Image Streaming to Build Image-Based Models

by
Karl Hillesland

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2005

Approved by:

Anselmo Lastra, Advisor

Radek Grzeszczuk, Reader

Dinesh Manocha, Reader

Leonard McMillan, Committee Member

Jon Tolle, Committee Member

ABSTRACT

KARL HILLESLAND: Image Streaming to Build Image-Based Models
(Under the direction of Anselmo Lastra)

An important goal in computer graphics is to synthesize photo-realistic images of objects and environments. The realism of the images depends on the quality of the model used for synthesis. Image-based modeling is an automated modeling technique aimed at achieving this realism by constructing models from photographs of real-world objects or environments. A wide number of image-based modeling techniques have been proposed, each with its own approach to model generation. However, we can treat all image-based modeling as a form of function fitting, where the function is our representation for appearance, and the data to which we want to fit are the photographs.

This dissertation addresses the use of nonlinear optimization to construct image-based models of an object's appearance, including a new algorithm for efficient computation that is designed to take advantage of the high computational throughput of programmable graphics hardware in order to generate the models in a timely manner. Application to two diverse types of image-based models demonstrates the versatility and computational efficiency of the approach. The two types are Light-Field Mapping (Chen et al., 2002a), which is a radiance model generated from data decomposition, and a per-texel Lafortune representation (McAllister et al., 2002), which is an analytic reflectance model.

Since GPUs (graphics processing units) lack the precision and accuracy of CPUs, this work also includes a closer look at the kind of numerical error that occurs in employing a numerical technique such as nonlinear optimization in this limited precision and accuracy context. Furthermore, since GPU floating-point operations are not fully documented, this work also includes a technique to measure the accuracy of GPU floating-point operations.

To my parents, for their selfless support and love.

ACKNOWLEDGMENTS

My journey in graphics began with the introduction, encouragement and support of Victor Roetman, Bob Lewis, and John Hart at Washington State University. I owe them the credit for getting me involved in computer graphics.

Anselmo Lastra has been the perfect advisor; always supportive, encouraging, and behind me 100%. It was through his encouragement that I was able to pull together a dissertation from what I had started at Intel. He has always provided me with sound advice with respect to both technical pursuits and the practicalities of completing my work at Chapel Hill.

Dinesh Manocha supported me in the Walkthrough group for three years, and was good enough to serve as a reader for this dissertation. One of his many contributions to my success was the encouragement to look more closely at the numerical issues in using graphics hardware.

Radek Grzeszczuk gave me a year long internship at Intel. It was during this time that he gave me the original suggestion to pursue this work, and he worked as hard as I did on the SIGGRAPH paper to make it a success. He also served as a reader on this dissertation and gave many helpful suggestions for presentation. Radek has also been a strong advocate for my success.

I took a course of nonlinear optimization from Jon Tolle, which helped me to formalize my approach to the topic. He was kind enough to help me out on this dissertation by serving on my committee and reviewing critical chapters related to nonlinear optimization.

Leonard McMillan was also kind enough to serve on my committee. He jumped right in with the proposal to offer many helpful suggestions of where to look for future pursuits, and to help expand my thoughts on how and where these techniques could be applied.

I would like to thank the people in the Department of Computer Science at the University of North Carolina at Chapel Hill who have been a terrific help in making this all happen. I've had great support from staff, who have kept things running smoothly for me. My professors and classmates have challenged me to go further than I could have alone; they have been generous with their time and support, and have made my time at UNC a pleasure.

Finally, I would like to thank Margaret Marshall for her love, support, and patience through this whole process.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

α	Stepsize in a line search
C_x, C_y, C_z	Lobe direction coefficients in the Lafortune model
C_xy	$C_xy = C_x = C_y$ for isotropic Lafortune lobes
$\mathbf{d}$	Direction in a line search
$E(\mathbf{p})$	Error function, which is a function of a parameter choice for a fixed set of data
$f(\mathbf{p})$	Objective function, typically including the error function $E(\mathbf{p})$ and any penalty terms $P(\mathbf{p})$.
$g(\mathbf{s})$	Surface map lookup function in LFM
$\mathbf{H}$	Hessian operator
$h(\omega)$	View map lookup function in LFM
$\mathbf{J}$	Jacobian operator
k	Index, typically used as a subscript to indicate which iteration of an iteration in nonlinear optimization
L_r	Reflected radiance
$m(\mathbf{p}, \mathbf{x})$	Model function, e.g. the Lafortune model function.
n	Number of model parameters
n	In the context of the Lafortune model, this is the specular exponent.
n_I	Number of iterations
n_i	Number of images
$P(\mathbf{p})$, $P_m(\mathbf{p})$	Penalty function and the *m*th penalty function, respectively
$\mathbf{p}$	Vector of model parameters, such as a specular exponent and albedo coefficients
ρ_d	Diffuse albedo
ρ_s	Specular albedo
$\mathbf{s}$	Surface parameterization
S	Number of data samples

T	A texture lookup function
ω	In the context of LFM, this is the view direction.
ω_r	Reflection direction (2D)
ω_i	Angle of incidence (2D)
$\mathbf{x}$	Vector of inputs to the model function, such as view direction, light direction and surface parameterization
$\mathbf{x}_i$	ith data sample input vector
y_i	ith data sample output, e.g. a pixel value from the photograph of the object

Chapter 1

Introduction

An important goal in computer graphics is to synthesize photo-realistic images of objects and environments. The realism of the images depends on the quality of the model used for synthesis. Image-based modeling is an automated modeling technique aimed at achieving this realism by constructing models from photographs of real-world objects or environments.

A wide number of image-based modeling techniques have been proposed, each with its own approach to model generation. However, we can treat all image-based modeling as a form of function fitting, where the function is our representation for appearance, and the data to which we want to fit are the photographs. This dissertation addresses the use of nonlinear optimization to construct image-based models of an object's appearance, including a new algorithm for efficient computation that is designed to take advantage of the high computational throughput of programmable graphics hardware in order to generate the models in a timely manner. Scalability is achieved by tying the active working set to model size rather than data size.

Application to two diverse types of image-based models demonstrates the versatility and computational efficiency of the approach. The two types are Light-Field Mapping (Chen et al., 2002a), which is a radiance model generated from data decomposition, and a per-texel Lafortune representation (McAllister et al., 2002), which is an analytic reflectance model.

Although faster than CPUs in terms of raw computational power, GPUs (graphics processing units) lack the precision and accuracy of CPUs. This work also includes a closer look at the kind of numerical error that occurs in employing a numerical technique such as nonlinear optimization in this limited precision and accuracy context. Furthermore, since GPU floating-point operations are not fully documented, this work also includes a technique to measure the accuracy of GPU floating-point operations.

The next section gives background on image-based modeling, and a summary of the approach proposed in this dissertation.

1.1 Background

In this dissertation, *Image-Based* refers to how a model is constructed, not the model's representation. The image-based models of interest here assume a known geometric description of the object to be modeled and a surface parameterization for one or more textures. The textures contain parameters for the appearance model, which is a function of the position on the surface of the object (surface parameterization), as well as other quantities such as viewing conditions and lighting conditions. These other quantities may also be stored in texture maps organized either by surface parameterization or some other parameterization, such as view direction. This kind of model will be called a *TIM* for *textured, image-based model* throughout this dissertation.

Pixels in an image represent samples of radiance. TIMs typically require hundreds to thousands of images, and a total of millions to billions of radiance samples. Construction of the model typically consists of an iterative process in which the TIM is used to synthesize an image corresponding to a photograph. The result of comparing the synthesized image to the original image is used to update the model parameters, which typically number in the millions for a complete object. The goal is to find the parameter values of the appearance model that best approximate the image.

Rather than trying to use a single model of this many parameters to fit all the data,

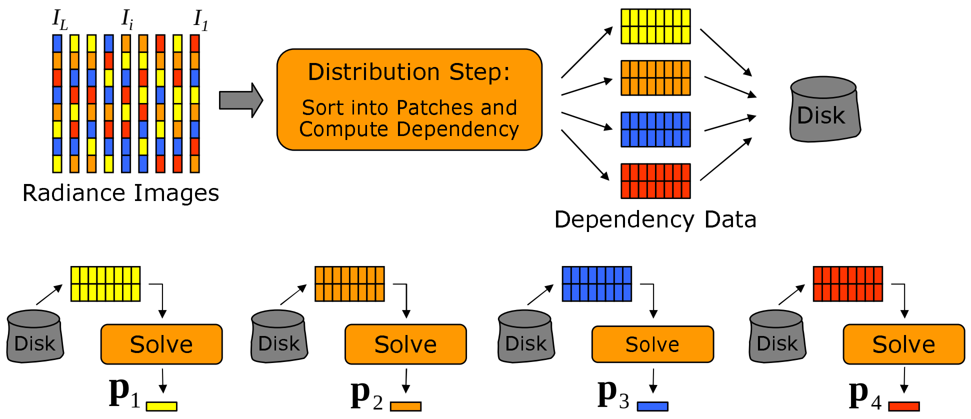


Figure 1.1: *The conventional approach is to preprocess all the image data, breaking it down into individual models and datasets. I call this preprocess step the "distibution step". Then each piece is solved on its own. This two-step approach requires augmentation of the original image-data through resampling and characterization of each image pixel in terms of the function space of the model, which is labelled as "dependency data" in this figure.*

the representation is often broken down into patches on the surface of the object and a local model is associated with each patch. The number of patches varies from about ten thousand to a million, and the number of parameters per patch varies from tens to thousands. The size of the patch is a trade-off between the number of patches and the number of parameters for each patch.

The conventional approach is to use a pre-process to turn the object and its images into independent, abstract models and their corresponding data sets. The data consist of image pixel data, which are typically resampled in some way, along with a characterization of each image pixel in terms of the function space of each model. The best fit for each individual model is then computed separately. This two-step approach is outlined in Figure 1.1.

In contrast, this dissertation proposes an image-streaming framework that treats the object and images as complete entities, and finds the best fit for all of the patches on the surface at once. In other words, notions of object, images, lights, and cameras, along with the entire rendering process, are kept throughout the solution process. This image-streaming approach is diagramed in Figure 1.2.

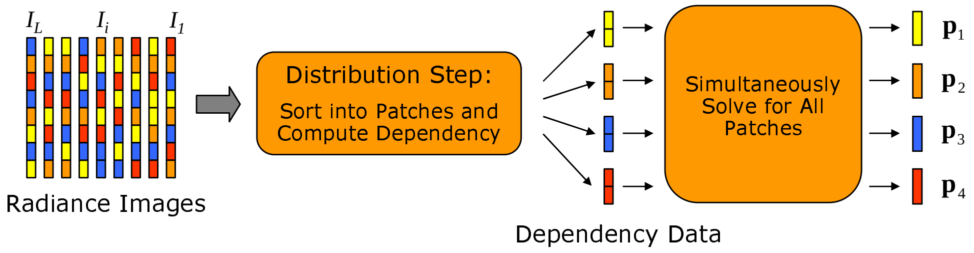


Figure 1.2: *In the image-streaming approach, the distribution step is performed on the fly, and only for a single image at a time, thus trading computation to save total storage and bandwidth requirements.*

An image-streaming approach on graphics hardware demonstrates favorable performance relative to the conventional approach. This is a result of paying a higher cost in terms of computation in exchange for reducing bandwidth requirements. By rearranging the computation in this manner, scalability is tied to model size, rather than data size and resampling on the fly for a single image at a time in the lower dimensional model space. Chapter 4 shows why this is true from a theoretical standpoint, while Chapters 7 and 8 give empirical results.

1.2 Summary of Previous Work

Related work comes from three main areas: 1) image-based modeling, 2) the use of graphics hardware for non-graphics applications and 3) nonlinear optimization. In this section, I give a brief summary of previous work. Further details are covered in later chapters.

1.2.1 Image-Based Modeling

Previous work in image-based modeling focused largely on developing new image-based models. With the development of each model, the authors developed an accompanying technique to generate the model parameters from the images. This dissertation instead proposes a model generation technique as a solution to a number of different kinds of image-based models.

For example, Chen et al. proposed Light Field Mapping (LFM), an image-based model generated using matrix factorization (Chen et al., 2002a; Chen, 2002). Lafortune et al. proposed a model generated using the Levenburg-Marquardt method, which is a nonlinear optimization technique only suitable for models with a small number of parameters (Lafortune et al., 1997). In this dissertation I look at nonlinear optimization techniques that will handle both a small parameter model like the Lafortune model, and also a large parameter model such as a Light Field Mapping model. In order to demonstrate this, I look more specifically at the LFM model as well as the SBRDF model, or spatial bidirectional reflectance-distribution function model, which is built from a large number of Lafortune functions (McAllister et al., 2002; McAllister, 2002). Chapter 2 gives further detail on these and other image-based models, describing how they fit into my framework.

1.2.2 Graphics Hardware for Non-Graphics Applications

Although the idea of using graphics hardware for non-graphics applications has been around for a while (Fournier and Fussell, 1988), there has been a recent resurgence due to the introduction of programmability in mainstream graphics processors (Lindholm et al., 2001). The GPGPU (general-purpose computation on GPUs) website, `http://www.gpgpu.org`, covers current work in the use of graphics hardware for general purpose computation.

Previous work includes exploration of numerical methods implemented on graphics hardware, particularly the solution of linear systems, partial differential equations which reduce to linear systems, and coupled map lattice approaches to simulation. This dissertation addresses the solution of nonlinear systems, or more specifically, nonlinear optimization.

The computer vision community has used graphics hardware to generate models (Yang et al., 2003; Yang and Pollefeys, 2003), but their goal is to reconstruct geometry using fairly simple appearance models to aid the process, whereas I focus on reconstructing the reflectance model itself. My work also shares some similarity to the work of Lensch et al. (Lensch et al., 2003), who use graphics hardware as part of a next-best view system for capturing an image-based model, but in their case, the computation is mainly performed on the CPU using a more conventional approach.

Further details on previous work in treating graphics hardware as a more general computational device are give in Chapter 4.

This dissertation also includes a new technique for measuring GPU floating-point behavior. Related work for CPUs concentrated on conformance testing and detection of logic error (Karpinski, 1985; Schryer, 1981). *GPUBench*, which measures precision of unary transcendental functions in graphics hardware (Buck et al., 2004), and my own test system for addition, subtraction, multiplication and division (Hillesland and Lastra, 2004) were presented at the first *ACM Workshop on General Purpose Computing on Graphics Processors* in 2004. I describe previous work along with my own work in Appendix A.

1.2.3 Nonlinear Optimization

I do not propose any new techniques for nonlinear optimization. However, since nonlinear optimization is not a typical subject covered in computer science, I give a brief tutorial in Chapter 3. Further details on nonlinear optimization can be found in a textbook on numerical nonlinear optimization such as *Numerical Optimization* by Nocedal and Wright (Nocedal and Wright, 1999) or *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* by Dennis and Schnabel (Dennis and Schnabel, 1996), or *Practical Optimization* by Gill, Murray and Wright (Gill et al., 1981).

1.3 Process Overview

Nonlinear optimization has been used for generating image-based models of appearance in the past. The complete procedure for generating models from images is given next.

1. Take pictures of the object with known viewing and lighting conditions.
2. Construct the object's geometry.
3. Choose an appearance model function.
4. Postulate a set of parameters for the appearance model.
5. Try to synthesize the reference images using the model. If the model does a good enough job of matching what was in the reference images, you are done. Otherwise proceed to the next step.
6. Use the result of this comparison to refine the model parameters and return to step 5.

This dissertation focuses on steps 5 and 6. All images are processed for a single iteration of the nonlinear optimization process, so there is an implied inner loop in step five. Figure 1.3 shows the stages of processing for each image.

1.4 Thesis

This section presents the thesis of the dissertation, and clarifies its claims. The next section outlines the demonstration of the thesis.

> An image-streaming approach to nonlinear optimization is a widely applicable and efficient technique for image-based model construction.

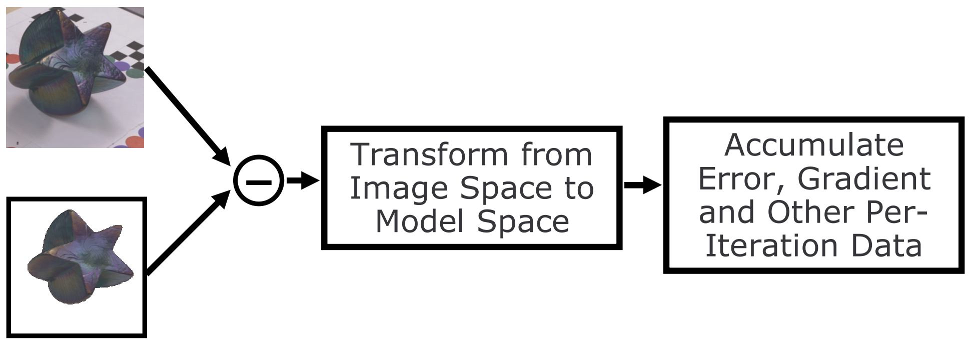

Transform from Image Space to Model Space
Accumulate Error, Gradient and Other Per-Iteration Data

Figure 1.3: *Conceptually, there are three stages of processing each image, which are outlined here. The first stage involves a comparison of the photograph (upper image) and an image created by using the image-based model (lower image). A transformation from image-space to model space occurs before or after the comparison, as discussed in Section 5; this figure signifies the latter case. Finally, some accumulation of information derived from each image is necessary.*

We have already discussed the overall process in terms of image-streaming. The remainder of this section clarifies some of the claims of the thesis statement.

1.4.1 What is Meant by "Image-Based Models"?

Although the image-streaming framework might be applied to reconstruction of geometric information, I only investigated the generation of models for surface appearance. In fact, the framework is specifically intended to create TIMs. Besides geometry, it is assumed that camera and lighting information are available as necessary for the model. In cases of relighting, there are no provisions in this dissertation for handling global illumination effects.

1.4.2 What is Meant by "Nonlinear Optimization"?

The nonlinear optimization technique must be one that works well within the streaming framework proposed. For example, a full Newton's method would be impractical in terms of space requirements, which are $O(n^2)$ with respect to the number of parameters. Secondly, the nonlinear optimization technique will require continuous derivatives.

Third, I have only considered cases where an analytical expression for the gradient is given. Finally, constraints are handled through a penalty method, which works well for soft constraints, but is difficult to use for hard constraints.

