^u_{jk}\alpha^v_{jk} \cdot \tilde{p}_j(U_{jk}, V_{jk}+1) + \alpha^u_{jk}\alpha^v_{jk} \cdot \tilde{p}_j(U_{jk}+1, V_{jk}+1)
\end{aligned} \tag{3.11}$$

3.3 Rendering with the Approximation

The images in this paper were generate by directly looking up values in our approximation and using them to Gouraud shade a highly tessellated model. A major purpose of this technique, however, is to allow rendering at high speed on graphics hardware. McCool suggests a way to use the vertex shader hardware in current NVIDIA chips [12] to speed up this process. Using this hardware, McCool achieves approximately half the frame rates of normal vertex lit rendering. We have not attempted to render in hardware with our approximations, however we can provide a rough comparison of the cost of our algorithm versus McCool's.

When using hardware, we use the factors p_j to texture the object in the same way as with any normal 2D texture. The difference is in the calculation of the 2D texture coordinates (u, v). Instead of corresponding to the object's shape, our texture coordinates correspond to the light and viewer positions, projected onto 2D by π_j. Once the texture coordinates are calculated, the final product approximation can be evaluated using multi-texture hardware. The job of the vertex shader is to calculate the texture coordinates for each vertex. For efficient rendering, the total number of operations per vertex must be kept to a minimum. We present the general concept and rough cost analysis here, however decent gains can be made by using optimization tricks in the hardware.

The main problem is how to evaluate $(u, v) = \pi(\hat{w}_i, \hat{w}_o)$ as efficiently as possible. Assume that we are given the surface normal $\hat{\mathbf{n}}$, tangent $\hat{\mathbf{t}}$, binormal $\hat{\mathbf{b}}$, and the two

directions $\hat{w_i}$ and $\hat{w_o}$, all specified as vectors in a world frame. Since our BRDF is parameterized with respect to the surface frame (where $\hat{\mathbf{n}}$ lies on the z-axis), we must first transform $\hat{w_i}$ and $\hat{w_o}$ into the surface frame. This is done by applying a rotation matrix of the form

$$\mathbf{R} = \begin{pmatrix} \hat{\mathbf{t}} & \hat{\mathbf{b}} & \hat{\mathbf{n}} \end{pmatrix}^{-1}$$

We then perform the parabolic mapping H specified in Chapter 1 on the transformed $\hat{w_i}$ and $\hat{w_o}$ to get a four valued vector $h = (H(\hat{w_i}, \hat{w_o}))^{\mathrm{T}}$. Finally, we apply the projection π_j (in 2x4 matrix form) to h to get the texture coordinates (u, v).

The total cost of this operation is six dot products to transform $\hat{w_i}$ and $\hat{w_o}$, two vector sums to compute H, and then two more dot products per term of the approximation, to compute the projections. For a two texture approximation under the standard $(\hat{w_i}, \hat{w_o})$ parameterization, the total cost is 10 dot products and two sums.

For comparison, using McCool's projections the cost per vertex would include the same six dot products to rotate $\hat{w_i}$ and $\hat{w_o}$ and two sums to compute H. In place of the projection computation cost, there is the cost of computing $\hat{\mathbf{h}}$. To compute $\hat{\mathbf{h}}$, we must perform one vector sum, a normalization operation, and finally another sum to find the parabolic coordinates. The total cost per vertex for McCool is six dot products, four sums, and one normalization operation. Taking into account the expensive nature of normalization, a two texture approximation in our algorithm (under the standard $(\hat{w_i}, \hat{w_o})$ parameterization) and McCool's is most likely roughly equivalent in cost.

For enforced reciprocity and the Rusinkiewicz parameterization, we have an additional cost. Under enforced reciprocity we count each projection, not each texture, as a term. A two texture approximation with enforced reciprocity has four terms. Essentially, we must perform the projection operation twice for each texture, doubling our projection computation cost.

If the BRDF is parameterized by the Rusinkiewicz parameterization, we also must perform a few extra steps. Once we have $\hat{w_i}$ and $\hat{w_o}$ in the surface frame, we must find the vectors $\hat{\mathbf{h}}$ and $\hat{\mathbf{d}}$. Recall Equations 1.4 in Section 1.2.1:

$$\hat{\mathbf{h}} = \mathrm{norm}(\hat{w_i} + \hat{w_o}) \quad \hat{\mathbf{r}} = \mathrm{norm}(\hat{\mathbf{t}} - (\hat{\mathbf{t}} \cdot \hat{\mathbf{h}})\hat{\mathbf{h}}) \quad \hat{\mathbf{s}} = \hat{\mathbf{h}} \times \hat{\mathbf{r}}$$
$$\mathbf{R} = \begin{pmatrix} \hat{\mathbf{r}} & \hat{\mathbf{s}} & \hat{\mathbf{h}} \end{pmatrix}^{-1}$$
$$\hat{\mathbf{d}} = \mathbf{R}\hat{w_i}$$

Without any optimization, computing $\hat{\mathbf{h}}$ and $\hat{\mathbf{d}}$ requires two vector sums, one scalar-vector multiplication, two normalization operations, four dot products, and one cross product. The extra cost associated with the Rusinkiewicz parameterization must be balanced with the smaller approximation errors that it usually provides.

Chapter 4

Results and Conclusion

In this chapter we give the results of our algorithm tested on several BRDFs, both theoretical and measured. The approximation errors for each BRDF are provided. For comparison, we also provide the error values for our implementation of McCool's original algorithm. We have tested each BRDF under both the standard $(\hat{w_i}, \hat{w_o})$ parameterization and the Rusinkiewicz parameterization. For each parameterization, we tested with enforced reciprocity and without.

Our best approximation improves on McCool's original algorithm in almost every case. Our algorithm can also automatically extend the approximation to any desired number of terms to improve the approximation further.

4.1 Testing Procedure

We tested each BRDF by running our search algorithm for a maximum of five textures. For our approximations with enforced reciprocity, five textures equates to ten projections. We used random projection guesses for our outer loop. For each term, we performed six random restarts to attempt to find the global minimum of our error function. In our inner loop, we performed a multiple term approximation of the last two terms ($n = 2$).

We used two theoretical BRDFs for testing: a general Cook-Torrance function as described in Foley *et al.* [4], and a blue paint function based on the model proposed by Lafortune *et al.* [11]. Both Cook-Torrance and Lafortune attempt to model surface reflectance generally, but each has different properties.

The Cook-Torrance model appears for most viewing angles very similar to the basic Phong model. The specular term of the Cook-Torrance function is modified from Phong to model the "off-specular" reflection effect. This phenomenon consists of a significant increase in specular reflection at grazing angles (where the incident

angle is close to $\pi/2$), which is observed in real materials. The general shape of the specular peak remains similar to Phong, however. Cook-Torrance uses a constant diffuse term.

The Lafortune model is more general than the standard Cook-Torrance function, and can be used to model materials that have more sophisticated reflection effects than just specular. The particular BRDF that we have implemented here is based on a measured BRDF of blue paint, as in the original Lafortune paper. The paint BRDF has several interesting features that are reproduced in the model, including retro-reflection and off-specular reflection. Retro-reflection is reflection back in the incident direction, which is visible in Figure 4.4 as the light halo around the sphere. The diffuse term of the Lafortune model may vary with viewing direction, though here we have just implemented it as a constant.

We have also tested several measured BRDFs from both the CURET and Cornell databases. These include red velvet and leather from the CURET database, and several automotive paints (garnet red, krylon blue, and cayman blue) from the Cornell database. The Cornell paint BRDFs exhibit fairly standard specular and diffuse behavior, close to the Cook-Torrance model. The CURET BRDFs were chosen for their more varied structure.