1.4.3 What is Meant by “Efficiently”?

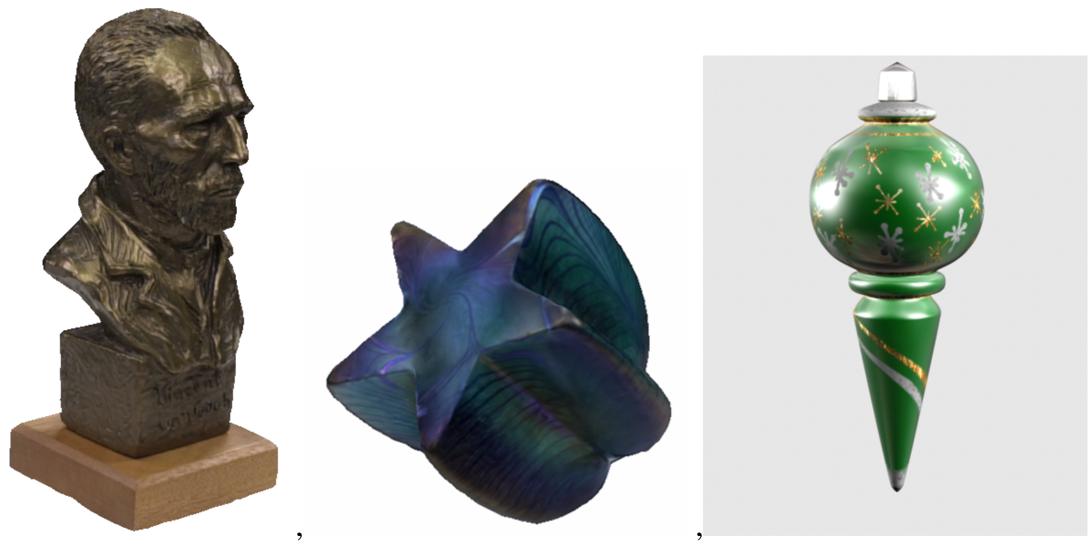
In order to be useful, the algorithm must scale well in terms of memory, bandwidth, and computational requirements, and it must show favorable performance relative to conventional methods on the CPU. The streaming model offers potential scaling benefits as discussed in Secton 4.1, and an algorithm that supports streaming can ride that faster curve.

1.5 Motivation and Demonstration

This section documents how the thesis is demonstrated by addressing each of the key aspects: that it is applicable to a fair range of image-based models and nonlinear optimization techniques, that it is efficient with respect to time and computational resources, and that it will produce the correct result.

1.5.1 Wide Applicability

Image-based modeling is a function fitting process. One goal of this dissertation is to explore how to fit an image-based model to image data. Rather than concentrating on a single modeling function, this work tries to address a wide class of image-based models by exploring the use of nonlinear optimization. Nonlinear optimization has been used to construct image-based models in some specific cases. This dissertation presents nonlinear optimization as a more general technique with respect to its applicability to image-based modeling. Chapter 3 provides background on nonlinear optimization that is relevant to its application in image-based modeling.



(a) *Bust* **(b)** *Star* **(c)** *Ornament*

Figure 1.4: *Three models models were used to demonstrate the application of the framework in practice. (a) and (b) are LFM models of real-world objects. (c) is a SBRDF model generated from a synthetic image-capture process.*

Any image-based modeling system with some form of surface parameterization will require a transformation from image space to model space. Chapter 5 describes the issues associated with this transformation. Not included are issues related to image registration and camera calibration, for which considerable literature already exists.

Chapter 2 describes the types of image-based models handled by this framework. I implemented two image-based modeling techniques under this framework to demonstrate its range of applicability in practice. The first is called *Light Field Mapping*, or LFM (Chen et al., 2002a; Chen, 2002). It is a model of view-dependent radiance that uses a compressed representation derived from a data decomposition approach. The second is a *spatial bidirectional reflectance-distribution function* or *SBRDF* (McAllister et al., 2002; McAllister, 2002). It fits an analytic reflectance model to acquired data. Figure 1.4 shows the models I used to demonstrate that both LFM and SBRDF models can be created using this framework.

I would not expect a general nonlinear optimization technique to be more effective than a special-purpose algorithm designed for a specific image-based modeling technique. However, in the cases I've tried so far, this framework is competitive in both time and accuracy with the original special-purpose algorithms used by the original authors. I believe an analysis of how to apply nonlinear optimization to image-based modeling in a more general sense should provide some useful insights with broad application in image-based modeling.

1.5.2 Scalability

Image-based modeling involves processing of large data sets. As already mentioned, there are on the order of thousands of images, and millions of model parameters. This work treats the image-based modeling process as an image-streaming process. The image-streaming approach and the choice of nonlinear optimization techniques reflect the need for handling data sets of this size.

The original images constitute the data in the stream. There are three key aspects to the scalability of this approach. First of all, the active working set is tied to model size (on the order of tens of megabytes) rather than data size (on the order of gigabytes). Second, any resampling is done on the fly, and only for a single image at a time. Third, resampling is done in model space, not in the original, and often much higher dimensional, data-space. Chapter 4 details how this is achieved. Chapter 3, which provides background on nonlinear optimization, concentrates on techniques designed for large problems, where storage requirements are $O(n)$ with respect to the number of model parameters.

1.5.3 Fast Computation

The computation to construct an image-based model can often be fairly time consuming. For example, the construction of an image-based model for re-lighting can take on the order of several hours or days for current techniques (McAllister, 2002), (Furukawa et al., 2002). This computation becomes an important bottleneck in the image-based modeling process. It is desirable to have some way to quickly evaluate the adequacy of the data captured, as it can be inconvenient, or even impossible, to return to a capture site after finding that there are data missing. With the ability to compute and evaluate a model in a timely manner, any missing or inadequate data can be identified while the object or environment is still readily available.

This dissertation will address speed in computation by showing how an image-streaming process using programmable graphics hardware results in efficient computation. This takes advantage of an important computational resource that is a component of nearly every computer system. Chapter 4 describes this streaming process and how to leverage graphics hardware for computation in this manner. Chapters 7 and 8 report timing results from implementing the framework on two different kinds of models to show its effectiveness in practice. In the LFM case study the GPU version performed a factor of five faster than an equivalent CPU version, and comparable to the original matrix factorization implementation. I believe the streaming formulation will prove to be important for real-time image-based model generation.

1.5.4 Numerical Error Analysis

Graphics hardware supports only single precision floating-point arithmetic (at best). Chapter 6 explores numerical error for the framework proposed in this dissertation. For example, image-based models often only need to match the photographs to within the eight-bit precision of a color buffer channel. Graphics hardware is restricted to a 32 bit floating-point representation, and also has a 16 bit mode to reduce bandwidth. Chapters 7 and 8 give empirical evidence that 16 bit precision is adequate.

Development of numerical methods requires knowledge of the floating-point behavior of the device. CPU manufacturers document specifications of the floating-point behavior of their hardware; therefore, CPU floating-point tests check for conformance to particular floating-point models (Karpinski, 1985) or tests for logic errors in their implementation (Schryer, 1981). GPU manufacturers, by contrast, do not currently publish specifications for the floating-point behavior of their hardware. Therefore, I have developed a system for *measuring* floating-point accuracy in addition, subtraction, multiplication, and division (Hillesland and Lastra, 2004), which is described further in Appendix A. To my knowledge, this is the first published technique for quantifying the accuracy of these operations.

1.6 Relationship to The SIGGRAPH Paper

Part of this work was presented as a 2003 SIGGRAPH paper (Hillesland et al., 2003). As the SIGGRAPH paper was a collaborative work with two others, it is important to distinguish the contributions of the individuals from my own contribution, as well as discuss what is different in this dissertation.

The SIGGRAPH paper presented the image-streaming framework, including an implementation for the same LFM and SBRDF models used in this dissertation. Optimization was performed using both the steepest descent and conjugate gradient methods. Time comparisons were made between a CPU implementation using the conventional approach and the image-streaming approach on the GPU. Error analysis consisted of comparing synthesized results against the original photographs. We analyzed resource utilization empirically. Next I outline the roles each of us played in the work for the SIGGRAPH paper. Naturally, there was some measure of overlap in our roles, but I have tried to emphasize the main responsibilities of each person.

Radek Grzeszczuk proposed the high level idea of creating image-based models in a feedback loop using graphics hardware. He also pointed me to nonlinear optimization as the feedback mechanism for this endevour. He developed the technique for building the stepsize incrementally using second-order derivative information as described in Section 4.3.3. He also did some of the early work on the SBRDF prototype, which was based on my LFM prototype. He was also responsible for the bulk of the writing in the SIGGRAPH paper.

Sergey Molinov assisted with implementation; particulary, setup of the basic structure for the DirectX implementation. He also implemented the full system for the SBRDF model.

I figured out how to use nonlinear optimization to build an LFM model. I developed the image-streaming framework, and how it could be implemented for graphics hardware. I suggested the SBRDF model as a 2nd kind of model to try. I first implemented a small prototype for building LFM models in OpenGL/Cg using a software emulator, and then implemented the full version using the DirectX structure that was setup by Sergey Molinov.

This dissertation takes a more disciplined approach to the line search (described in Chapter 3) and resampling (Chapter 5). In addition to empirical timing results, I provide theoretical analysis of the scalability of this approach in Chapter 4.1. Error analysis is augmented by a more theoretical approach, including development of conditions for convergence in the presence of numerical error in Chapter 6. I have also developed a system for measuring floating-point error to facilitate this, which is documented in Appendix A.

1.7 Outline Summary

This dissertation begins by examining what kinds of image-based models fit into this framework (Chapter 2). Included are descriptions of the LFM and SBRDF models, which are used as case studies. The chapter ends with a description of how construction of image-based models is treated as a function-fitting process.

Chapter 3 provides background on nonlinear optimization as it applies to this framework. It discusses steepest descent, conjugate gradient, and limited memory BFGS, which are descent techniques that work well for large problems. Penalty methods are discussed as a technique for imposing some amount of smoothness on the model function parameters and for bounding the range of model parameters.

The image-streaming framework is described in Chapter 4. This image-streaming framework is built on the observation that the model can be built from the images incrementally. Quantities such as the error function, which is a measure of how well the model matches the reference images, are expressed as a summation across images.

Chapter 5 is focused on implementation of some of the core functions of the framework in graphics hardware. The model parameters are the texels in texture maps organized according to various spaces of the model, such as surface location and view direction, so these operations generally involved moving data between these different spaces.

Graphics hardware has limited precision, and so Chapter 6 examines the implications of round-off error in this framework. Since graphics hardware floating-point behavior is undocumented, I have developed a technique to measure this behavior, which is documented in Appendix A.

In Chapters 7 and 8 this theory is put to practice in applying the framework to the LFM and SBRDF models respectively. These chapters include analysis of both accuracy and performance; showing favorable performance relative to a conventional implementation on the CPU while maintaining reasonable accuracy.

Work still remains for building on this framework to make it both easier to use and for realtime model generation. Chapter 9 includes suggestions for future work that can achieve these kinds of goals.

Chapter 2

Image-Based Modeling

Image-based modeling refers to a process of constructing a model from images. The purpose of this chapter is to give some background on the kinds of image-based models covered by this dissertation. As discussed in Section 1.4.1, this dissertation concentrates on models with an underlying, known geometric model. The goal is to acquire a model for the appearance of the object. The model should be *spatially-varying* meaning there is some sort of variation over the surface of the object, primarily represented by a set of texture maps. I call these models *textured, image-based models* or *TIM*s.

These models fall into two broad categories: models of exitant radiance and models of surface reflection. The next two sections provide background on radiance and reflectance models, concentrating on issues most relevant to this dissertation. Included are descriptions of *Light Field Mapping* (LFM) (Chen et al., 2002a; Chen, 2002) and the *spatial bidirectional reflectance-distribution function* (SBRDF) (McAllister et al., 2002; McAllister, 2002), which are two diverse image-based modeling approaches used as case studies. The chapter ends with a description of how construction of image-based models is treated as a function-fitting process.

2.1 Radiance Models

Radiance is a measure of light power passing through a differential area in a certain direction (per solid angle) and has units of W/m^2/sr. The plenoptic function (Adelson and Bergen, 1991) describes radiance at a point, and is in general a seven-dimensional function describing the radiance at point (x,y,z) in direction (θ,ϕ), a wavelength λ, at time t:

$$f(\theta,\phi,\lambda,x,y,z,t) \tag{2.1}$$

From now on, wavelength dependence will be handled by using an independent radiance model for each of three specific wavelengths: red, green, and blue, and will therefore be dropped from further notation. Since this dissertation only covers time-independent models, time will also be dropped from subsequent notation.

Radiance along a ray is constant as long as there there are no occluders along this path, making one of the dimensions redundant in this case. This is why position and direction were replaced by a 4D parameterization (u,v,s,t) for Light Field Rendering (Levoy and Hanrahan, 1996) and the Lumigraph (Gortler et al., 1996).

A surface light field (Miller et al., 1998) is parameterized on the surface of the object. I will use $\mathbf{s}$ as the two dimensional parameterization on the surface, and ω for the two-dimensional parameterization of direction. This reduces Equation 2.1 to

$$f(\mathbf{s},\omega). \tag{2.2}$$

A surface light field is more specifically intended to be a model of exitant radiance, meaning it is for radiance that is reflected from the surface, rather than the radiance incident on the surface.

Direct construction of a single function describing the spatially-varying radiance emitted from an entire object is impractical for all but the most trivial models. Therefore the appearance model is typically handled by breaking the surface down into independent *patches*. We might write the function in this manner:

$$f_j(\mathbf{s},\omega). \tag{2.3}$$

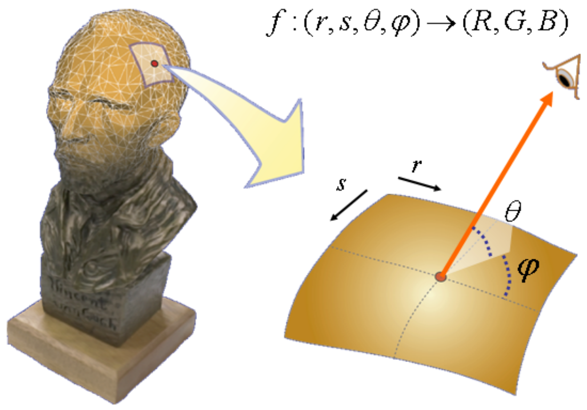


Figure 2.1: *The surface light field parameterization. This parameterization is suitable for representing static sample-based scenes. This figure is from Chen's dissertation (Chen, 2002).*

Of course, this must be a finite, discrete set of functions in practice, and therefore I use the subscript j to denote this fact. A triangle is an example of a patch that is simple and frequently used (Nishino et al., 1999). A more sophisticated example of a patch would be a vertex and the ring of triangles sharing that vertex. This is the formulation used in LFM.

Typically, a surface light field is built by first constructing an independent function for each point on the surface:

$$f_{\mathbf{s}_j}(\omega), \tag{2.4}$$

where the index j enumerates the patch. The framework proposed in this dissertation is quite amenable to this representation. The paper by Wood et al. (Wood et al., 2000) starts with a point-sampled approach of this nature. Their function corresponding to Equation 2.4 is called the "lumisphere". The lumisphere is a fairly data-intensive representation, so Wood et al. compress these data using what they call "function quantization" and "principal function analysis", which are based on vector quantization and principal component analysis (PCA) respectively. The original representation prior to compression consists of independent models at each point on the surface. Because surface light fields are spatially coherent, compression is achieved by taking advantage of this coherence, at the cost of creating an inter-dependence between points on the surface of the object in the compressed representation. Wood et al. used a global compression scheme, whereas this work focuses on more localized coherence.

The smallest patch size considered in this work is a texel-sized or point-sized patch, where each independent model has no explicit dependence on surface parameterization, resulting in a model in the form of Equation 2.4. Later, we will look at the SBRDF which uses a texel-sized patch, but for modeling reflectance.

I chose to look more specifically at LFM as an example of a radiance model. It is a patch-based model of the form in Equation 2.2. Further details of LFM are given in Section 2.4.

2.2 Reflection Models

This dissertation also addresses models of reflectance. Nicodemus et al. (Nicodemus et al., 1977) defined the *bidirectional scattering-surface reflectance-distribution function*, or BSSRDF as the function $S(\omega_i, \mathbf{s}_i, \omega_r, \mathbf{s}_r)$ relating differential radiation flux $d\phi_i$ coming from direction $\omega_i = (\theta_i, \phi_i)$ incident on a surface at point $\mathbf{s}_i = (u_i, v_i)$ to its differential reflected radiance dL_r leaving point $\mathbf{s}_r = (u_r, v_r)$ in the direction $\omega_r = (\theta_r, \phi_r)$:

$$dL_r = S \cdot d\phi_i \tag{2.5}$$

Analogous to the plenoptic function for radiance, and nearly equivalent to the BSSRDF, Debevec et al. proposed the *reflectance field*:

$$R = R(R_i; R_r) = R(\mathbf{s}_i, \omega_i; \mathbf{s}_r, \omega_r), \tag{2.6}$$

which relates the incident light field $R_i(\mathbf{s}_i, \omega_i)$ arriving at a surface to the light field $R_r(\mathbf{s}_r, \omega_r)$ leaving the surface (Debevec et al., 2000). In their discussion of the radiance field, they do not necessarily parameterize on a surface corresponding to a physical surface; this is in contrast to what's implied by the description of the BSSRDF.

Another formulation where exitant radiance at one point is related to incident radiance at another point is the BTF, or *Bidirectional Texture Function* (Dana et al., 1999). The BTF is conceptually a texture image recorded under a number of viewing and lighting directions. There have since been a number of techniques for representing the BTF for the sake of compression and rendering (Müller et al., 2004). Although I have not tried a model of this type, experience with light-field mapping and the SBRDF (described later) leads me to believe that BTF-based models are possible within my framework. The reason is that the effects in BTFs arising from interaction between different points on the surface are somewhat localized. Note that BTFs have been measured by looking at a single material sample, not an object. The framework I propose should be applicable to both capture of a material sample's BTF and an object's BTF. However, more general models that relate incident radiance at a point $\mathbf{s}_i$ to a more distant point $\mathbf{s}_r$ would be difficult to handle in this dissertation's framework.

Nicodemus et al. go on to define a more familiar notion by assuming a homogeneous surface and a large enough area (i.e., no edge effects) with uniform radiance such that the reflectance can be characterized without reference to position on the surface. This results in the *bidirectional reflectance-distribution function*, or BRDF, which depends only on direction, and not on position. The relationship between incident radiance, exitant radiance, and the BRDF is

$$L_r(\omega_r) = \int_{\Omega_i} f_r(\omega_i, \omega_r) L_i(\omega_i) \cos(\theta_i) d\omega_i, \tag{2.7}$$

where $L_i(\omega_i)$ is the radiance incident on a surface with reflectance $f_r(\omega_i, \omega_r)$, resulting in exitant radiance $L_r(\omega_r)$. The BRDF has units of inverse steradians (unit solid angle).

The SBRDF extends the notion of a BRDF to describe spatial variation in the BRDF on the surface of the object (McAllister et al., 2002; McAllister, 2002). It is a discrete sampling of a surface, assigning an independent BRDF to each sample point. Implied is the assumption that each sample lies in a region where the BRDF model is appropriate:

$$f_{\mathbf{s}_j}(\omega_i, \omega_r). \tag{2.8}$$

Again, I have employed the j subscript to indicate a discrete sampling of the surface. This is the underlying notion for the SBRDF, which is discussed further in Section 2.5. The SBRDF is analogous to the point-sampled representation for radiance models given in Equation 2.4.

There is also a notion of patch size associated with reflectance models. Increased patch size is often used to achieve sufficient angular sampling when there are only a few images available. This is achieved by assuming that the shape of the reflectance function is constant over fairly large areas, so that many image pixels can be used as samples for the same function. Early work that used this approach includes that of Sato et al. (Sato et al., 1997), Marschner (Marschner, 1998), and Yu et al. (Yu and Malik, 1998; Yu et al., 1999). A signal-processing approach (Ramamoorthi and Hanrahan, 2001) gives us a formal explanation of the relationship between sampling frequency and the credible frequency in the reconstructed model. The other case where a larger patch size occurs is in the case where BTFs are compressed using matrix factorization. This is analogous to LFM (Section 2.4).

The important difference between models of radiance and models of reflectance is that radiance models, models of $L_r(\omega_r)$, do not allow for relighting, since they are only a function of the outgoing angle, ω_r. Reflectance models, models of $f_r(\omega_i, \omega_r)$, allow for relighting, although they require known incident radiance, $L_i(\omega_i)$, in order to compute the exitant radiance via Equation 2.7. LFM is an example of a radiance model; as such, an LFM model is only valid for the original lighting environment under which it was captured. The SBRDF technique (Section 2.5), by contrast, is a reflectance model, and is designed to accommodate arbitrary lighting environments.

2.3 Decomposition and Analytic Models

My goal in choosing LFM and SBRDF models for case studies was to cover a fair range of different TIMs. In the above sections, we have covered some of the differences: The LFM model is a radiance model, whereas the SBRDF model is a reflectance model. The LFM model is broken down into patches with local surface parameterization, whereas the SBRDF uses a point-sampled approach. The last distinction, discussed here, is that the LFM is built through *decomposition*, whereas the SBRDF is built by fitting to an *analytic* BRDF model. Each of these terms is discussed next.

The raw image data are often too large to be used efficiently at runtime for image synthesis. One approach to image-based modeling is to use data analysis, finding a structure or decomposition of the data to enable effective compression. An example is LFM, which is a statistical model that is built through dimensionality reduction. I use the term *decomposition* for this kind of approach.

Another approach is to choose a function based on the physics or observed characteristics of a material. An *analytic* model assumes some functional form for the BRDF or radiance. The choice of function and parameters of the model are often determined experimentally, which can be a time-consuming and tedious process itself. An analytic model is typically a more compact representation, but tends to be more limiting with respect to the kinds of materials a single choice of function can adequately represent. The SBRDF formulation is built on analytic models to store per-texel BRDFs.

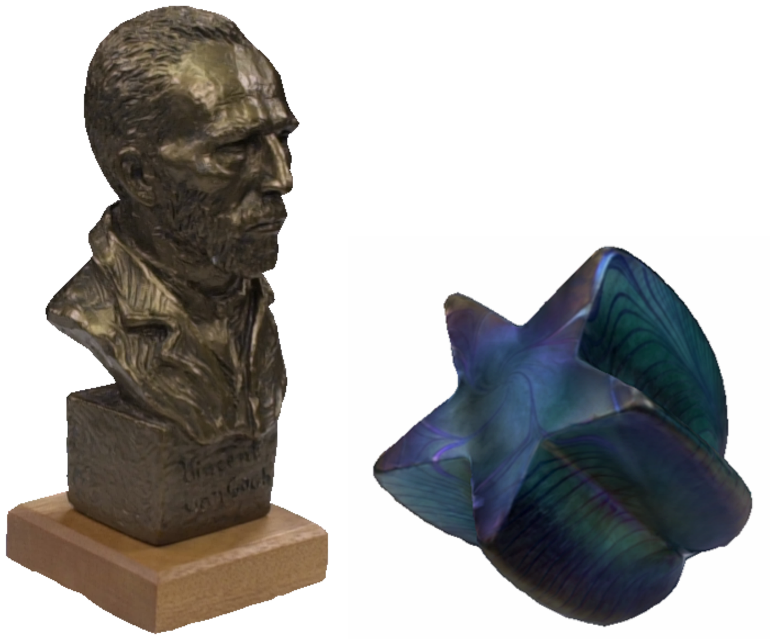

(a) *Bust* **(b)** *Star*

Figure 2.2: *Light Field Mapping example models.*

2.4 Light Field Mapping

Light Field Mapping (LFM) encodes the view-dependent appearance of an object (Chen et al., 2002a; Chen, 2002). It does not encode light-dependence; in other words, it assumes fixed lighting conditions that cannot be changed. Diffuse effects are treated as constants and are subtracted from the light-field before creating the LFM approximation. Details on how this diffuse component is derived are outlined in Section 5.8, as it is not an LFM-specific process. The remainder of this discussion assumes the constant term has already been subtracted.

The LFM approximation is designed for use with a triangular mesh and partitions the surface light field data (Equation 2.2) around each vertex. This is analogous to the lighting and texturing model used by OpenGL, where the Phong model is applied to each vertex, and its interpolated results are either added to, or modulated by the texture of the triangles surrounding it (Segal and Akeley, 1999).

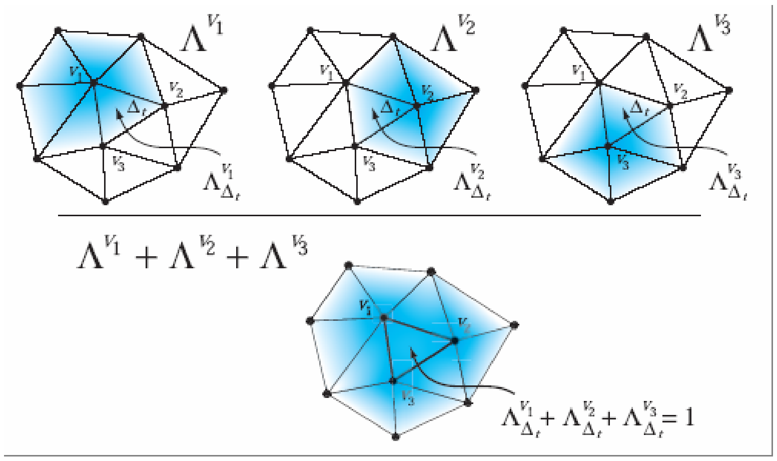

Λ^{v_1}
Λ^{v_2}
Λ^{v_3}
v_1
v_2
v_3
Δ_t
$\Lambda^{v_1}_{\Delta_t}$
$\Lambda^{v_2}_{\Delta_t}$
$\Lambda^{v_3}_{\Delta_t}$
$\Lambda^{v_1} + \Lambda^{v_2} + \Lambda^{v_3}$
$\Lambda^{v_1}_{\Delta_t} + \Lambda^{v_2}_{\Delta_t} + \Lambda^{v_3}_{\Delta_t} = 1$

Figure 2.3: *The finite support of the hat functions Λ^{v_j} around vertex $v_j, j = 1,2,3$. $\Lambda^{v_j}_{\triangle_t}$ denotes the portion of Λ^{v_j} that correspond to triangle $\triangle_t$. Functions Λ^{v_1}, Λ^{v_2}, and Λ^{v_3} add up to one inside $\triangle_t$. This figure is from Chen's dissertation (Chen, 2002).*

The light field unit corresponding to each vertex is called the *vertex light field* and for vertex v_j is denoted as $f_j(\mathbf{s}, \omega)$, where $\mathbf{s}$ refers to surface location in the neighborhood of the vertex, and ω refers to the view direction in a coordinate system associated with the vertex.

Partitioning is achieved by weighting the radiance function

$$f_j(\mathbf{s}, \omega) = \Lambda^{v_j}(\mathbf{s})\, f(\mathbf{s}, \omega) \tag{2.9}$$

where Λ^{v_j} is the barycentric weight of each point in the ring of triangles relative to vertex v_j. Rendering a triangle is accomplished by summing the contributions of the three vertex light fields of the triangle's vertices as shown in Figure 2.3.

Consider the case where the function $f_j(\mathbf{s}, \omega)$ is tabulated for a fixed set of surface locations and viewing directions. We can write this table as a matrix by considering each row to be a surface location (texel), and each column to be a view direction. The matrix for m surface locations and n views would then look like this:

$$F_j = \begin{bmatrix} f_j(\mathbf{s}_1, \omega_1) & \cdots & f_j(\mathbf{s}_1, \omega_n) \\ \vdots & \ddots & \vdots \\ f_j(\mathbf{s}_m, \omega_1) & \cdots & f_j(\mathbf{s}_m, \omega_n) \end{bmatrix}. \tag{2.10}$$

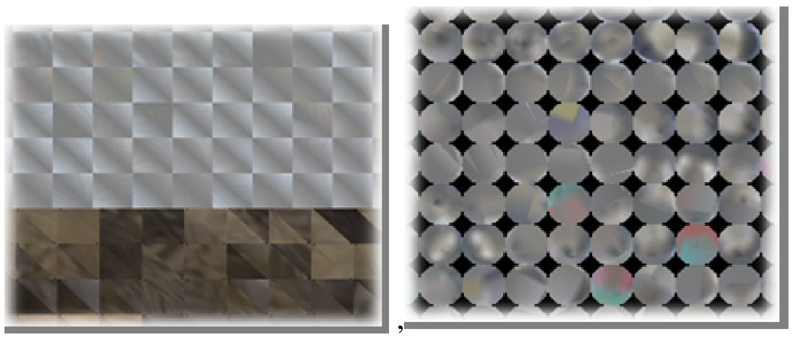
For adequate coverage, there will typically be on the order of $m = 1000$ and $n = 1000$. The primary means of compression in light-field mapping is to approximate this large matrix using outer products of vectors. The approximation is

$$f_j(\mathbf{s}, \omega) \approx m_j(\mathbf{p}, \mathbf{x}) = \sum_{k=1}^{K} g_k^{v_j}(\mathbf{s})\, h_k^{v_j}(\omega) \tag{2.11}$$

where surface maps $g_k^{v_j}(\mathbf{s})$ and view maps $h_k^{v_j}(\omega)$ can be thought of as lookup functions into the outer product vectors. The vector $\mathbf{x} = (\mathbf{s}, \omega)$ is the vector of inputs to the model function. The parameters $\mathbf{p}$ are the elements of the vectors in $g_k^{v_j}(\mathbf{s})$ and $h_k^{v_j}(\omega)$. The distinction between *inputs* and *parameters* will be made clear in Section 2.6.

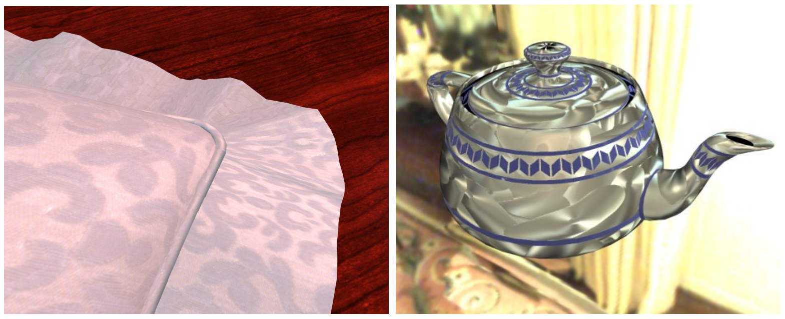
You can see that if K is small enough, and indeed $K \leq 3$ is typically quite sufficient in practice, this achieves considerable compression. In other words, instead of storing a $1000 \times 1000 = 1,00,000$ element matrix, it is sufficient to store three vectors of length 1000 and three vectors of length 1000 for a total of $3 \times 1000 + 3 \times 1000 = 6000$ elements.

The factorization is performed by using either principal component analysis (PCA) (Bishop, 1995) or non-negative matrix factorization (NMF) (Lee and Seung, 1999). PCA is performed by using the power iteration algorithm in order to get the dominant eigenvectors after subtracting the mean, resulting in some negative values (Golub and Loan, 1991). By contrast, NMF results in non-negative surface and view map elements, as suggested by the technique's name. Note that although not necessary, the minimum is typically subtracted from the light field before computing an NMF, as subtracting the



(a) *Diffuse (bottom) and surface maps* **(b)** *View maps*

Figure 2.4: *Light Field Mapping breaks surface light fields down into lower dimensional, principal components called surface maps and view maps, which are stored in textures.*



(a) *Pillow* **(b)** *Teapot*

Figure 2.5: *Renderings using SBRDF material samples*

minimum makes for better approximation. Although not relevant to this dissertation, it should be noted that PCA generates a better approximation than NMF for the same number of approximation terms.

The surface map and view map are treated as vectors with respect to the mathematics, but are eventually stored in 2D texture maps (2.4). $g(\mathbf{s})$ and $h(\omega)$ are evaluated through conventional texture mapping using $\mathbf{s} \in \Re^2$ and $\omega \in \Re^2$ as texture coordinates.

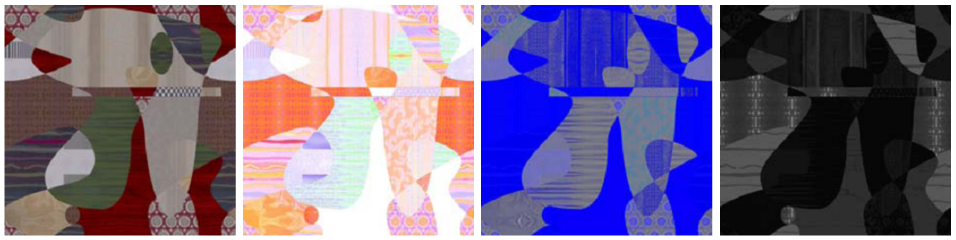

(a) *Diffuse albedo* **(b)** *Lobe albedo* **(c)** *Shape* **(d)** *Exponent*

Figure 2.6: *SBRDF models encode spatially-varying reflectance by storing the parameters of analytical BRDF models in texture maps. The diffuse albedo is ρ_d, the lobe albedo is $\rho_{s,j}$, the shape is $[C_{x,j}, C_{y,j}, C_{z,j}]$, and the exponent is n_j.*

2.5 SBRDF

A BRDF describes how light reflects from a point on the surface of an object, and is a function of lighting and viewing conditions. McAllister et al. developed a system to capture and render spatially varying reflectance by taking a discrete sampling of the surface, and applying an independent BRDF at each sample point (McAllister et al., 2002; McAllister, 2002). The discrete sampling corresponds to texel locations in a texture map. They call this type of model a *spatial bidirectional reflectance-distribution function* or SBRDF.

They chose the Lafortune model (Lafortune et al., 1997) to approximate the BRDF at each texel. It consists of the sum of a single diffuse term and a number of specular lobes.

$$f_r(\mathbf{x}) \approx m(\mathbf{p},\mathbf{x}) = \rho_d + \sum_j \rho_{s,j}(C_{x,j}\, u_x\, v_x + C_{y,j}\, u_y\, v_y + C_{z,j}\, u_z\, v_z)^{n_j} \tag{2.12}$$

The parameters $\mathbf{p}$ in the above function are ρ_d and $\rho_{s,j}$, which define the diffuse and specular albedo, $C_{x,j}$, $C_{y,j}$, $C_{z,j}$, which define the specular peak direction, and n_j, the specular exponent. The input variables, $\mathbf{x}$, are u_x, u_y, u_z, v_x, v_y, and v_z. They are projections of the viewing and lighting directions on the local coordinate system. This model is discussed further in Chapter 8.

Although the reflectance function allows for any kind of lighting environment, the data used to construct the model are typically acquired in a controlled environment with a single light source approximating a point light source. The integral in the reflectance equation (2.7) reduces to

$$L_r(\omega_r) = f_r(\omega_i, \omega_r) L_c \cos(\theta_i)/r^2, \tag{2.13}$$

where L_c is a constant, and r is the distance from the light to the point on the surface.

McAllister et al. collected SBRDFs from material samples. This dissertation looks at collecting SBRDFs from objects, which introduces further complications that are discussed in Chapter 5. Lensch et al. also fit spatially-varying Lafortune models to 3D objects (Lensch et al., 2001). Rather than starting with per-texel models, they start with a single model and split as necessary according to the co-variance matrix that arises in their fitting process.

In more recent work (Lensch et al., 2003) they explore how to adaptively measure the spatially-varying BRDF of an object. Here they use graphics hardware to compute error and gradient information on the texture atlas of the object, similar to the approach taken in this dissertation, although all other optimization-related computations are performed on the CPU. By contrast, this dissertation is focused on techniques where all computations can be performed on graphics hardware.

2.6 Image-Based Modeling as Nonlinear Optimization

This section describes the link between image-based modeling and nonlinear optimization. The goal of image-based modeling is to compute parameter values of a mathematical model that best fit the image data. The model should synthesize new images from novel views, but it should also reproduce the reference images with minimal error. An image-based model requires choice of an appropriate representation; then a computation is required to construct the model in this representation. We can express this as a function fitting process, where the image-based model's representation is our function, and the images contain the data to which we want to fit our function. An image-based model can be expressed as a function of two vectors: $m(\mathbf{p},\mathbf{x})$. The vector $\mathbf{x}$ is the vector of inputs to our function. For LFM, the $\mathbf{x}$ are the texture coordinates and view direction. For SBRDF it would be the view and light directions. The vector $\mathbf{p}$ is the vector of parameters used to fit the model to the images.

A *residual* is the difference between the i^{th} datapoint and the function's approximation for the data point

$$r_i(\mathbf{p}) = m(\mathbf{p},\mathbf{x}_i) - y_i. \tag{2.14}$$

The i^{th} datapoint consists of sample radiance y_i, and the corresponding inputs, $\mathbf{x}_i$. A measure of the function fit is the Euclidian or L_2 norm:

$$E(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{S} r_i(\mathbf{p})^2, \tag{2.15}$$

where S denotes the number of samples (datapoints). The goal is to find the set of parameters, $\mathbf{p}$, that minimizes this error function. With the exception of the Lambertian model of reflection, model functions, $m(\mathbf{p},\mathbf{x})$ are typically nonlinear with respect to $\mathbf{p}$. Nonlinear models imply a nonlinear error function; therefore, this dissertation adopts nonlinear optimization to solve the problem.

The nonlinear optimization techniques considered in this dissertation are limited in that they only work for functions that are continuous, differentiable, and have a continuous derivative (C^1). This limitation can be met by a proper choice of model, $m(\mathbf{p},\mathbf{x})$. The appearance functions used in image-based modeling generally meet this criterion. The nonlinear optimization techniques adopted in this work are only guaranteed to find a local minimum. This second limitation must be addressed by careful choice of an initial guess, possibly trying a number of initial guesses.

The next chapter covers nonlinear optimization techniques that can be applied to make this fit in order to construct an image-based model.

Chapter 3

Nonlinear Optimization

Our goal is to find the vector $\mathbf{p}$ that minimizes $f(\mathbf{p})$[1], which is known in the literature as the *minimization* problem. The function f is called the *objective function*. Later, we will look more specifically at an objective function that measures error of an image-based model with respect to reference images, but for now there is no need to get that specific. This chapter, and indeed this dissertation, only considers functions with first order continuity (C^1). Constrained optimization will not be considered except that a penalty term will be used to steer the solution into the feasible region.