4.2 Speed Issues

Our algorithm is slow. Depending on BRDF and texture size, one evaluation of the inner loop can take from several seconds to over a minute on an 800MHz Athlon with 256MB RAM. One evaluation with a BRDF of 45k samples at a texture size of 16, solving two terms simultaneously, takes approximately 30 seconds. Depending on the quality of the initial guesses, the outer loop takes approximately 100 function evaluations per term. The cost of directly searching for larger textures, such as 32 squared and above, is prohibitive.

To generate approximations with large textures, we run the search procedure at a low resolution first, and then use those projections as guesses for our larger texture sizes. Generally the best projections will work well at both low and high resolutions, so this is an effective time saving technique. The time taken to generate an approximation is still significant, however. The Lafortune model was one of the longest runs, finishing in just under six hours for a five term approximation with size 16 textures and approximately 45k samples.

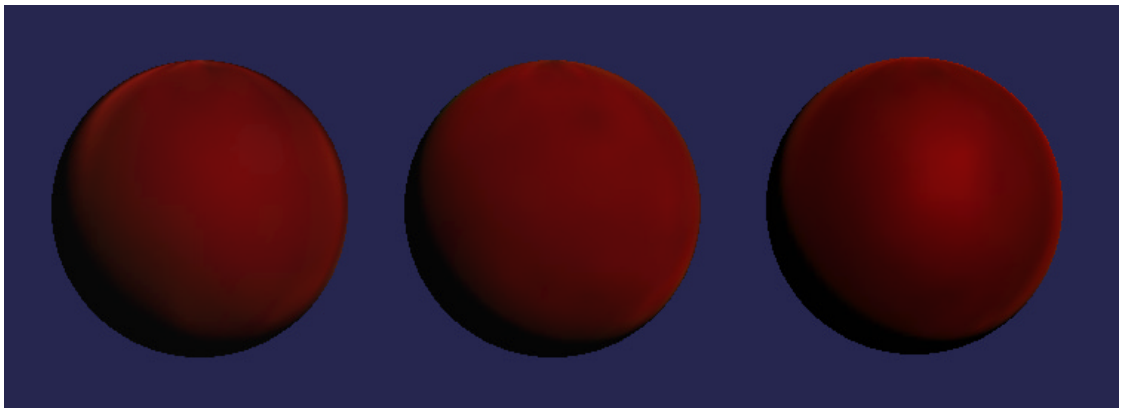

Figure 4.1: The CURET velvet BRDF. Left is the full BRDF, center is an approximation using the $(\hat{w_i}, \hat{w_o})$ parameterization, and right is the Rusinkiewicz parameterization.

4.3 Error Measurements

We measured six error metrics for each approximation: the absolute, squared, and maximum errors both in log space (relative error) and in the normal space. The algorithm tends to minimize the relative square metric most efficiently, the inner loop algorithm minimizes that metric. It is important to note that all of these metrics can compete with each other. Though the trend lines of all generally match, the approximation with the smallest relative error may not be the same as the approximation with the smallest normal error. Moreover, it may be that none of these metrics best represents the subjective quality of the approximation.

More experimentation would be useful to devise a better subjective error metric. Two approximations may have very similar error values under our metrics, but appear quite different. Figure 4.3 shows an approximation of the CURET velvet BRDF under the standard and Rusinkiewicz parameterizations. Both have similar relative squared error values (0.09 and 0.1 respectively), however their appearances are quite different. Both approximations have five textures, with a texture size of 16.

4.4 Result Analysis

McCool's original projections are very effective for approximating the classic single specular lobe BRDF. This is shown by the fact that for Cook-Torrance, our algorithm can match but not improve upon the approximation error of standard McCool until at least another term is added. It appears that in this case the restrictions on our