The goal of this chapter is to give a brief introduction to nonlinear optimization, with a focus on nonlinear objective functions. Further details on the concepts discussed here can be found in a textbook on numerical nonlinear optimization such as *Numerical Optimization* by Nocedal and Wright (Nocedal and Wright, 1999) or *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* by Dennis and Schnabel (Dennis and Schnabel, 1996), and *Practical Optimization* by Gill, Murray and Wright (Gill et al., 1981).

[1]Nonlinear optimization literature conventionally uses $\mathbf{x}$ where I have used $\mathbf{p}$. The rationale for my choice was to use $\mathbf{x}$ to indicate inputs to the model such as view direction, light direction and surface parameterization, and to use $\mathbf{p}$ to indicate parameters of the model such as specular coefficients and specular exponents.

3.1 Necessary Conditions

Ideally, the objective function would be convex. The nice characteristic of convex functions is that the local minimum is also the global minimum. In general, this is not the case, even though the techniques discussed here work on the basis of convexity. In other words, these techniques find a local minimum, which may or may not be the global minimum. A local minimum is a vector $\mathbf{p}^*$ such that $f(\mathbf{p}^*) \leq f(\mathbf{p})$ for all $\mathbf{p}$ in a neighborhood of $\mathbf{p}^*$.

The most important conditions for a minimum, called the *first order necessary conditions*, are

> If $\mathbf{p}^*$ is a local minimizer and $f(\mathbf{p})$ is continuously differentiable in an open neighborhood of $\mathbf{p}^*$, then $\nabla f(\mathbf{p}^*) = 0$ (Nocedal and Wright, 1999).

These are not sufficient, as it is also true for a local maximum or a local saddle point, for example. However, they are the primary conditions used in finding a minimum.

The *second order necessary conditions* are

> If $\mathbf{p}^*$ is a local minimizer of $f(\mathbf{p})$ and $\nabla^2 f(\mathbf{p})$ is continuous in an open neighborhood of $\mathbf{p}*$, then $\nabla f(\mathbf{p}^*) = 0$, and $\nabla^2 f(\mathbf{p}^*)$ is positive semidefinite (Nocedal and Wright, 1999).

The condition involving the second derivative essentially requires the curve to be bowed upward. The second derivative is a matrix, as it involves all combinations of partial derivatives with respect to two variables. It is called the *Hessian* of $f(\mathbf{p})$, and is often written as $\mathbf{H} f(\mathbf{p})$. Positive semi-definite means that $\mathbf{v}^T \mathbf{H} f(\mathbf{p}) \mathbf{v} \geq 0$ for any non-zero vector $\mathbf{v}$.

3.2 Line Search Approaches

Numerical techniques for nonlinear optimization are iterative in nature. They begin with some initial guess, $\mathbf{p}_0$, and progress at each iteration to a better guess, $\mathbf{p}_k$, that decreases the function. I have chosen to concentrate on the class of techniques referred to as line-search approaches. There are two parts to finding the next guess in a line-search approach: choosing a direction, and performing a line-search along that direction. The expression for this is

$$\mathbf{p}_{k+1} = \mathbf{p}_k + \alpha_k \mathbf{d}_k, \tag{3.1}$$

where α_k is the scalar used in the search along the direction $\mathbf{d}_k$. Section 3.3 discusses how to choose a direction, and Section 3.4 discusses how to choose a stepsize.

3.3 Direction Choices

A *descent direction* is defined mathematically as any direction $\mathbf{d}$ that satisfies:

$$\mathbf{d}^T \nabla f(\mathbf{p}) \leq 0 \tag{3.2}$$

If the function and its derivative are continuous, a small enough step along this direction must generate a decrease in the function. This condition leaves considerable leeway in a specific choice for the direction of the line search. This section describes a number of algorithms for choosing a particular direction.

3.3.1 Steepest Descent

The safest choice is the *steepest descent* or *gradient descent* direction, $-\nabla f(\mathbf{p})$, but this can result in slow convergence in many cases. Nonetheless, it is used fairly often because of its ease of use and robustness. Often there is a trade-off between robustness and performance, and steepest descent represents the robust end of the spectrum.

3.3.2 Conjugate Gradient Method

The conjugate gradient differs from the steepest descent method in that it takes into consideration properties of previous search directions. It is based on the idea that once a search has been made along some direction of a convex, quadratic function, the component of the gradient in that direction is zero, and should remain so. Any successive points should not have a gradient with a component in the direction already minimized. In fact, if a quadratic function has *n* dimensions, the conjugate gradient method is guaranteed to achieve the minimum after *n* direction searches (barring roundoff error issues). In practice, the algorithm converges in much less than *n* steps for problems where *n* is large.

The conjugate gradient method was originally developed for solving linear systems, and is still a popular technique for this application today. However, this dissertation focuses on the application of conjugate gradient to nonlinear systems. The linear conjugate gradient method can be viewed as a special case of the nonlinear conjugate gradient method, but this work does not address the specifics of the linear case.

The first step in the conjugate gradient method is in the direction of the negative gradient, just as in steepest descent. Subsequent directions are computed as a weighting of the previous search direction and the negative of the current gradient:

$$\mathbf{d}_{k+1} = -\nabla f(\mathbf{p}_k) + \beta_k \mathbf{d}_k. \tag{3.3}$$

The weighting factor, β_k, is computed as:

$$\beta_k = \frac{\|\nabla f(\mathbf{p}_k)^T \nabla f(\mathbf{p}_k)\|}{\|\nabla f(\mathbf{p}_{k-1})^T \nabla f(\mathbf{p}_{k-1})\|}. \tag{3.4}$$

Polak and Ribière proposed another form of this function that seems to perform better based on numerical experience:

$$\beta_k = \frac{\|\nabla f(\mathbf{p}_k)^T(\nabla f(\mathbf{p}_k) - \nabla f(\mathbf{p}_{k-1}))\|}{\|\nabla f(\mathbf{p}_{k-1})^T \nabla f(\mathbf{p}_{k-1})\|}. \tag{3.5}$$

The conjugate gradient method generally has better convergence properties than steepest descent. Although more powerful techniques exist, they cost more in terms of memory. The conjugate gradient requires storage only for the current and previous directions and gradients, which means that it is $O(n)$ in storage requirements. For this reason, it is often the technique of choice for large problems.

3.3.3 Newton's Method

Neither of the above methods use second-order derivative information. The Newton method, which is described here, is considered the golden standard in terms of convergence. It uses second-order derivative information to solve a quadratic approximation, thus achieving a quadratic convergence for functions that are C^3.

The goal is to find $\mathbf{p}^*$ such that $\nabla f(\mathbf{p}^*) = 0$. We can write the first order Taylor expansion for $\nabla f(\mathbf{p})$ by expanding around $\nabla f(\mathbf{p}_k)$:

$$\nabla f(\mathbf{p}) = \nabla f(\mathbf{p}_k) + \mathbf{J}\nabla f(\mathbf{p}_k)(\mathbf{p} - \mathbf{p}_k) + \varepsilon_1(\mathbf{p};\mathbf{p}_k), \tag{3.6}$$

where $\varepsilon_1(\mathbf{p};\mathbf{p}_k)$ represents the remaining terms in the Taylor expansion and therefore represents the error in the approximation. $\mathbf{J}$ is the Jacobian operator. The Jacobian of the gradient of a function is the Hessian of the function. Using this substitution, as well as defining $\mathbf{d} = (\mathbf{p} - \mathbf{p}_k)$, the first-order Taylor expansion tells us that the gradient is zero where

$$\mathbf{H}f(\mathbf{p}_k)\mathbf{d} = -\nabla f(\mathbf{p}_k). \tag{3.7}$$

$\mathbf{H}$ is the Hessian operator. The solution to this system, $\mathbf{d}$, is the vector that takes us to the minimum in this quadratic approximation of the function $f(\mathbf{p})$.

3.3.4 BFGS and Limited Memory BFGS

Newton's method has two critical short-comings: it requires evaluation of the Hessian at each iteration, which can be quite expensive, and it is only robust when the iterate is near the minimum. One popular set of alternatives are the *quasi-Newton* methods. The *secant condition* is similar to Equation (3.7):

$$\mathbf{B}_k f(\mathbf{p}_k)(\mathbf{p}_k - \mathbf{p}_{k-1}) = \nabla f(\mathbf{p}_k) - \nabla f(\mathbf{p}_{k-1}). \tag{3.8}$$

Here, the assumption is that $\mathbf{p}_k$ and $\mathbf{p}_{k-1}$ are two iterates already computed. The goal is to choose a $\mathbf{B}_k$ that satisfies this condition in the same way that the Hessian would as $p_k - p_{k-1} \to 0$. Once $\mathbf{B}_k$ is generated from $\mathbf{p}_k$ and $\mathbf{p}_{k-1}$, it can be used to compute the next iterate. Since $\mathbf{B}_k$ is $n \times n$, but there are only n equations, there are a number of choices for $\mathbf{B}_k$. It can be chosen to be positive definite (and non-singular) so that the next direction computed from $\mathbf{d}_k = -\mathbf{B}_k^{-1}\nabla f(\mathbf{p}_k)$ is a descent direction. Another goal is that $\mathbf{B}_k$ approach $\mathbf{H}f(\mathbf{p}_k)$ as $k \to \infty$.

Although the formulation above only mentions two iterations in the generation of $\mathbf{B}_k$, it is quite possible to build $\mathbf{B}_k$ incrementally over all iterations. In other words, $\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{E}_k$, where $\mathbf{E}_k$ represents the update computed at each iteration. The most popular update is called the *Broyden-Fletcher-Glodfarb-Shanno*, or BFGS update. Since the update $\mathbf{d}_k = -\mathbf{B}_k^{-1}\nabla f(\mathbf{p}_k)$ requires a matrix inversion, $\mathbf{A}_k = \mathbf{B}_k^{-1}$ is often kept and updated instead.

In the limited-memory BFGS (L-BFGS) method, this idea of building $\mathbf{B}_k$ or $\mathbf{A}_k$ across all iterations is relaxed. This reduces space requirements, at the expense of slower convergence to the solution. Rather than storing either a full matrix $\mathbf{B}_k$ or $\mathbf{A}_k$, which are each $n \times n$ in size, the L-BFGS method stores a few vectors of size n that can be used to approximate $\mathbf{B}_k$ or $\mathbf{A}_k$. This reduces storage requirements quite drastically for large problems.

Remember that $\mathbf{A}_k$ is multiplied by the gradient to get a new search direction. L-BFGS works by never creating a matrix, but by calculating the next direction through a series of operations involving only vectors. Algorithm 1 describes how the search direction, $\mathbf{d}_k = -\mathbf{A}_k \nabla f_k$ is computed from a set of vectors kept from the previous m calculations. To simplify notation, it uses the following shorthand and definitions: $f_k = f(\mathbf{p}_k)$, $\nabla f_k = \nabla f(\mathbf{p}_k)$, $\mathbf{s}_k = \mathbf{p}_{k+1} - \mathbf{p}_k$, $\mathbf{y}_k = \nabla f_{k+1} - \nabla f_k$, and $\rho_k = \mathbf{y}_k^T \mathbf{s}_k$. It is taken from the textbook by Nocedal and Wright (Nocedal and Wright, 1999). Note that $\mathbf{A}_k^0$ is the

Algorithm 1 L-BFGS Computation of the Search Direction

```
Require: inputs: $m > 0$, $k \geq 0$, $\mathbf{s}_i$, $\mathbf{y}_i$, where $i = k-m, \ldots, k-1$
  $\mathbf{q} \leftarrow \nabla f_k$
  for $i = k-1, k-2, \ldots, k-m$ do
    $\eta_i \leftarrow \rho_i \mathbf{s}_i^T \mathbf{q}$
    $\mathbf{q} \leftarrow \mathbf{q} - \eta_i \mathbf{y}_i$
  end for
  $\mathbf{r} \leftarrow \mathbf{A}_k^0 \mathbf{q}$
  for $i = k-m, k-m+1, \ldots, k-1$ do
    $\beta \leftarrow \rho_i \mathbf{y}_i^T \mathbf{r}$
    $\mathbf{d} \leftarrow \mathbf{d} + \mathbf{s}_i(\eta_i - \beta_i)$
  end for{The search direction is $\mathbf{d} = -\mathbf{r} = -\mathbf{A}_k \nabla f_k$}
```

only matrix in Algorithm 1. By choosing a particular sparsity pattern, say a diagonal matrix, $\mathbf{A}_k^0$ can have both a $O(n)$ storage and a matrix-vector product that costs $O(n)$. A typical choice is to compute $\mathbf{A}_k^0$ at each iteration using:

$$\mathbf{A}_k^0 = \gamma_k I \tag{3.9}$$

where:

$$\gamma_k = \frac{(\mathbf{s}_{k-1})^T \mathbf{y}_{k-1}}{(\mathbf{y}_{k-1})^T \mathbf{y}_{k-1}} \tag{3.10}$$

The description of L-BFGS (Algorithm 2) includes mention of the Wolfe conditions for testing the stepsize. They are discussed in Section 3.4.

Algorithm 2 L-BFGS

```
Require: Starting point $\mathbf{p}_0$, integer $m > 0$
  $k \leftarrow 0$
  repeat
    Choose $\mathbf{A}_k^0$ (for example, by using (3.9))
    Compute $\mathbf{d}_k \leftarrow -\mathbf{A}_k \nabla f_k$ from Algorithm1
    $\mathbf{p}_{k+1} \leftarrow \mathbf{p}_k + \alpha_k \mathbf{d}_k$, where $\alpha_k$ is chosen to satisfy the Wolfe conditions (Section 3.4)
    if $k > m$ then
      Discard $\mathbf{s}_{k-m}$ and $\mathbf{y}_{k-m}$ from storage;
    end if
    Compute and save $\mathbf{s}_k \leftarrow \mathbf{p}_{k+1} - \mathbf{p}_k$, $\mathbf{y}_k = \nabla f_{k+1} - \nabla f_k$.
    $k \leftarrow k+1$
  until Convergence
```

3.4 Line Search

Once we have chosen a direction, we have a choice in how far we want to move along that direction in hopes of finding a better guess for the minimum. It would seem that as long as the function value decreases, the sequence will eventually converge to the minimum. This is actually not always true if too little progress is made at each step, either with regards to the stepsize, or the amount of decrease in the function. The *Wolfe conditions* give stronger criteria for how much the function should decrease, as well as a minimum stepsize to avoid this problem.

The Wolfe conditions use two fixed parameters $0 < \sigma_1 < \sigma_2 < 1$. According to Nocedal and Wright (Nocedal and Wright, 1999), σ_1 is often chosen to be quite small, giving a typical value of 10^{-4}, and σ_2 is 0.9 for Quasi-Newton, or 0.1 for nonlinear conjugate gradient. The Wolfe conditions are:

$$f(\mathbf{p}_k+\alpha_k\mathbf{d}_k) \leq f(\mathbf{p}_k)+\sigma_1\alpha_k\nabla f(\mathbf{p}_k)^T\mathbf{d}_k, \tag{3.11}$$

$$\nabla f(\mathbf{p}_k+\alpha_k\mathbf{d}_k)^T\mathbf{d}_k \geq \sigma_2\nabla f(\mathbf{p}_k)^T\mathbf{d}_k. \tag{3.12}$$

The first condition ensures adequate decrease in the function at each step, and the second condition keeps the stepsizes from getting too small. Notice that there is no prescription on how to choose a specific α; the Wolfe conditions only provide an evaluation of a candidate α once it is chosen. The remainder of this section describes algorithms for choosing a candidate α.

3.4.1 Backtracking

The *Backtracking* method is the simplest approach for choosing a specific stepsize. A choice must be made for the first α. At each subsequent iteration, the first Wolfe condition (3.11) is checked. If the first condition is met, the step is taken. If it is not, α is replaced by $\lambda\alpha$, where λ is some factor less than one, until the first condition is met. Note that this method does not explicitly check the second condition, where a derivative evaluation at the trial step would be required. The only control on the stepsize with respect to the second condition is that a failed step with respect to the first condition will be followed with a candidate step is at least within some constant factor (λ) of a stepsize large enough *not* to satisfy the first condition.

3.4.2 One-Sided Quadratic Approximation Using the Hessian

There is an analytic expression for the stepsize using a quadratic approximation:

$$\alpha_k = -\frac{\nabla f(\mathbf{p}_k)^T\mathbf{d}_k}{\mathbf{d}_k^T\mathbf{H}f(\mathbf{p}_k)\mathbf{d}_k} \tag{3.13}$$

Since this is based on an approximation of the minimum, it needs to be checked against the Wolfe conditions in practice. It is "one-sided" in the sense that it only uses information from a single point, $\mathbf{p}_k$. The next technique uses information from two points.

Note that this expression involves the Hessian of the objective function, which in general is impractical to compute, particularly for large problems. However, for TIMs the cost of using the Hessian in this context is not much more than the gradient. In the case of LFM, for example, each data sample contributes two values to the gradient and four values to the Hessian. Section 4.3.3 shows how this stepsize can be used in practice by building the denominator in an incremental fashion.

3.4.3 Two-Sided Quadratic Approximation

In this case, the quadradic fit is constructed from the gradient and two function evaluations along the line, therefore avoiding need of the Hessian. It is written in terms of the one-dimensional line search: $\phi(\alpha) = f(\mathbf{p}+\alpha\mathbf{d})$. Here we consider the quadratic approximation using $\phi(0) = f(\mathbf{p})$, $\phi'(0) = \nabla f(\mathbf{p})^T\mathbf{d}$ and the function value at some other point, $\phi(\hat{\alpha}) = f(\mathbf{p}+\hat{\alpha}\mathbf{d})$. The quadratic function defined by these values is:

$$q(\alpha) = \left(\frac{\phi(\hat{\alpha}) - \phi'(0)\hat{\alpha} - \phi(0)}{\hat{\alpha}^2}\right)\alpha^2 + \phi'(0)\alpha + \phi(0). \tag{3.14}$$

The α that corresponds to the minimum of this function is:

$$\alpha_{\min} = -\frac{\phi'(0)\hat{\alpha}^2}{2[\phi(\hat{\alpha}) - \phi'(0)\hat{\alpha} - \phi(0)]} \tag{3.15}$$

Note that this is only a minimum if $q''(0) > 0$. However, since $q'(0) < 0$, we know that the function at least decreases as $\alpha \to 0$ in this case, which motivates Algorithm 3.

Algorithm 3 Quadratic Line Search

```
Require: $\hat{\alpha} = 1$ and $\gamma < 1$.
  repeat
    Compute $\alpha_{\min}$ as per (3.15) above.
    If $\alpha_{\min} \in (0, \hat{\alpha})$, set $\hat{\alpha} = \alpha_{\min}$; else set $\hat{\alpha} = \gamma\hat{\alpha}$
  until The first Wolfe condition (3.11) is satisfied for $\hat{\alpha}$.
  Set $\alpha_k = \hat{\alpha}$.
```

3.5 Function Fitting

Up to this point, the discussion has been about general nonlinear optimization problems. Now we move to a more particular form of objective function typically used in fitting a function to a set of empirical data. As discussed in Section 2.6, the goal is to find the set of parameters, $\mathbf{p}$, that minimize the error function:

$$E(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{n_s} r_i(\mathbf{p})^2, \tag{3.16}$$

where

$$r_i(\mathbf{p}) = m(\mathbf{p}, \mathbf{x}_i) - y_i. \tag{3.17}$$

is the residual, or difference, between the model function $m(\mathbf{p}, \mathbf{x}_i)$ and the data, which consists of $(\mathbf{x}_i, y_i)$ pairs. The function $E(\mathbf{p})$ constitutes the objective function in the minimization process, which we have previously denoted as $f(\mathbf{p})$ in the general case.

Minimizing the above error function is a *nonlinear least-squares* problem, for which there are special-purpose techniques. A well known method is the Levenburg-Marquardt method, which is a trust-region method. It has been used in the field of image-based modeling in the past. For example, it has been used in fitting to the model proposed by Lafortune (Lafortune et al., 1997) in a number of papers (Lafortune et al., 1997; McAllister, 2002; Lensch et al., 2001). Unfortunately, this technique is $O(n^2)$ in storage requirements, and due to its unsuitability for solving large systems, is not discussed further.

In this work I consider solving the nonlinear least-squares using the nonlinear optimization techniques already discussed: steepest descent, conjugate gradient and L-BFGS. In the next section, the error function will be augmented by penalty terms to constitute a new objective function.

3.6 Penalty Method

Both hard and soft constraints are considered for this framework. A hard constraint is a condition on the model parameters that must be met. A soft constraint might bias the solution towards a desirable condition, but does not form a strict requirement. Both of these kinds of constraints can be handled by a *penalty method*, which adds an additional penalty term to the unconstrained objective function. The total objective function is

$$f(\mathbf{p}) = E(\mathbf{p}) + \mu P(\mathbf{p}), \tag{3.18}$$

where $P(\mathbf{p})$ is the penalty term and μ is used to weight the penalty term. I have used $E(\mathbf{p})$ to denote the original, unconstrained objective function, as this work concentrates on fitting data according to some error metric.

For hard constraints, the standard approach using the penalty method is to start by solving the unconstrained problem ($\mu = 0$). If the solution is within the feasible set, then the solution is also the solution to the constrained problem. If it is not, then the penalty term is added, and the process is continued. As long as the solution is outside of the feasible region, μ is increased, and the unconstrained problem with penalty term is solved again.

The original penalty method is not a preferred technique for solving problems with hard constraints. Even so, I have found this to be sufficient, and do not consider other approaches. In this framework, the parameter μ is set once, and the unconstrained problem is solved with the penalty term a single time. If a given constraint is intended to be strictly enforced, the associated parameter is simply clamped to the feasible region as needed.

There are basically two kinds of penalty terms of interest in this work, and are the subject of the next two subsections. Bounds constraints were applied as both soft and hard constraints. Regularization, on the other hand, only makes sense in terms of a soft constraint.

3.6.1 Bounds Constraints

A bounds constraint is one in which a parameter is restricted to some range, specified as $l_i \leq p_i \leq u_i$, where l_i is the lower bound and u_i the upper bound for parameter p_i. An example of this kind of constraint is that exponents be non-negative, i.e. $l_i = 0$, $u_i = \infty$. Bounds constraints are handled using a quadratic penalty term in this work:

$$P(\mathbf{p}) = \sum_i (\max(l_i - p_i, 0))^2 + (\max(p_i - u_i, 0))^2. \tag{3.19}$$

Other penalty functions, such as a logarithmic barrier function, would also work in this framework, but I did not test any others.

3.6.2 Regularization

The other major constraint considered in this work is regularization. It is, by nature, a soft constraint. McCool et al. used regularization in constructing BRDF factorizations in order to ensure a measure of conditioning of the system, as well as to control smoothness of the solution (McCool et al., 2001). This work follows suit. Regularization is primarily intended to accommodate under-sampled regions of the data space by interpolation and extrapolation in the parameter space. A negative side-effect is that regularization tends to filter out high frequencies. A regularization constraint is expressed as

$$\mathbf{Lp} = 0. \tag{3.20}$$

For the one-dimensional case, the matrix **L** is

$$\begin{bmatrix} 1 & -1 & 0 & & & \cdots & 0 \\ -1 & 2 & -1 & 0 & & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ \vdots\ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots\vdots \\ 0 & \cdots & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & & 0 & -1 & 2 & -1 \\ 0 & \cdots & & & 0 & -1 & 1 \end{bmatrix}$$

For a two-dimensional case, the pattern in the matrix L depends on the mapping of **p** to two dimensions. However, we can at least write the basic operator in the two-dimensional case. I have included the one-dimensional operator as well in order to see how this relates to the matrix **L**.

One-dimensional operator: $\begin{bmatrix} -1 & 2 & -1 \end{bmatrix}$

Two-dimensional operator: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Using the penalty method, we need to incorporate this constraint into the objective function. A quadratic penalty term is formed using:

$$P(\mathbf{p}) = (\mathbf{Lp})^T(\mathbf{Lp}). \tag{3.21}$$

3.7 Summary

The goal of this chapter was to provide the necessary background to understand the next chapter, where I present the image-streaming implementation of nonlinear optimization for solving the image-based modeling problem. This chapter started with a discussion of the nonlinear optimization techniques I have considered. Function fitting was reviewed in terms of this background. The discussion of penalty methods was geared towards the kinds of constraints that arise in image-based modeling and the implementation discussed in the next chapter.

Chapter 4

Image-Streaming

Chapters 2 and 3 both ended with a description of function fitting. In Chapter 2 the purpose was to show the link between image-based modeling and function fitting. In Chapter 3 the purpose was to provide background on nonlinear optimization and relate it to function fitting. This chapter brings these ideas together in an image-streaming context.

The streaming model of computation was designed for applications where data can be classified into two sets: the data that are local, small and reused often, and the data that are large, and for all practical purposes, never reused. This later data form the *stream*. I will discuss this data in terms of *local* and *streaming* bandwidth and memory requirements. In the streaming model, the set of operations performed on each element of the stream is called a *kernel*. The output stream from one kernel can be used as the input stream to another kernel.

This model for computation is not new. In 1961, one copy of the IBM Stretch (IBM 7030) supercomputer was equipped with a streaming coprocessor called the IBM Harvest (IBM 7950) (Buchholz, 1962). It was to set up to apply a short set of instructions to each element of a long string of characters. Since then, the streaming model has been the driving factor behind other architectures: video processors, DSPs and GPUs are contemporary classes of architectures that are designed to work on streams of data. Khailany et al. designed *Imagine*, a more general architecture designed to work on streams of data (Khailany et al., 2001). They have shown that this single architecture performs well on a number of different media processing tasks of a streaming nature. Their work includes empirical and theoretical evidence of how a stream processing architecture performs much better than general purpose processor (CPU) on data-streaming applications.

In the case of GPUs, the input stream is a set of triangles, and the output stream is a set of pixels. Multiple triangles and multiple pixels are processed in parallel. Purcell et al. (Purcell et al., 2002; Purcell et al., 2003) treated graphics hardware as a stream processor in terms of the inputs and outputs to the fragment processing and raster operation unit of the GPU. In their case, kernels were implemented using fragment programs and raster operations. The output stream from one kernel, which would be written into the framebuffer, would become the input stream for another kernel by rebinding the contents of the framebuffer as a texture in a subsequent pass. They implemented global illumination renderers using graphics hardware.

In addition to the multi-pass streaming paradigm of Purcell et al., I use a higher abstraction of streaming. The input stream in this dissertation is a set of images, and the output is an appearance model described by a set of textures. There are two levels of local memory in this model: the internal registers of the GPU, and the attached external memory of the GPU. The images form the stream data and are typically read from disk. This is why I seek to minimize stream bandwidth requirements.

4.1 Image Streaming

The choice to pursue an image-streaming approach is partly built on a few observations of the data and model sizes involved, which are listed next.

1. The image-based model is designed to have a memory footprint that fits easily into local memory.

2. The stream data, i.e. images, have a large memory footprint that will not typically fit into local memory.

3. The model for an object is broken down into independent patches on the surface of the model. Given that the model of the complete object is designed to easily fit into local memory, a single patch is quite small in total size.

Using these assumptions, I will show where I would expect an image-streaming approach to be more efficient. This section begins by outlining both the conventional approach and the image-streaming approach, and wrapping up this section with an analysis of stream bandwidth for each method, which is assumed to be the most precious commodity.

I have limited my analysis with respect to some further possible optimizations, particularly those that are system and model dependent, for the sake of clarity.

4.1.1 Conventional Approach

This section gives an overview of the conventional approach for generating image-based models (Algorithm 4). The process is usually broken down into two stages. In the first stage, data points are extracted from the images. This is what I have called the *distribution* step, which is described more fully in Chapter 5.2. The second stage is typically handled as an abstract computational stage where the image-based model is actually generated, such as through nonlinear optimization (Chapter 3).

In the first stage, the data are expanded to include the input vector, $\mathbf{x}_i$, which includes information such as view direction and surface parameterization, along with each data sample, y_i, which is radiance or pixel color. Keep in mind that y_i is not necessarily an original image pixel, as it is typically resampled according to the surface sampling of the model (Section 5.2).

Note that in the first stage, each image is processed once for each patch. For example, if there were a thousand patches, each image would be touched one thousand times. This represents a substantial amount of stream bandwidth. To reduce the amount

Algorithm 4 Conventional Approach to Building Image-Based Models

```
{Distribution}
for all patches do
  for all images do
    for all surface elements in patch do
      extract data pair $(\mathbf{x}_i, y_i)$
    end for
  end for
end for
{Optimization}
for all patches do
  for all iterations do
    for all data pairs, enumerated by $i$ do
      accumulate error and gradient
    end for
  end for
end for
```

of image reads, some trade-off is often made by processing batches of patches. The extreme version of batching is to process all the patches at once, which closely resembles the image-streaming approach described in the next section. However, maintaining the assumption that the image data are too large to fit into local memory, it's clear that not all of the patches can be processed at once. Analysis of blending this approach with the image-streaming approach would be the subject of future work (Chapter 9).

The second stage, as already mentioned, can be considered an abstract solution process. There is no notion of images or rendering at this stage. The working set of the model consists only of the one patch, which is typically quite small in relation to the complete model (Assumption 3 at the start of Section 4.1). This is another important difference between the conventional approach and the image-streaming approach proposed in this dissertation. One salient feature of the conventional approach is that during the second stage only the data for a single patch needs to be used at a time, which *might* mean that all the data could be kept in local memory after the data has been read from disk a single time.

The end result is that in the first stage the images will be read a number of times equal to the number of patches. Then, in the second stage, the *resampled* image data will be read again, but this time each resampled image pixel will be augmented by the vector $\mathbf{x}$. If this augmented data set is larger than fits into local memory, then the number of reads from disk (stream memory) will be multiplied again by the number of iterations. This is summarized in Table 4.1.

4.1.2 Image Streaming Approach

The image-streaming framework proposed in this dissertation combines the two stages of the conventional approach, and the loop over patches is moved from the outside loop to the inside loop (Algorithm 5).

Algorithm 5 Image-Streaming Approach to Building Image-Based Models

```
for all iterations do
  for all images do
    for all patches do
      for all surface elements in patch do
        extract data pair $(\mathbf{x}_i, y_i)$ {Distribution}
        accumulate error and gradient {Optimization}
      end for
    end for
  end for
end for
```

The data size is presumed to be too large to be considered part of the active working set. Instead, the entire model is used as part of the active working set. The images will be read a number of times corresponding to the number of iterations. Table 4.1 provides a summary of streaming bandwidth requirements for both the conventional and image-streaming approaches. The image-streaming approach will result in lower streaming bandwidth requirements when the number of iterations is less than the number of patches. Considering that models of interest typically have 10,000 or more patches, and require less than 100 iterations, this certainly seems a reasonable approach.

Approach	Image Reads	Resampled Data Reads
Conventional	n	1 or n_I*
Image Streaming	n_I	0

Table 4.1: *Stream bandwidth scaling in terms of the number of patches (n) and the number of iterations (n_I). * The second value represents the case when not all of the resampled and augmented data for a single patch can be stored in fast memory.*

A drawback of the image-streaming approach is that all patches are run for the same number of iterations, and the extraction of the model inputs and outputs is repeated for every iteration, even though they are the same in every iteration. Therefore, streaming bandwidth is saved at the cost of higher computational cost. However, the distribution step is essentially a rendering computation, and constitutes a small fraction of the processing time at each iteration when performing the computation on graphics hardware. Furthermore, the additional computational cost is hidden by the pipelined nature of the computation, which works well for the long streams of data that are processed in this framework.

4.2 Streaming Nonlinear Optimization

Migdalas et al. compiled a survey of parallelization techniques in nonlinear optimization (Migdalas et al., 2003). They begin by repeating the three avenues of parallelization that were suggested by Schnabel (Schnabel, 1995), and are repeated again here:

1. parallelization of the function and/or derivative evaluations in the algorithm;
2. parallelization of the linear algebra kernels;
3. modifications of the basic algorithms which increase the degree of intrinsic parallelism, for instance, by performing multiple function and/or derivative evaluations.

In the specific case of nonlinear least squares, there is a particular opportunity for parallelism that is important to this work: computation of the terms in the error function.

This falls under the first item in Schnabel's list. Each term requires the evaluation of the model function, a differencing of the model results with the target output data, and the squaring of this difference. Each term can be computed independently, and therefore represents a straightforward opportunity for parallelism. This is also true for evaluation of gradients.

With patch-based models, there is an additional opportunity for parallelism: each patch is independent of the others. This could loosely be described as falling under the third item in Schnabel's list. The image-streaming framework described here takes advantage of this parallelism as well, as all patches could be processed in parallel.

This work includes implementation of both the conjugate gradient algorithm and the steepest descent algorithm. This framework also applies to limited-memory BFGS, although it has not been implemented. The computationally intensive operations are computing the error function and the gradient of the error function. These functions can be written as a sum over all samples. As an example, the error function can be written as:

$$E(\mathbf{p}) = \sum_{i=1}^{S} r_i(\mathbf{p})^2, \tag{4.1}$$

where S is the number of radiance samples and r_i is the residual in radiance sample i.

The model parameters are typically stored in texture maps for rendering. Since the model is intended for realtime rendering, the number of texture reads must be limited to a small number, which means that even in models with a large number of parameters, only a small number are required to synthesize each pixel. This means that the first and the second derivatives of each residual with respect to the model parameters $\mathbf{p}$ will have a small number of nonzero entries. This becomes an important issue in discussing the gradient computation in Section 4.3.2.

The observation that the error function and its gradient can be expressed as a sum over data samples leads naturally to the formulation of optimization as a stream process. In this formulation, the data samples are continuously streamed through the processor, while the fragment unit updates the optimization information based on the contribution of each sample. The model parameters are updated only after all the data samples are processed. Algorithm 6 outlines the image-streaming implementation of nonlinear optimization for image-based modeling.

Algorithm 6 Image-Streaming Algorithm for Building Image-Based Models using Nonlinear Optimization

```
 1: Compute initial guess for the model parameters $\mathbf{p}_0$
 2: $k \leftarrow 0$
 3: repeat
 4:    $E(\mathbf{p}_k) \leftarrow 0$ {$E$ is the error function}
 5:    $\nabla E(\mathbf{p}_k) \leftarrow 0$ {$\nabla E$ is the gradient of error}
 6:    for $i = 1$ to $S$, the number of image data samples do
 7:       {Here we increment error and gradient according to the term from sample $i$}
 8:       $E(\mathbf{p}_k) + = E_i(\mathbf{p}_k)$ (Section 4.3.1)
 9:       $\nabla E(\mathbf{p}_k) + = \nabla E_i(\mathbf{p}_k)$ (Section 4.3.2)
10:    end for
11:    {$f$ is the objective function, $P_m$ are the penalty functions}
12:    $f(\mathbf{p}_k) = E(\mathbf{p}_k) + \sum_m P_m(\mathbf{p}_k)$ (Section 4.4)
13:    $\nabla f(\mathbf{p}_k) = \nabla E(\mathbf{p}_k) + \sum_m \nabla P_m(\mathbf{p}_k)$ (Section 4.4)
14:    if $k = 1$ or the Wolf Conditions are met (Section 4.5.2) then
15:       $k \leftarrow k + 1$
16:    end if
17:    Compute search direction $d_k$ (Section 4.5.1)
18:    Compute stepsize $\alpha_k$
19:    $\mathbf{p}_{k+1} \leftarrow \mathbf{p}_k + \alpha_k \mathbf{d}_k$
20: until User is satisfied
```

The error function is not necessarily the complete objective function. This approach allows for a set of penalty terms that together with the error function, constitutes the complete objective function. This is reflected in Lines 12 and 13 of Algorithm 6. Penalty terms are discussed in Section 4.4.

4.3 Incremental Computation

This section focuses on the quantities that are computed incrementally while looping over images in an image-streaming framework. There are two quantities that are always built incrementally: the error function, and the gradient. It is also possible to use the Hessian to compute a second order model for the stepsize estimate (Equation 3.13) in an incremental fashion.

4.3.1 Evaluating the Error Function

The objective function is the sum of the error function and any applicable penalty terms. The error function is itself a summation:

$$E(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{S} r_i(\mathbf{p})^2. \tag{4.2}$$

This is the most straightforward incremental computation when streaming images. The error is a per-patch quantity, meaning that in a case such as LFM there is a scalar value for error for each vertex of the model, and for the SBRDF model there is a scalar for each texel. As each image is processed, the error terms from the image are accumulated into the appropriate error space.

4.3.2 Evaluating the Gradient

The gradient of the error function is a vector for each patch. Using the chain rule, the gradient of the error term can be written as

$$\nabla E(\mathbf{p}) = \mathbf{J}^T\mathbf{R} = \sum_{i=1}^{S} r_i(\mathbf{p})\nabla r_i(\mathbf{p}). \tag{4.3}$$

J is the Jacobian matrix, and **R** is the vector of residuals. As you can see, the gradient can also be computed in an incremental fashion.

One nice characteristic of the image-based models intended for rendering is that there can only be a few parameters used for a single function evaluation (Section 4.2). This translates into a small upper bound on the number of non-zero entries in each row of the Jacobian. For example, a single term of LFM rendered with nearest-filtering has no more than two non-zeros in each row of the Jacobian.