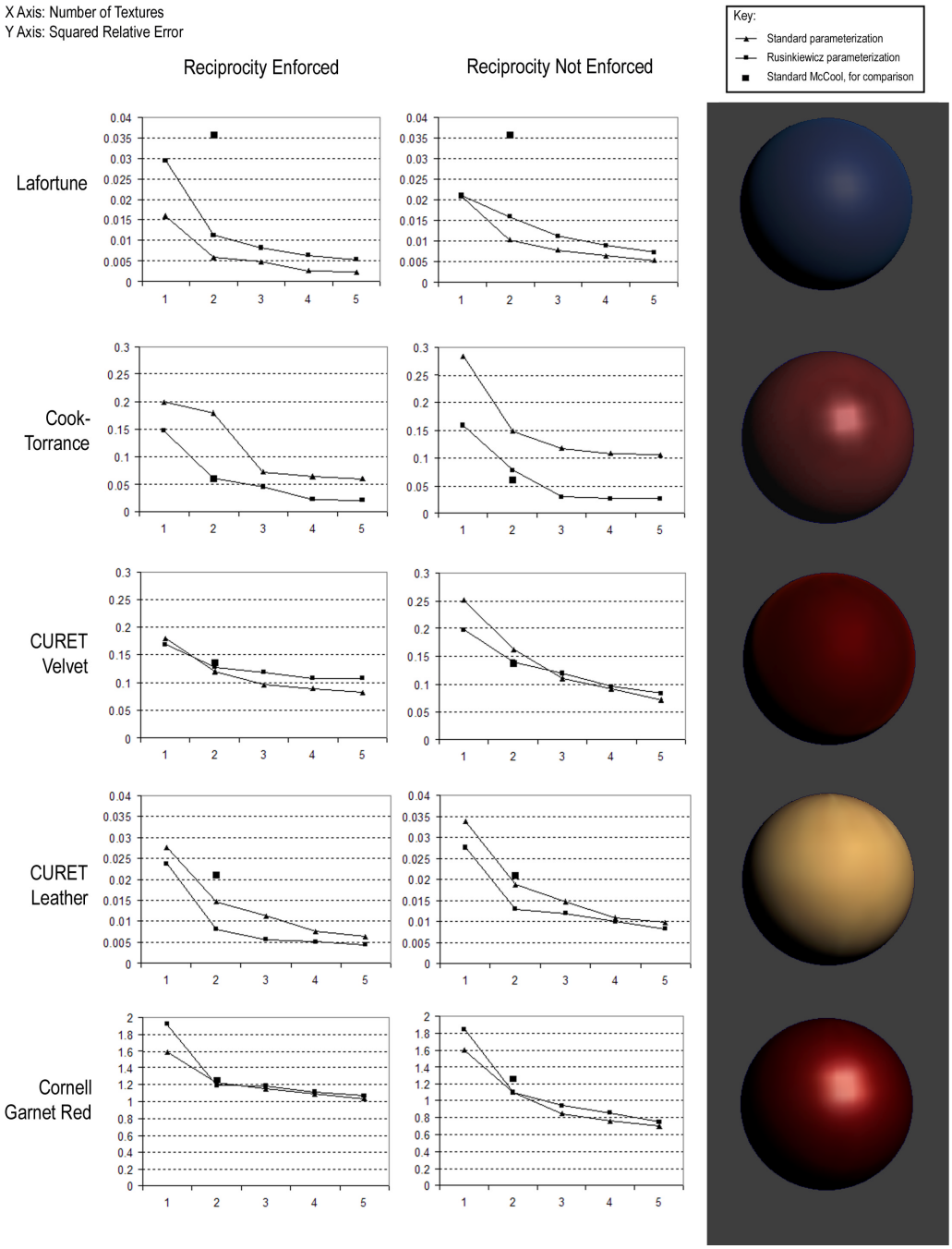

Figure 4.2: Error values for BRDF approximations, with and without reciprocity and for both standard and Rusinkiewicz parameterizations. All approximations have five textures, and a texture size of 24.