4.3.3 Computing the Stepsize using the Hessian

Section 3.4.2 gave the analytic expression for the stepsize. This is the only stepsize method discussed in this work that requires per-image incremental computation. The expression for the Hessian-based stepsize is

$$\alpha_k = \frac{-\nabla f(\mathbf{p})^T \mathbf{d}}{\mathbf{d}^T \mathbf{H} f(\mathbf{p}) \mathbf{d}}. \tag{4.4}$$

Recall that the objective function is the sum of the error function and any applicable penalty terms. Separating these terms, the Hessian is written as

$$\mathbf{H} f(\mathbf{p}) = \mathbf{H} E(\mathbf{p}) + \mathbf{H} \sum_m P_m(\mathbf{p}_k), \tag{4.5}$$

where P_m is the m^{th} penalty function. The error term in the Hessian can be computed incrementally by using the fact that it is a summation.

$$\mathbf{H} E(\mathbf{p}) = \mathbf{H} \sum_{i=1}^{S} r_i(\mathbf{p})^2 = \sum_{i=1}^{S} \mathbf{H} r_i(\mathbf{p})^2. \tag{4.6}$$

Of course, it is not practical to store the complete Hessian, as it is $O(n^2)$, but since only the product of the Hessian with the direction vector is required, the denominator can be built incrementally using

$$\mathbf{d}^T \mathbf{H}E(\mathbf{p})\mathbf{d} = \sum_{i=1}^{S} \mathbf{d}^T \mathbf{H}r_i(\mathbf{p})^2 \mathbf{d}. \tag{4.7}$$

The catch is that $\mathbf{d}$ must be computed before building $\mathbf{d}^T \mathbf{H}E\mathbf{d}$ in this fashion. Therefore, all images are processed a second time for each iteration in order to complete this stepsize calculation. In other words, line 18 in Algorithm 6 would include another loop over all images to compute the stepsize in this case.

4.4 Penalty Terms

Regularization and bounds constraints are handled via the penalty method as described in Section 3.6. The penalty method adds an additional term to the objective function. The difference in calculating penalty terms as opposed to the error terms is that the penalty terms depend only on the parameters, not the image data. This is why they are added outside of the loop over images in Algorithm 6. As this is essentially in an outer loop, it is not as critical for efficient implementation. However, implementing this outer loop on the graphics card avoids the need for moving data across the bus between the host and the graphics card.

4.4.1 Bounds Constraints

Bounds constraints were discussed in Section 3.6.1. The quadratic penalty function for bounds constraints is:

$$P(\mathbf{p}) = \sum_i (\min(l_i - p_i, 0))^2 + (\max(u_i - p_i, 0))^2 \tag{4.8}$$

Each term corresponds to each parameter of the model. The sum is simply a sum reduction.

The i^{th} component of the gradient vector is

$$\nabla P(\mathbf{p})_i = 2\min(l_i - p_i, 0)\max(u_i - p_i, 0). \tag{4.9}$$

The Hessian of the bounds constraints penalty function is a diagonal matrix. The penalty term in the denominator for the stepsize in the conjugate gradient algorithm is

$$\mathbf{d}^T\,\mathbf{H}P(\mathbf{p})\,\mathbf{d} = \sum_{i=1}^{n} d_i^2\,\Theta, \tag{4.10}$$

where d_i is the i^{th} element of the direction vector $\mathbf{d}$ and

$$\Theta = \begin{cases} -2, & \text{if } p_i < l_i; \\ 2, & \text{if } p_i > u_i; \\ 0, & \text{otherwise.} \end{cases} \tag{4.11}$$

4.4.2 Regularization

Regularization was discussed in Section 3.6.2. The penalty function is:

$$P(\mathbf{p}) = (\mathbf{Lp})^T(\mathbf{Lp}). \tag{4.12}$$

The matrix $\mathbf{L}$ is not stored explicitly, as it can be constructed in a procedural manner on-the-fly. As an example, consider a template for the two-dimensional case:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Regularization is typically not enforced across different model subspaces. If the parameters are already laid out in their two-dimensional space, evaluating this template for each model parameter p_i simply involves reading the neighboring parameters, multiplying by their corresponding weight, and summing the results to get the total for element i in $\mathbf{Lp}$. The penalty function is evaluated by squaring each term, followed by summation.

The above template is not appropriate for parameters at the edge of the subspace. For example, if a parameter is at the top of the subspace, it's template would be

$$\begin{bmatrix} -1 & 3 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

where the 3 lies over the parameter in question. The view map in Figure 5.6 is a good example to illustrate where the border case comes into play. Notice that there is a circle of model parameters, and anything outside of this circle is empty texture, and should not factor into the regularization term. The validity of a neighboring texel can be tested by either a check of the coordinates against the circle center and radius, or by marking invalid texels in some way, e.g. putting a mask in the alpha channel. Algorithm 7 shows how the template is built procedurally.

The gradient of the regularization penalty term is

$$\nabla P(\mathbf{p}) = 2\mathbf{L}^T\mathbf{L}\mathbf{p}. \tag{4.13}$$

This is implemented by applying the regularization operator to the parameter set twice and multiplying by two. In other words, running Algorithm 7 twice, but with a multiplication by two instead of the square in the last step, and without the summation.

Again, the Hessian is only used to compute a term in the denominator for the stepsize

Algorithm 7 Procedural evaluation of the two-dimensional regularization penalty term

```
$(i, j)$ = coordinates of the output term.
$T$ = texture map of model parameters
$w \leftarrow 0$ {This will be the center weight}
$f \leftarrow 0$ {This will be the output}
if $(i+1, j)$ is valid then
   $f \leftarrow f + (-1)T(i+1, j)$
   $w \leftarrow w + 1$
end if
if $(i-1, j)$ is valid then
   $f \leftarrow f + (-1)T(i-1, j)$
   $w \leftarrow w + 1$
end if
⋮
$f \leftarrow f + wT(i, j)$
$f \leftarrow f^2$ {This takes care of the squaring in Equation 4.12, leaving only the summation
to finish}
```

using a one-sided quadratic approximation. The expression for this term is

$$\mathbf{d}^T \mathbf{H} P(\mathbf{p}) \mathbf{d} = 2\, \mathbf{d}^T \mathbf{L}^T \mathbf{L} \mathbf{d}, \tag{4.14}$$

which is the same as the function for $P(\mathbf{p})$, including the square and summation, but the direction vector $\mathbf{d}$ that appears in Equation 4.14 takes the place of the parameter vector $\mathbf{p}$ that appears in equation 4.12, and there is an additional factor of two.

4.5 Computation of the Next Guess

Once all the images have been processed, we will have the complete gradient and error. This information is used to compute the next guess in the nonlinear optimization process. Again, since this is essentially an outer loop, it is not as critical for efficient implementation.

Recall that a step $\mathbf{p}_{k+1}$ is tried to see if it meets the criteria for acceptance, e.g. the Wolfe conditions (Equations 3.11 and 3.12). If it does not, then the stepsize will be modified to find another candidate $\mathbf{p}_{k+1}$. In the conventional approach (Algorithm 8), the error for a step is computed first, and the gradient is only computed if the step is accepted, thus saving a gradient evaluation when the step is not accepted.

Algorithm 8 Conventional Line Search: Deferred Gradient Computation

```
repeat
    Choose a new $\mathbf{p}_{k+1}$
    Compute $E(\mathbf{p}_{k+1})$
until $E(\mathbf{p}_{k+1})$ meets the first Wolfe condition (Equation 3.11)
Compute $\nabla E(\mathbf{p}_{k+1})$
Accept $\mathbf{p}_{k+1}$ as the new iterate, i.e. $k \leftarrow k+1$
```

In an image-streaming approach, any data that must be gleaned from an image should be collected while the image is available. Therefore, the gradient for a step is computed regardless of whether the step is accepted (Algorithm 9). As a consequence, the second Wolfe condition (Equation 3.12) can be evaluated with little additional cost.

Algorithm 9 Line Search With Concurrent Error and Gradient Computation

```
repeat
    Choose a new $\mathbf{p}_{k+1}$
    Compute $E(\mathbf{p}_{k+1})$
    Compute $\nabla E(\mathbf{p}_{k+1})$
until $E(\mathbf{p}_{k+1})$ meets the Wolfe conditions (Equations 3.11 and 3.12)
Accept $\mathbf{p}_{k+1}$ as the new iterate, i.e. $k \leftarrow k+1$
```

4.5.1 Direction

The direction for steepest descent is straightforward, as it is the negative gradient. The conjugate gradient direction is slightly more sophisticated. It is a weighted average of the negative gradient and the previous search direction (Equation 3.3). The weighting is determined by the dot products of gradients from two iterations (Equation 3.5). This requires additional storage for the extra gradient and direction history. The computation itself is relatively straightforward to implement using the sum reduction operations outlined in Section 5.1.

The L-BFGS computation for direction is quite a bit more complex than either of the previous methods. Like conjugate gradient, it requires storage of information from previous iterations. In the case of conjugate gradient, only information from the previous iteration is required. In the case of L-BFGS, information from some constant number of previous iterations, $m > 1$ is required. This is fine, as long as memory allows. Also, the computation of the direction requires two loops over these previous iterations (Algorithm 1 in Chapter 3). This results in a substantial increase in cost for computing a direction relative to conjugate gradient and steepest descent. This will, however, be small relative to the computation of the error and gradient itself.

The direction can be tested to insure that it is a descent direction, which is particularly important in the conjugate gradient method where the computed direction is not guaranteed to be a descent direction. The test is:

$$\frac{\nabla f(\mathbf{p})^T \mathbf{d}}{\|\nabla f(\mathbf{p})\| \|\mathbf{d}\|} < -\gamma \tag{4.15}$$

where γ is set to some small positive number. If this test does not pass, the direction is set to the negative gradient. This is easily expressed as a conditional assignment (Algorithm 10), which is amenable to implementation on current graphics hardware.

Algorithm 10 Implementation of the direction computation

```
Compute candidate direction $\mathbf{d}'_k$
$\mathbf{d}_k \leftarrow$ if $\frac{\nabla f(\mathbf{p}_k)^T \mathbf{d}_k}{\|\nabla f(\mathbf{p}_k)\| \|\mathbf{d}_k\|} < -\varepsilon$ assign $\mathbf{d}'_k$ else assign $-\nabla f(\mathbf{p}_k)$
```

4.5.2 Line Search

In Algorithm 6, step 12 is expressed as an increment of the variable k. In practice this is implemented by conditional assignment, not unlike the direction test (Algorithm 10). The condition is that the step is “good enough”, e.g. satisfies the Wolfe conditions. There are a number of quantities that are conditionally assigned. Algorithm 11 lists a few of those involved in the steepest descent method as an example. Although all patches are

Algorithm 11 Implementation of the Line Search ($k \leftarrow k+1$).

```
Boolean $b \leftarrow$ result of the Wolfe conditions (Equations 3.11 and 3.12)
$\mathbf{p}_k \leftarrow$ if $b$ assign $\mathbf{p}_{k+1}$ else assign $\mathbf{p}_k$
$f(\mathbf{p}_k) \leftarrow$ if $b$ assign $f(\mathbf{p}_{k+1})$ else assign $f(\mathbf{p}_k)$
$\nabla f(\mathbf{p}_k) \leftarrow$ if $b$ assign $\nabla f(\mathbf{p}_{k+1})$ else assign $\nabla f(\mathbf{p}_k)$
```

evaluated in parallel, each patch will have a different result when evaluating the Wolfe condition. This means they do not progress at the same speed in terms of number of *accepted* steps.

4.6 Summary

This completes the description of the image-streaming framework. The next chapter discusses the resampling issue that arises due to the fact that the original data (image pixels) are sampled at regular intervals in image space, but the image-based model is expressed in terms of regular intervals in texture space. The other major topic of the next chapter is detail related to implementation on programmable graphics hardware. Further details on specific case studies are given for LFM (Chapter 7) and SBRDF (Chapter 8).

Chapter 5

Texture Space Transformations

Chapter 4 laid out the operations required to construct an image-based model in an image-streaming context. The details of how to perform some fundamental math operations were not discussed. That is the subject of this chapter, which also focuses on an implementation for graphics hardware.

Most GPGPU work concentrates on operations on data in a single, abstract space: vectors, matrices, or perhaps sparse matrices. By contrast, I need to take into account various spaces related to the geometry of the model, such as patches, triangles, vertices, surface parameterization, and view direction. These spaces must be expressed in terms of textures. For example, error is a per-patch value, and therefore requires a texture laid out such that each texel corresponds to a patch.

This chapter outlines the basic transformations that must occur in the image-based modeling process. The first section describes the basic limitations of graphics hardware, motivating the choices of how transformations are implemented. Some common transformations are described next. Particular attention is given to the first transformation: moving from image space to the space of surface parameterization, which I call *surface location space* or SLS, following the nomenclature adopted in our SIGGRAPH paper (Hillesland et al., 2003). This is actually the only transformation required for a model like SBRDF, where each texel in SLS represents a complete patch. The remaining transformations are necessary for models with larger patch sizes, such as LFM. They form part of the mathematical operations that are necessary for the nonlinear optimization process, such as sum reductions to gather information across an entire patch of the model.

In constructing a reflectance model, there is an additional operation to convert radiance to reflectance using lighting information. This is discussed in Section 5.7. Many image-based models treat view-independent effects separately, and so Section 5.8 is a brief discussion of the issues relating to the meaning and treatment of view-independent effects.

5.1 Basic Limitations of Graphics Hardware

At the start of the traditional hardware graphics pipeline the vertices are transformed into screen space. In the rasterization stage, per-vertex quantities are interpolated to generate per-fragment values. The fragment engine takes inputs from rasterization, performs lookups into textures, and outputs resulting colors and depths. Raster operations and z-buffering are performed on fragments to generate the final pixels. The first key restriction is that the output address is determined in the vertex processing and rasterization stages. The second restriction is that the number of outputs from the fragment stage is fixed. This means that operations implemented in the fragment stage cannot use a *scatter*, which is an operation describing the distribution of information from one location (the fragment processor) to a number of output locations (pixels). Instead, fragment programs need to be described in terms of gathers and reductions, which are described next.

A *gather* refers to the operation of gathering data from a number of locations to a single place. A combination of these data to a smaller dimensionality is referred to as a *reduction*. An example is a *sum reduction*, where elements are combined by summation. The terms scatter, gather, reduction and sum reduction are from the parallel computing community. These are operations included in the Message Passing Interface (MPI), for example, which is a popular API for parallel computing (Message Passing Interface Forum, 1995). The implementation of a reduction for general computation on graphics

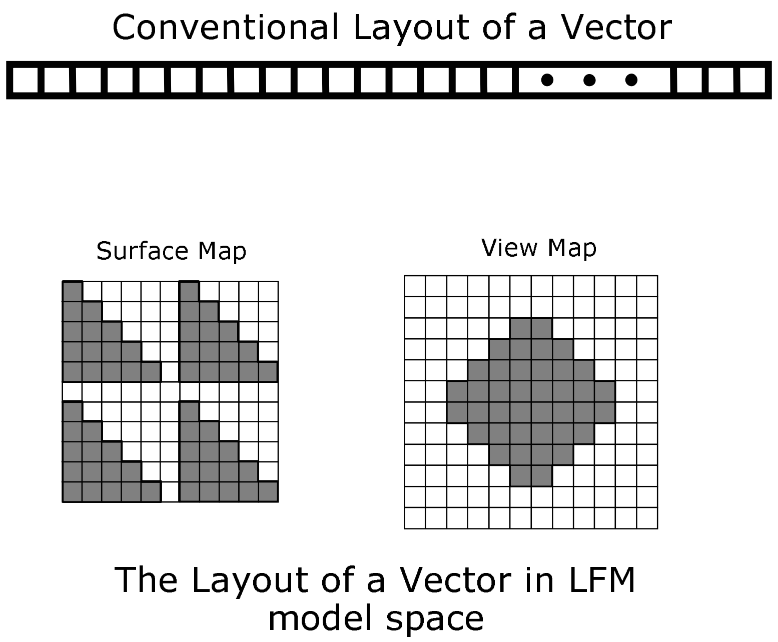

Conventional Layout of a Vector
Surface Map
View Map
The Layout of a Vector in LFM model space

Figure 5.1: *This shows the difference between the conventional notion of how a vector is laid out, and how the vector of model parameters,* **p**, *is laid out in LFM model space for a single patch. Each texel represents a parameter, and texture coordinates are used to address the parameters in this space. For example, a model input of view direction is a coordinate in view map space that returns a view map parameter.*

hardware is well understood. The typical approach is described in an article on “A Toolkit for Computation on GPUs” in the *GPU Gems* collection (Buck and Purcell, 2004).

Now that we are talking about graphics hardware implementation, we can relate model constructs to their representations in this implementation. Model spaces can be thought of as texture coordinates, and model parameters as texels.

Some models have parameters arranged in a number of different subspaces. The LFM model has parameters organized according to surface location in the surface map, and according to view direction in the view map as illustrated in Figure 5.1. A vector in the complete model space is a composite of vectors in each of these subspaces; therefore, reductions across the model parameter space requires reductions in each of it’s subspaces, and then a final gather across spaces. An example is a dot product:

$$\sum_{S \in modelsubspaces} \text{sum-reduce}(\mathbf{p}_S . \mathbf{p}_S) \tag{5.1}$$

The lowered dot in the above expression indicates a component-wise multiplication. The vector $\mathbf{p}_S$ represents the components of the model parameters $\mathbf{p}$ that are in the subspace S. This gather is part of the evaluation of the bounds constraints penalty function, for example (Equation 4.8).

Returning to the LFM example, the sum reduction must occur across the surface patch (triangle ring) as well as the viewmap. The surface map is implemented in two stages: first a sum reduction that collects data in each triangle to a single scalar per-triangle. Then, a sum reduction that collects the scalars from each triangle into a single scalar for the triangle ring.

The next few sections give a more complete view of transforming between spaces with more of a focus on implementation issues with respect to graphics hardware. This constitutes one of the basic toolsets for the framework in this dissertation.

5.2 The Distribution Step: From Image Space to Surface Location Space

The images are a parameterization of the radiance data according to their position in image space. With each image, there is associated information that describes the camera settings, such as the field of view, the resolution, and the transformation from the world reference frame to the camera reference frame. In order to generate the image-based model, these data must be reparameterized according to the image-based model's coordinate system. For example, the view direction in a local coordinate system is typically needed for each image pixel. In other words, it is necessary to translate what happens in the image space of the original reference images back to the space of the model. This section concentrates on the *distribution* step, which refers to the process of sorting image pixels according to their coordinates in some model subspace. This is conventionally handled as a pre-process, but with image-streaming it is performed on-the-fly.

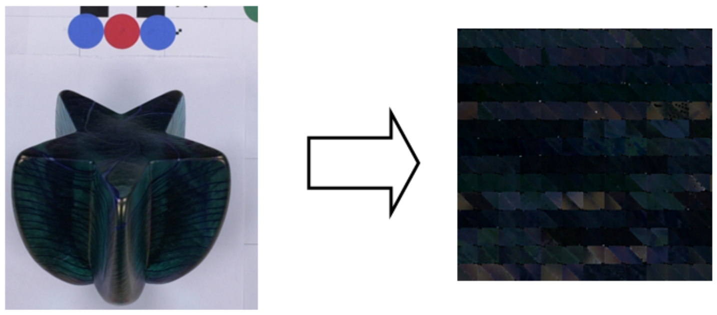


Figure 5.2: *Before any model fitting can occur, the image data must be transformed into model space. This includes operations such as transformation, rasterization, filtering, and visibility determination. The image on the left is resampled into surface location space in the figure on the right.*

A key assumption of this work is that the object has some sort of surface parameterization. Any transformation from image space to model space begins with a transformation from image space to surface location space (SLS). This task is not unique to my framework; however, it is a process that is not usually covered in much detail elsewhere. I have chosen to give more detail here because it is a critical phase for the framework's effectiveness.

The first subsection describes the most straightforward technique for making this transformation, but it requires a scatter operation. The second subsection shows how to accomplish this transformation by using gathers instead of scatters.

5.2.1 Mimicking the Rendering Process

In rendering, the inputs to the shading model are quantities such as view direction and light direction in a local coordinate system. The easiest approach to recovering the various model inputs for each image pixel is to simply mimic the rendering process and write the model inputs to the framebuffer. Figure 5.3 shows an example where the model inputs form a set of images corresponding to the original photograph. Typically, more than one image is required for each photograph because the number of model inputs is often greater than the number of framebuffer channels. This corresponds to Step 4 in Algorithm 12.

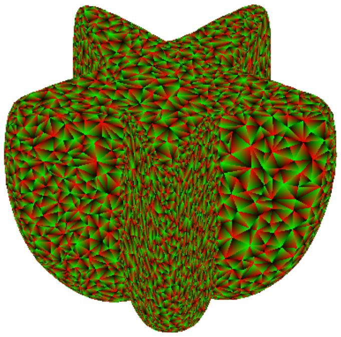
The complication arises in Step 9. The information computed for each photograph pixel must be distributed according to texel addresses used in the filtering process. This is a scatter operation which, as discussed in Section 5.1, is not currently available in the programmable fragment stage of graphics hardware. To implement this operation on current graphics hardware would require writing the texel address into the framebuffer, and then using this information as vertex coordinates in a second pass. Instead, we can organize this computation in terms of a gather by replacing the loop over image pixels by a loop over model texels. This is the subject of the next section.

5.2.2 From Image-Space to SLS Using Gather

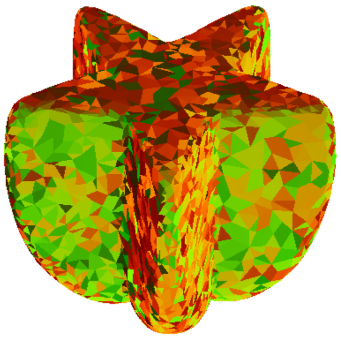
Algorithm 13 is an outline of how the distribution step may be implemented in terms of gathers rather than scatters. It's helpful to compare Algorithm 12 to Algorithm 13. One difference is that the inner-most loop is over pixels in the Algorithm 12, and SLS texels in Algorithm 13. This inner loop defines the space of rasterization, thus defining the sampling of the model function and the organization of the output. The second difference is where visibility is determined. The results of visibility determination are actually used in the construction of the filter weights, not to determine which fragments are present in the final output.

The filter weight choice is different than for the rendering case. Instead of choosing weights to apply to texels in order to generate an image pixel, we must choose weights to apply to image pixels in order to generate a texel. One option is to choose a filter weight proportional to distance in image space. This is the filtering function $\mathbf{F}_I$ in Al-

(a) *Reference image*



(b) *Surface map parametrization*



(c) *View direction parametrization*

Figure 5.3: *It is necessary to find the inputs that correspond to each reference image pixel. The most straight forward technique is to modify the renderer to write out the inputs as is shown here for some LFM inputs of the star model.*

Algorithm 12 From Image-Space to SLS Using Scatter

```
1: for all images do
2:   for all triangles do
3:     for all Image pixels in the triangle called $P_I$ do
4:       $\mathbf{P} \leftarrow$ Compute texture coordinates for $P_I$.
5:       $(\mathbf{T}, \mathbf{w}) \leftarrow \mathbf{F}(\mathbf{P})$
6:       $y_i =$ pixel in the photograph.
7:       if $P_I$ passes the Z-test then
8:         for $j = 1 \ldots n$, where $n$ is the number of elements in $\mathbf{T}$. do
9:           Send element $w_j$ and $y_i$ to address $T_j$ for further processing
10:        end for
11:      end if
12:    end for
13:  end for
14: end for
```

Line 4 In the case of SBRDF, this would only be a single texture coordinate. In general, this is a vector because there will be more than one texture coordinate for an image-based model. For LFM, there would be a texture coordinate in the surface map and view map for each of the triangle's three vertices.

Line 5 $\mathbf{F}$ represents the texture filtering function. It returns both the addresses of texels that are used in the filter ($\mathbf{T}$) and the weights of the filter ($\mathbf{w}$).

Line 9 This is a scatter operation, which is not possible on current graphics hardware.

Algorithm 13 From Image-Space to SLS Using Gather

```
1: for all images do
2:   $V_I \leftarrow$ rasterize and z-buffer from camera view, storing visibility information.
3:   for all triangles do
4:     for all SLS texels with address $T$ and 3D location $T_{xyz}$ do
5:       $T_I$ = Projection of $T_{xyz}$ to image-space.
6:       $(\mathbf{A}, \mathbf{w}) \leftarrow \mathbf{F}_I(T_I, V_I)$ or $\mathbf{F}_T(T, V_I)$
7:       $\mathbf{c} \leftarrow$ color values in image pixels addressed by $\mathbf{A}$.
8:       $y_i \leftarrow \mathbf{c} \cdot \mathbf{w}$
9:       Store $y_i$ for texel $T$
10:    end for
11:  end for
12: end for
```

Line 5 $\mathbf{F}_I$ is a texture filtering function operating in image space. $\mathbf{F}_T$ is a texture filtering function in texture space.

gorithm 13, and is the approach taken in LFM work (Chen et al., 2002a; Chen, 2002), SBRDF work (McAllister, 2002), and by Marschner (Marschner, 1998). The difficulty in this kind of approach is that there is a nonlinear relationship between image and texture space. Points that are adjacent in image space can be arbitrarily distant in texture space. The most extreme case is where a polygon is viewed at a near-grazing angle. To compensate for this effect, Chen and McAllister simply throw out samples from polygons that are at near-grazing angles. Marschner weights the importance of the sample by a function of the dot product between the viewing direction and the normal. $\mathbf{F}_T$ will be discussed after taking a look at visibility.

One or more of the pixels surrounding P_I in image space could be from completely different parts of the object, or not part of the object at all. This is because visibility is not continuous. For pixels far from P_I in model space, the weights of the filter should be zero. When the models are simple enough, for example when analyzing a material sample (McAllister, 2002), where the geometry is simply a single polygon, this is fairly straightforward to determine.

One possible approach for more complex models is to determine visibility using a geometric technique, although this is rather computationally intensive. The approach used in the OpenLF software (Chen et al., 2002b), which is an implementation of LFM, is an analytic approach of this nature. Although it makes some approximating simplifications to speed this process, it is fundamentally an $O(n^2)$ approach that is not suitable for large models.

Another approach is to use ray-casting. This is a convenient approach because it does not rely on regular sampling in image-space. The ray is simply cast from the point on the surface to the appropriate point in the image plane. This is the approach that Marschner took (Marschner, 1998).

For rasterization, the visibility problem is a bit more complicated to handle, but with the use of graphics hardware can be quite fast. This technique is closely associated with shadow mapping (Williams, 1978), where one rasterizes from the point of view of the camera, while looking up into a shadow map that was generated from the point of view of the light source.

The first step is to render the model from the perspective of the camera in order to resolve visibility in image-space using conventional z-buffering. The visibility information is used when it is time to compute filter weights. Either the resulting depth map can be consulted in a manner similar to shadow mapping, or an item buffer can be used. Lensch et al. (Lensch et al., 2003) used the depth comparison approach in their work. They were evaluating the error and gradient in a spatially-varying Lafortune model, similar to the work in this dissertation.

In the conventional item buffer algorithm, a unique identifier (number) is assigned to each triangle. The identifier number is rasterized in image space, rather than color. In a later pass, the visibility of the triangle can be determined by looking for its identifier in the item buffer. Another way of looking at an item buffer, is that it is an image that specifies what triangle is visible at each pixel.

I used an approach resembling the item buffer approach. Instead of rasterizing item numbers, I rasterized texture coordinates (Cignoni et al., 1998; Apodaca et al., 1999; Carr and Hart, 2002). Consider a texture atlas for the entire model: a coordinate in that space can only belong to a single triangle. If the texture coordinate of each fragment is written to the framebuffer, then the resulting *texture coordinate buffer* specifies not only what triangle is visible at each pixel, but what specific point on that triangle is visible at each pixel in texture space. This additional information becomes useful in constructing filter weights for resampling. With access to the texture coordinates of each pixel, I was able to use a linear filter with a fixed size in texture space and avoid some of the problems associated with interpolation in image space. This is also consistent with texture mapping in conventional rendering, where the filter is also linear in texture space.

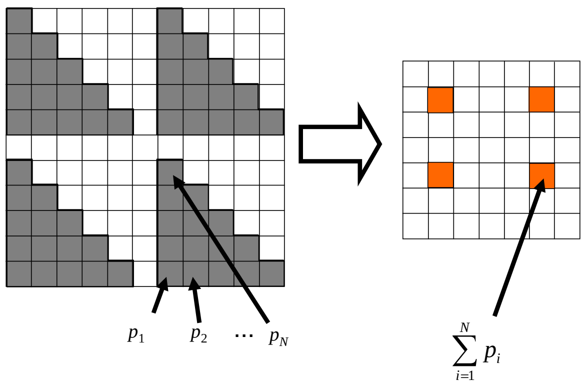


Figure 5.4: *Transforming from SLS to a space of a single scalar per model triangle. In particular, this figure shows a sum-reduction. The left image is a simplified view of the SLS texture, and is equivalent to the right image in Figure 5.2. The right image here is per-triangle space, which means that each texel corresponds to a triangle.*

5.3 Surface Location Space to Per-Triangle Space

Per-triangle space refers to a layout where each texel corresponds to a triangle in the geometric model. I'm using the term 'triangle' a bit loosely here, as any polygon could be used. Any model that has a patch defined in terms of polygons, such as a quadrilateral in the case of typical BTF models, or a triangle ring as is the case of LFM.

In moving from SLS to a per-triangle space we are gathering results across an entire triangle of the model. This is relatively straight forward, as it resembles the conventional sum reduction, but with the additional complication of gathering from a triangular region instead of the rectangular region that is more common in other GPGPU work. In my case, I laid surface location space out in a regular fashion to make this transformation relatively straight-forward. This can be seen in Figure 5.4. With this regular layout, the rendering primitives can either be the model triangles, which will be sub-pixel sized in the output space, or a screen-filling quad if masking or procedural address computation is used to read only from the triangle region in the input space.

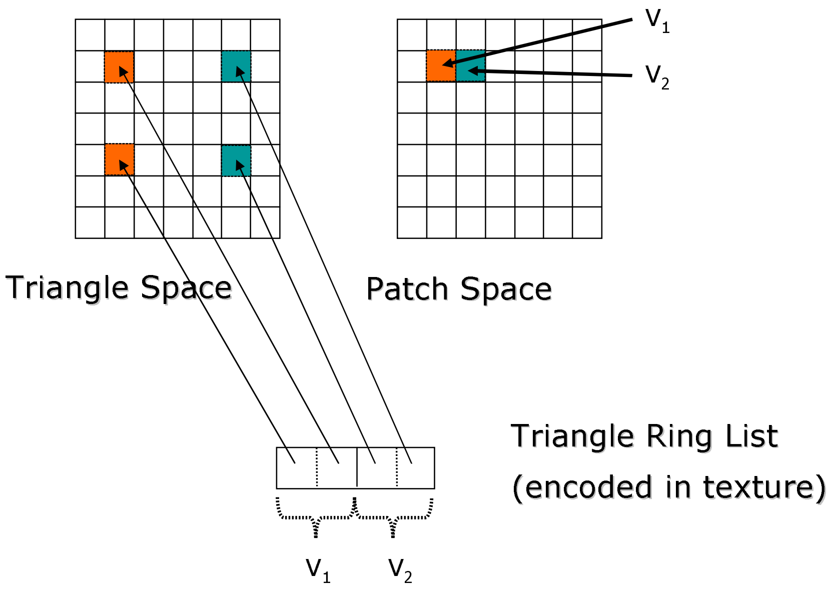


Figure 5.5: *Gathering from all triangles in a patch requires indirection. For an implementation on graphics hardware, an additional texture has a list of triangles for each patch. In this figure, v_1 is a patch consisting of two triangles, and so the triangle ring list has coordinates for two triangles, likewise for v_2. Each output pixel corresponds to a patch.*

5.4 Per-Triangle Space to Per-Patch Space

In some cases, a patch is made up of a number of geometric primitives. The LFM model, for example, consists of patches made up of triangles sharing a single vertex. In this case, it is necessary to gather information across the patch. Unlike the gathers discussed so far, this is typically handled by indirection.

For the example of the LFM model, indirection is accomplished by creating a texture that contains a list of pointers to triangles that share each vertex. The pointers are addresses into the per-triangle texture. This is an example of how mesh connectivity is encoded in texture maps. To gather information for a patch, a single screen-filling quadrilateral is rendered to the framebuffer. Each output fragment corresponds to a vertex of the model. There are two textures. The first is in per-triangle space (Section 5.3). The second is the indirection texture just described, which is organized by model vertex, but has a number of texels (pointers) devoted to each one. Each texel contains the address in triangle space of a triangle in the ring around the vertex. The fragment program executes for each patch, reading a texture to get the addresses of the triangles in the ring, and gathering the data from these triangles.

5.5 Per-Patch Space to View Space

Image-based models usually include a parameterization of the view direction in some local reference frame attached to each patch. View direction is two-dimensional, and therefore maps naturally to a two-dimensional texture. Chen et al. used a view map for each vertex (Chen et al., 2002a; Chen, 2002), using the Nusselt analog. In short, the Nusselt analog is a two dimensional cartesian coordinate parameterization of the direction hemisphere projected down onto the plane (Figure 5.6) (Nusselt, 1928; Cohen and Wallace, 1993). Chen et al. used the same square resolution for each view map, and had a coordinate system associated with each vertex. The view maps for all vertices were tiled into a single texture as illustrated in Figure 2.4b. I followed this same approach in my implementation. Therefore, the view space can be described as being laid out according to vertex at the top level, and according to a local Nusselt parameterization within the space of each vertex.

The eyepoint has a discrete position in space, and therefore corresponds to a single point in the view space of the local reference frame. Since there is one view direction for each local reference frame, a single point would be rendered into each view map. Note, however, that a view map is a discrete sampling of view directions, so we again have a resampling issue. In my implementation, the point was placed in the closest discrete view location.

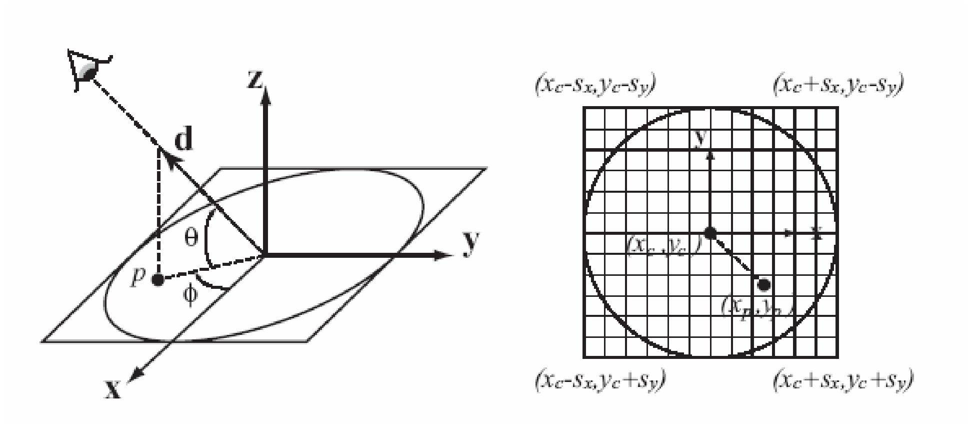


Figure 5.6: *The Nusselt analog is a mapping of view direction into a cartesian coordinate system. Each local coordinate system has an associated center position* (x_c, y_c) *in the view map texture. This image is from Chen's Dissertation (Chen, 2002)*

5.6 Composite Transformation Example: From SLS to View Space

A transformation from SLS to view space serves as an example of a composite transformation. First I describe what it means to go from SLS to view space. The goal is to move data stored according to surface parameterization to a position parameterized according to view direction. In this case, I'm really talking about a number of texels in a surface parameterization, and ending up with a scalar, so there must be a reduction of some sort. This is actually the process that is required to compute part of the gradient in LFM, and will be mentioned in that context in Chapter 7.

The view direction is typically parameterized in a local reference frame, as discussed in Section 5.5. In LFM, the local reference frame is attached to a vertex. This transformation can be accomplished by a sequence of transformations:

1. Transform data from SLS to per-triangle space.

2. Transform data from per-triangle space to per-vertex (per-patch) space.

3. Transform data from per-vertex space to per-view space.

There's an implied approximation in this example: all points in a triangle ring are presumed to have the same view direction. This approximation is manifested in the first two steps. The gather in the first step implies the approximation that all view directions within a triangle are the same. The gather in the second step implies the approximation that all triangles in a triangle ring share the same view direction.

5.7 From Radiance to Reflectance

Images capture radiance. When working with reflectance models, the radiance must be converted to reflectance. Reflectance models are usually captured under controlled lighting conditions, typically with a light approximating a single point-light source. Since I specifically looked at the SBRDF, this section proceeds with the lighting model used by McAllister (McAllister, 2002), Equation 2.13:

$$L_r(\omega_r) = f_r(\omega_i, \omega_r) L_c \cos(\theta_i)/r^2, \tag{5.2}$$

where L_c is a constant, and r is the distance from the light to the point on the surface.

Radiance can be converted to reflectance by simply solving the above equation for f_r. As the light position and intensity are preseumed to be known, the resulting equation is straightforward to implement on graphics hardware. Also known on a per-fragment basis is the surface normal and position. The cosine term is simply the dot product of the light direction with the normal, and the distance is simply the length of the vector between the point on the surface and the light position.

5.8 Extracing the Diffuse Component

Most image-based models treat view-independent radiance differently from view-dependent radiance. The view-independent portion is often referred to as the "diffuse" component.

The view-independence allows for the diffuse component to be treated as a constant. In the case of a reflectance model, the diffuse term is a constant reflectance.

Actually, this discussion applies to constant terms in general. What is often called the "diffuse" term in image-based modeling often does not correspond to the physical notion of diffuse reflection. An example is the constant term in Light Field Mapping when using PCA, which actually finds the best zero-mean approximation to a matrix. Since data will not have a zero-mean, it must first be processed by subtracting out the mean. This mean becomes the constant term that Chen et al. call the "diffuse" term (Chen et al., 2002a; Chen, 2002). When they use NMF, the "diffuse" term is the minimum, which is closer to what we typically think of as a diffuse term.

The diffuse term is often constructed from assuming the minimum of the radiance or reflectance. McAllister notes that noise in the system sometimes makes this assumption problematic. He instead chose a particular percentile of the reflectance (McAllister, 2002). It is also possible to include the diffuse parameter as part of the minimization process. In working with us on the SIGGRAPH paper (Hillesland et al., 2003), Sergey Molinov used a minimum for the initial guess, and included the diffuse term in the minimization process in solving for the SBRDF model.