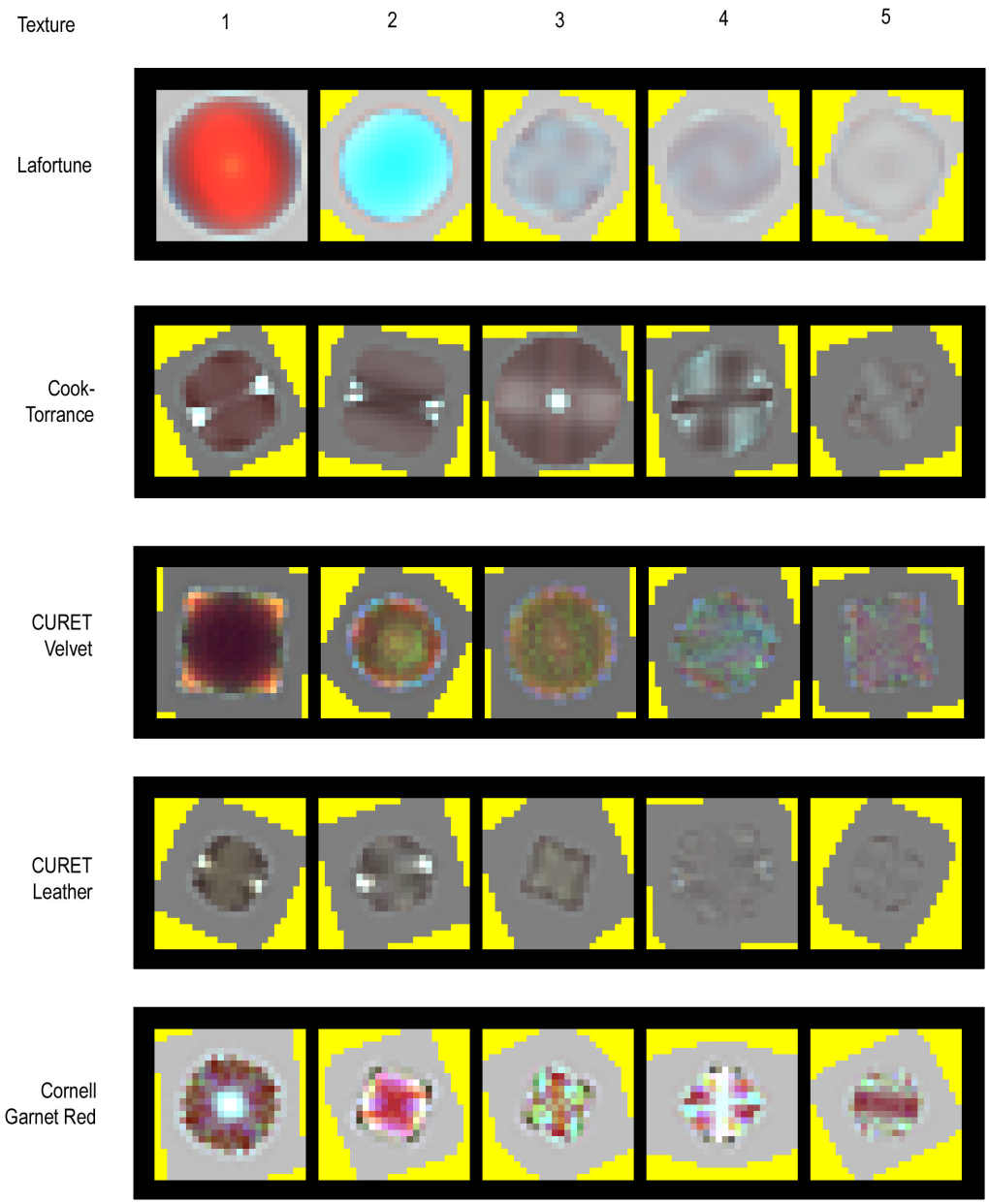

Figure 4.3: 2D textures for the approximations on the previous page. Texture size is 24.

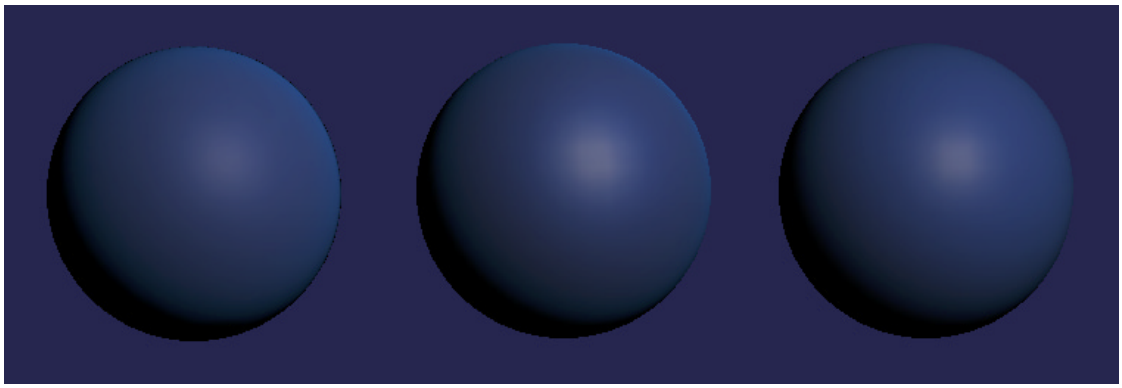

Figure 4.4: The Lafortune BRDF. From left, the full BRDF, our approximation, and the McCool approximation. Both approximations have two textures, texture size is 32.

projection search space are significantly detrimental. The measured BRDFs that are close to this specular model, such as the Cornell paint functions, also are approximated well by McCool, although in those cases our algorithm is able to find a better fitting two texture approximation.

Once the structure of the BRDF becomes more complex, however, we start to see marked improvements from our optimization algorithm. Most of the measured BRDFs benefit significantly from our optimization. The relative squared error for the CURET leather BRDF is cut by 60%. The most dramatic improvements are seen on the blue paint Lafortune model. Our algorithm is able to find an approximation that includes the retro-reflection effect, while the standard McCool projections can not. For Lafortune, our algorithm is able to cut the relative squared error for a two texture approximation by more than 83%. If we allow the approximation to run to five textures, the error is cut by more than half again. Figure 4.4 shows a comparison between the full Lafortune BRDF, our approximation, and the standard McCool approximation. Note the retro-reflection effects along the upper right of the sphere, which our captured by our approximation.

These results support the conclusion that the greater the complexity of the BRDF, the more that it benefits from a optimization algorithm such as ours. Error charts for several BRDFs are provided in Figure 4.2. The textures associated with the our best approximations are shown in Figure 4.3. The yellow areas are areas that are not included in the projection mask. For approximations with smaller approximation errors, such as the Lafortune and the CURET leather BRDFs, the final textures are almost uniformly gray, showing that the approximation is converging to an optimum. For approximations with higher errors, such as the Cornell garnet BRDF, the final

textures are still heavily colored, showing that the algorithm has difficulty converging.

Like Kautz and McCool [10], we find that neither the Rusinkiewicz or the $(\hat{w_i}, \hat{w_o})$ parameterization fits all BRDFs the best. BRDFs specifically based on the half-angle vector, such as Cook-Torrance, obviously benefit the most from the Rusinkiewicz parameterization. Highly specular BRDFs like the automotive paints also benefit, since a classic specular peak is dependent on the half-angle vector. In these cases we expect the standard parameterization to lag behind both the Rusinkiewicz parameterization and the standard McCool algorithm, since under the standard parameterization the half-angle projection is not in our search space.

In the Lafortune case, however, we find that the $(\hat{w_i}, \hat{w_o})$ parameterization is superior to the Rusinkiewicz. The approximation error for the standard parameterization is half that of the Rusinkiewicz parameterization. This result may have to do with the three separate specular lobes of the Lafortune approximation. The retro-reflection effects of the BRDF may be more easily captured without the half-angle projection.

Additionally, enforced reciprocity does not always increase the accuracy of the approximation, at least for large numbers of textures. Both cases where reciprocity does not yield a superior approximation (CURET velvet and Cornell garnet) have relatively high approximation errors, however, so preservation of reciprocity in these cases may not be so important in any event.

A common theme among all the BRDFs is the difficulty of effectively approximating the specular peak. The specular highlight tends to become washed out, overly widened, or simply off-color. This is not surprising, since such a prominent and narrow feature of the function is hard to approximate well. McCool uses a separate specular texture outside of his approximation to improve the appearance of his final product. We have not implemented a similar solution, although McCool's technique applies equally to our approximations.

4.5 Future Work

We would like to test our algorithm on more varied BRDFs in the future. The more exotic the BRDF, the more likely that our algorithm will find an effective approximation that would not have been found by hand.

Another direction for further work would be examining possible expansion of the search space of projections. Our tests show us that some special non-linear projections can be very effective. Arbitrary projections and our algorithm can coexist. One could, for example, use McCool's algorithm to determine the first three terms of an approximation. Beyond those three, one could use our algorithm to systematically find more refining terms. Finding an effective way to integrate this projections into our search space could enhance the effectiveness of the algorithm.

We would also like to examine rendering with our approximations using graphics hardware. Especially interesting would be the impact of the Rusinkiewicz parameterization on the speed of hardware rendering. Many BRDFs can be more effectively approximated using this parameterization, however it seems that the near real-time performance of hardware rendering may be severely impacted. Of course, as graphics hardware advances this will become less of a problem. Future graphics hardware should also allow us to use more approximation textures easily.

One other obvious avenue of future work is improving the speed of the search algorithm. Building a more sophisticated infrastructure to handle multi-resolution approximation is one direction of approach.

4.6 Conclusion

In this thesis we have presented a method of automatically creating a homeomorphic factorization for an arbitrary BRDF. The algorithm extends prior work in the field by reducing the amount of user input required to create an effective BRDF approximation. No prior analysis of BRDF features is required. The disadvantage of the approach is that the flexibility of the approximation is compromised to create a representation that can be automatically generated. We have found that the algorithm provides marginal gains for simple BRDFs, but provides significant improvements for complex functions.

Acknowledgements

Many thanks go to Shlomo, for his invaluable help on every aspect of the project. Thanks also to Gu and Pedro, for encouragement and help with the proofs.

The BRDF data was obtained from the CURET online database, *www.cs.columbia.edu/CAVE/curet/*, and from the Cornell online database, *www.graphics.cornell.edu/online/measurements/reflectance/index.html*. Thanks to Wojciech Matusik for additional BRDF data.

Finally, thanks to my readers for making it through to the end.

Appendix A

Proofs

A.1 Sufficiency of Orthogonal Projections

We wish to show that any linear projection $P : \mathbb{R}^m \rightarrow \mathbb{R}^n$ $(m > n)$ can be represented by an orthogonal projection $Q : \mathbb{R}^m \rightarrow \mathbb{R}^n$ followed by a linear transformation $L : \mathbb{R}^n \rightarrow \mathbb{R}^n$. P and Q can be represented by $n \times m$ matrices, and L can be represented by a $n \times n$ matrix, with respect to some bases for $\mathbb{R}^m$ and $\mathbb{R}^n$. By orthogonal projection, we mean a projection that has basis vectors that are orthogonal. The projection matrix of an orthogonal projection is row-orthogonal.

Formally (P, L, Q defined as above),

$$\forall P \; \exists L, Q \text{ s.t. } \forall \mathbf{x} \in \mathbb{R}^m, \; P\mathbf{x} = LQ\mathbf{x} \tag{A.1}$$

This can be shown by applying the singular value decomposition theorem. The singular value decomposition of a $U \times V$ matrix A (where $U \geq V$) is

$$\mathsf{A} = \mathsf{USV}^{\mathrm{T}} \tag{A.2}$$

where U is a $U \times V$ matrix, and S and V are both $V \times V$ matrices. S is a diagonal matrix, and both U and V are column-orthogonal matrices. If we take the transpose of the SVD equation and let $U = m$ and $V = n$, we obtain

$$\mathsf{A}^{\mathrm{T}} = \mathsf{VSU}^{\mathrm{T}} \tag{A.3}$$

where A^{T} is a $n \times m$ matrix and U^{T} is a n *timesm*, row-orthogonal matrix. We can then make the following substitutions: let A^{T} equal the matrix representation of P and U^{T} equal the matrix representation of Q. The $n \times n$ matrix VS is then the linear transformation L.

We have shown the existence of an orthogonal projection Q and a linear transformation L, given an arbitrary projection P, by using singular value decomposition.

Bibliography

[1] Beers, A. C., Agrawala, M., and Chaddha, N. "Rendering with Compressed Textures." *Computer Graphics (SIGGRAPH 1996 Conference Proceedings)*, pages 373-378, 1996

[2] Blinn, J. F., "Models of Light Reflection for Computer Synthesized Pictures." *SIGGRAPH 1977 Conference Proceedings*, pages pages 192-198, 1977

[3] Downs, L. "Do quaternions really perform rotations?" *CS182 Course Notes.* www.cs.berkeley.edu/∼laura/cs184/quat/quatproof.html, 1999

[4] Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F. *Computer Graphics: Principles and Practice*, Second Edition in C. Addison-Wesley, 1997

[5] Fournier, A. "Separating Reflection Functions for Linear Readiosity." *Rendering Techniques '95 (Eurographics Workshop on Rendering)*, pages 383-392, 1995

[6] Gortler, S. J. "Quaternions and Rotations." *Unpublished Course Notes,* 2001

[7] Gu, X. Personal Communication

[8] Heidrich, W., and Seidel, H. -P. "View-Independent Environment Maps." *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39-45, 1998

[9] Heidrich, W., and Seidel, H. -P.. "Realistic, Hardware-Accelerated Shading and Lighting." *Computer Graphics (SIGGRAPH 1999 Conference Proceedings)*, pages 39-45, 1999

[10] Kautz, J., and McCool, M. D. "Interactive Rendering with Arbitrary BRDFs using Separable Approximations." *Rendering Techniques '99 (Eurographics Workshop on Rendering)*, pages 281-292, 1999

[11] Lafortune, E., Foo, S. C., Torrance, K., and Greenberg, D. "Non-Linear Approximation of Reflectance Functions." *Computer Graphics (SIGGRAPH 1997 Conference Proceedings)*, pages 117-126, 1997

[12] Lindholm, E., Kilgard, N., and Moreton, H. "A User-Programmable Vertex Engine." *Computer Graphics (SIGGRAPH 2001 Conference Proceedings)*, pages 149-158, 2001

[13] McCool, M. D., Ang, J., and Ahmad, "A. Homomorphic Factorization of BRDFs for High-Performance Rendering." *Computer Graphics (SIGGRAPH 2001 Conference Proceedings)*, pages 171-178, 2001

[14] Mebius, J. E. *History of the Quaternion Representation Theorem for Four-Dimensional Rotations.*
www.xs4all.nl/∼plast/So4hist.htm, 2001

[15] Phong, B. -T. "Illumination for Computer Generated Pictures." *Communications of the ACM*, 18(6), pages 311-317, June 1975

[16] Poulin, P., and Fournier, A. "A Model for Anisotropic Reflection." *Computer Graphics (SIGGRAPH 1990 Conference Proceedings)*, pages 273-282, 1990

[17] Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, 1992

[18] Rusinkiewicz, S. "A New Change of Variables for Efficient BRDF Representation." *Eurographics Workshop on Rendering,* pages 1123, 1998.

[19] Torrance, K. E., and Sparrow, E. M. "Theory for off-specular reflection from roughened surfaces." *Journal of the Optical Society of America* 57, 9, pages 1105-1114, 1967

[20] Ward, G. "Measuring and Modeling Anisotropic Reflection." *Computer Graphics (SIGGRAPH 1992 Conference Proceedings)*, pages 265-272, 1992

[21] Weisstein, E. W. *Eric Weisstein's World of Mathematics.*
http://mathworld.wolfram.com