5.9 Summary

This concludes the description of the resampling issues and various space transformations necessary to implement the image-streaming framework. In Chapters 7 and 8, we will see how these tools are applied in practice. Although this concludes a description of the general framework and its implementation on graphics hardware, the limited precision of graphics hardware warrants further exploration in terms of its implications for this framework. Chapter 6 takes a closer look at issues related to precision and roundoff error.

Chapter 6

Error Analysis

Up until now, we have used the term "error" to describe how well the model function fits the data. This chapter addresses a different kind of error: *roundoff error*. The goal is to address the limited precision available on graphics hardware by looking more closely at the computations of the error function and the gradient, where roundoff error buildup is most detrimental. Before proceeding with this, however, we begin with a brief note on the convergence conditions for the nonlinear optimization techniques investigated in this work.

If a limit point exists, then it is fairly easy to guarantee either

$$\lim_{k\to\infty} \|\nabla f_k\| = 0 \tag{6.1}$$

or at least

$$\liminf_{k\to\infty} \|\nabla f_k\| = 0 \tag{6.2}$$

for the optimization techniques we have covered. The first case is possible for steepest descent and quasi-Newton methods with a well conditioned and positive semi-definite B_k (see Section 3.3.4 for the definition of B_k). The second case is the best that can be guaranteed for conjugate gradient. The condition for convergence is that the directions never be too close to orthogonality with the gradient, or in other words

$$\frac{\nabla f(\mathbf{p}_k)^T \mathbf{d}_k}{\|\nabla f(\mathbf{p}_k)\| \|\mathbf{d}_k\|} < -\gamma \tag{6.3}$$

for every step k and for some positive constant γ. The second condition is that the step-sizes satisfy the Wolfe condtions (Section 3.4). Proofs for this can be found in *Numerical Optimization* (Nocedal and Wright, 1999), for example. Note that this does not guarantee convergence to a minimum, only to a critical point. Second order conditions are required to guarantee convergence to a minimum; however, they are difficult to establish, particularly for large problems. There is also no guarantee that the iterates will converge to a global minimum. These issues are often handled by performing the minimization process multiple times with different starting conditions.

We begin by investigating the roundoff error in evaluating the model function, then the error function, and finally the derivative of the error function. This essentially covers the steepest descent method. For other methods, global convergence is achieved by any algorithm for which (1) every iteration produces a decrease in the objective function, and (2) every mth iteration is a steepest descent step, with step length chosen to satisfy the Wolfe conditions (Nocedal and Wright, 1999). All discussion is focused on floating-point representation. Concepts such as error in summation are introduced as needed.

6.1 Error in the Model Function

Although we might think of the model function as a mathematical function with conventional definitions of mathematical operations, it is really being executed on a finite-precision computing device, with all of its accompanying implications. For example, the single term LFM function is

$$f(\mathbf{s}, \omega) = g(\mathbf{s}s)h(\omega), \tag{6.4}$$

whereas what is actually computed is

$$\tilde{f}(\mathbf{s},\omega) = g(\mathbf{s}) \otimes h(\omega). \tag{6.5}$$

The $\otimes$ indicates an operation that only approximates true multiplication. The result will have to fit into some finite representation. The exact result would require precision equal to the sum of the number of bits in the operands. If the result is to be stored at the same precision as the operands, then some sort of rounding must occur. The difference $g(\mathbf{s})h(\omega) - g(\mathbf{s}) \otimes h(\omega)$ is referred to as *roundoff error*.

Recall that the goal is to find an approximation to the data as evaluated by the computing device. Therefore, it is precisely the second function ($\tilde{f}$) that we are really trying to fit to the data, not the first (f). Of course, we would like the finite computational version of the function to match the abstract definition well in order to make the analysis easier. We also want the computed gradient to be consistent with the computed function values, and having the function and its gradient match their abstract definitions is one way to accomplish this. Discussion of numerical error in computing the gradient will be addressed in Section 6.3. It is also desirable that the fit be suitable for multiple systems, which is another incentive to look for a fit to f rather than the $\tilde{f}$ for a particular system.

Finally, we need to ask what numerical error is acceptable. For example, if the final data will be stored in 8-bit fixed point format, then there is no need to reduce the error below about 0.5%. This is the precision that was used in the original LFM and SBRDF work.

6.2 Roundoff Error in the Error Function

The fitting process involves minimizing the error function, as well as any additional cost functions corresponding to additional objectives, such as regularization and bounds constraints. This analysis concentrates on the error function, since it constitutes a much larger set of operations relative to constraints.

As previously described, the goal is to fit a parametric model to sample data through nonlinear optimization. The approach assumes the data are generated by an unknown function, and so defines a model to approximate the original function and tries to minimize the difference between the model and the data by adjusting the model's parameters. The difference in terms of an L_2 norm is

$$E(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{S}(m(\mathbf{p},\mathbf{x}_i) - y_i)^2, \tag{6.6}$$

where $E(\mathbf{p})$ is the function we wish to minimize, and S is the number of data samples used in the fit. The model function is $m(\mathbf{p},\mathbf{x}_i)$ and the data we wish to match are the y_i. The vector $\mathbf{p}$ contains the parameters we are free to adjust in order to minimize the error function.

The critical operation here is addition. The sum of two numbers in finite precision can be written as

$$z = x \oplus y = (x+y)(1+\varepsilon) \tag{6.7}$$

The ε represents the difference between the true sum of x and y and the computed sum of x and y. The difference includes any approximations made in the summation process, and any rounding that occurs in order to fit the result into finite precision (roundoff error). We will assume that ε is bound by some constant. For example, with exact rounding and 32 bit IEEE 754 format, this constant would be 2^{-24} (IEEE, 1987). This number is not always readily available. Graphics hardware vendors do not provide information from which ε can be surmised. Therefore, I have developed a technique to measure ε, which is given in Appendix A.

Unfortunately, computing the total error requires that we make a number of successive sums, say $\sum_i^n x_i$. What would actually be computed is

$$\sum_{i=1}^{n} x_i(1+\delta_i) \tag{6.8}$$

where $|\delta_i| < (n-i)\varepsilon$. Derivation of this bound can be found in many books on numerical analysis. The book by Wilkinson includes derivation of this bound and many others (Wilkinson, 1963).

If all the terms are positive then we can derive a simpler expression; although it introduces another level of conservatism:

$$\sum_{i=1}^{n} x_i \delta_i < \delta \sum_{i=1}^{n} x_i \tag{6.9}$$

where $\delta < n\varepsilon$ is the relative error with respect to the true solution. This is a common model for assessing roundoff error: that each operation (excepting catastrophic cancellation) results in an accumulation of ε in the relative error. However, this is a conservative bound, and in order to be achieved would require that every intermediate value have a certain worst-case mantissa. For example, in a scheme where all values are rounded down this worst case bound is based on assuming that all mantissas are an infinite sequence of repeating ones. If instead we assume an evenly distributed set of mantissas the relative error grows as $\sqrt{n}$.

The standard approach to handle the summation of a large number of terms is to compute the sum in double precision. If, for example, the summation is done using a floating point representation with 53 bits of precision and exact rounding, and the results are to be stored in a format that has 24 bits of precision, up to $2^{53-24} = 2^{29}$ operations may be performed without losing precision in the result.

Some architectures, such as graphics hardware, do not support double precision, instead relying on 10 to 23 bits of mantissa. Since adding 256 terms will result in the potential loss of about eight bits of precision, this creates a potential problem in performing long calculations.

One tactic used to alleviate precision problems is to sort the terms from smallest to largest before adding. This can be time consuming and, in fact, is not practical in the streaming framework. On the other hand, *Kahan summation* (Algorithm 14) requires no re-ordering, and reduces relative error to only 2ε in each term (Kahan, 1965). More specifically, Kahan summation results in

$$\sum_{i=1}^{n} x_i(1+\delta_i) + O(n\varepsilon^2)\sum |x_i| \tag{6.10}$$

where $|\delta_i| < 2\varepsilon$. This is derived in (Knuth, 1998). Kahan summation subtracts a "correction" with each term of the summation. The correction value is actually the error that was incurred in adding the previous term.

Algorithm 14 Kahan Summation

```
1: sum = $x_1$
2: correction = 0
3: for $i = 2 \ldots n$ do
4:     Y = $x_j$ - correction
5:     temp = sum + Y
6:     correction = (temp - sum) - Y
7:     sum = temp
8: end for
```

Line 5 introduces error due to roundoff. Line 6 recovers this error to be used as a correction in the next iteration.

Roundoff error restricts how small of an improvement in the error function is measurable. For naive summation, worst case bounds tell us we cannot discern any improvement in the error function by less than a factor of $n\varepsilon$. For Kahan summation, this becomes 2ε. This condition could be added to the Wolfe conditions in a line search.

6.3 Gradient Error

The gradient for the error function is

$$\nabla E(\mathbf{p}) = \mathbf{J}^T\mathbf{R} = \sum_S (m(\mathbf{p},\mathbf{x}_i) - y_i)\nabla m(\mathbf{p},\mathbf{x}_i), \tag{6.11}$$

where **J** is the Jacobian and **R** is the vector of residuals. For convenience, I will discuss a single element of the gradient vector, which is a partial derivative. I also exclude the tilde notation, which was previously used to distinguish the value computed in finite precision.

$$\frac{\partial}{\partial p_j}E(\mathbf{p}) = \sum_{i=1}^{c_j}(m(\mathbf{p},\mathbf{x}_i) - y_i)\frac{\partial}{\partial p_j}m(\mathbf{p},\mathbf{x}_i) \tag{6.12}$$

where c_j is the number of non-zeros in column j of the Jacobian. This is important because it can be far less than the number of samples. For example, LFM data would have no more than approximately 2^8 non-zeros in a column, even though the total number of samples would be roughly 2^{16}.

6.3.1 Catastrophic Cancellation

The difference in computing the gradient as opposed to computing the error function, is that the terms in the summation can be both positive and negative. The terms are the result of multiplying the residual by the derivative of the model function $(m(\mathbf{p},\mathbf{x}_i) - y_i)\frac{\partial}{\partial p_j}m(\mathbf{p},\mathbf{x}_i))$. Since the terms are computed quantities, there is cause to consider catastrophic cancellation. Catastrophic cancellation occurs when two computed quantities have nearly the same magnitude but are of opposite sign, such that when they are added together most of the digits of precision are lost, leaving only questionable digits.

Now we look again at the terms in the derivative. If the residual is small, then the term should become unimportant in computing the derivative. The danger is that the gradient of the model function may be large. Unfortunately, this depends on the model function. However, there are some measures that can be taken to alleviate this problem. The simplest solution is to simply drop all terms where the residual is small. Another option is to use bounds constraints to keep the derivative bounded. For example, LFM parameters can be restricted to the range of [0,1], resulting in a term of the derivative restricted to being in [0,1] as well.

In cases where there is cancellation between terms with large residuals, the solution can be more difficult. Since each term is the product of a residual and a derivative, there are two cases to consider: (1) the residuals have opposite sign, but the derivatives have the same sign and (2) the residuals are of the same sign, but the gradients are of opposite sign. These situations occur in cases where the model function is ill-suited to match the data, or as a result of choosing an initial guess that is ill-suited for the model function.

6.3.2 Summation

In the previous subsection, we approached the problem by looking at each term of the summation. Here we look at accumulation of error in the summation process. The computed sum using the shorthand $x_i = (m(\mathbf{p},\mathbf{x}_i) - y_i)\frac{\partial}{\partial p_j} m(\mathbf{p},\mathbf{x}_i)$ is

$$\frac{\partial}{\partial p_j} E(\mathbf{p}) = \sum_{i=1}^{c_j} x_i(1+\delta_i) \tag{6.13}$$

where $|\delta_i| < (c_j - i)\varepsilon$ for naive summation, and $|\delta_i| < 2\varepsilon$ for Kahan summation. Although we cannot make the same simplification as we did for the error function, we can define $|x_{\max}|$ as the summand with largest absolute value, and use the Cauchy-Schwarz inequality to claim the following:

$$\left| \sum_{i=1}^{c_j} x_i \delta_i \right| < |x_{\max}| \sum_{i=1}^{c_j} |\delta_i| \tag{6.14}$$

$$|x_{\max}|\sum_{i=1}^{c_j}|\delta_i| = |x_{\max}|\varepsilon\sum_{i=1}^{c_j} i = |x_{\max}|\varepsilon K \tag{6.15}$$

where $K = c_j(c_j+1)/2$ for naive summation, and $K = 2c_j$ for Kahan summation. Bounds on the parameter values and function range helps to bound the size of this error. Even assuming $|x_{\max}| \leq 1$, as would be the case for LFM, this error model tells us that a calculation involving 16 bits could be corrupted with as little as $c_j \approx \sqrt{2^{16}} = 2^8 = 256$ summands using naive summation. Coincidentally, this is approximately the number of terms in the case of a typical LFM model. However, we've made some fairly conservative assumptions. For Kahan summation, it would require approximately 2^{15} terms, which is much more reasonable. As we will see in Chapter 7, precision was not actually an issue in practice; in fact, fair convergence was achieved with only 10 bits of precision in the mantissa using the naive summation.

Note that the sequence of iterates will converge as long as

$$\frac{\nabla f(\mathbf{p})^T\mathbf{d}}{\|\nabla f(\mathbf{p})\|\|\mathbf{d}\|} < -\gamma \tag{6.16}$$

and the stepsize satisfies the Wolfe conditions. The parameter γ can be adjusted according to confidence in the computation of the search direction. More conservative choices of γ can result in slower convergence, as it will result in more frequent fall-backs to the steepest descent direction.

6.4 Summary

The critical computations are the evaluation of the error function and its gradient. Roundoff error can corrupt these calculations, particularly because they involve large sums. I have given some guidelines on the bounds on errors caused by roundoff. Since graphics hardware supports limited precision, I have included a description of Kahan summation, which greatly reduces roundoff error over the naive approach. To wrap up this chapter, remember that convergence depends only on a descent direction and a stepsize that meets the Wolfe conditions.

Chapter 7

Case Study: Light Field Mapping

This chapter documents the results of applying the image-streaming framework to Light Field Mapping (Chen et al., 2002a), a decomposition-based model for radiance. Results with the image-streaming framework were first reported in the SIGGRAPH paper (Hillesland et al., 2003). Both steepest descent and conjugate gradient implementation results are included. The goals in this chapter are to:

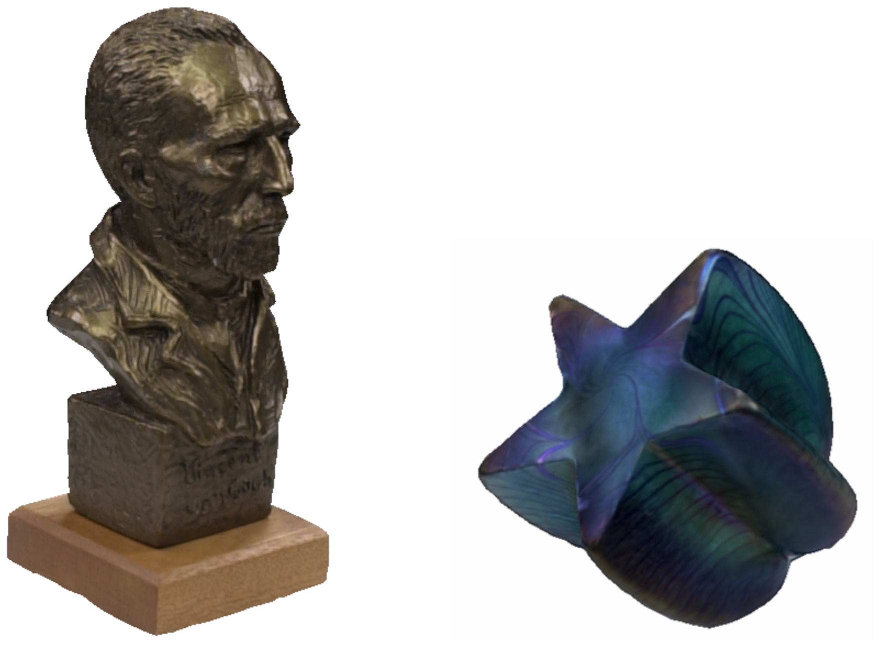
1. measure the error of the models in reproducing the reference images,
2. test the effects of limited GPU numerical precision and accuracy,
3. evaluate time efficiency of implementations under this framework, and
4. measure bandwidth and computational utilization.

The first task is to provide details on how LFM is implemented using the image-streaming framework proposed. This is the subject of the first section.

7.1 Implementation

Chapter 2 introduced Light Field Mapping (LFM). The implementation included a single term ($K = 1$ in Equation 2.11), therefore each model is expressed as

$$m(\mathbf{p},\mathbf{x}) = g(\mathbf{s})\,h(\omega), \tag{7.1}$$



(a) *Bust* **(b)** *Star*

Figure 7.1: *The bust model and the star model are used for the light field mapping approximation experiments.*

where g is the surface map with parameterization $\mathbf{s}$ and h is the view map with parameterization ω. There is an independent model for each color channel. First, note that $\mathbf{s}$ and ω constitute the inputs such that $\mathbf{x} = [\mathbf{s}\,\omega]$. The functions $g(\mathbf{s})$ and $h(\omega)$ are texture lookup functions. The texel values are the parameters, $\mathbf{p}$. To distinguish between surface map and view map texels, I will write them as $\mathbf{p}_s$ and $\mathbf{p}_\omega$. A complete model is made up of on the order of thousands of LFM patches corresponding to vertices of the model.

From Equation 2.9 and Figure 2.3 it follows that each light field mapping approximation uses three surface maps per triangle and one view map per vertex. We tiled the view maps into one texture map and the surface maps into another texture map. The typical resolution of individual surface maps and view maps used in our experiments was 8×8 pixels.

We now need to think about how this fits into the idea of an optimization problem.

For review, I write the error function again:

$$E(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{S}(m(\mathbf{p},\mathbf{x}_i) - y_i)^2, \tag{7.2}$$

where S is the number of data samples used in the fit, and where each data sample consists of an input set ($\mathbf{x}_i$) and an output (y_i).

LFM can be thought of as a sequence of optimizations, one per vertex of the model. Each optimization corresponds to finding the view map and the surface map texels that best approximate the portion of the surface light field for the triangle ring of the given vertex.

$g(\mathbf{s})$ and $h(\omega)$ are texture lookups. We implemented a model with nearest neighbor texture filtering. This means that only a single texel (parameter) is used for each texture lookup. Let's define $T(\mathbf{u}) : \Re^2 \rightarrow Z$ as the nearest filter lookup into a texture that takes a 2D texture coordinate $\mathbf{u}$ and returns an integer Z for indexing into the set of texels in the texture map. Assume the texture lookup function is the same for both the surface and view maps. We will express the surface map texels as a vector $\mathbf{g}$, with the jth element being g_j.

Before writing the derivative, let's define indices returned by texture lookup functions to simplify notation. A texture lookup function $T(\mathbf{s}_i)$ will return the index of a surface map texel, which we will call i_s, and a texture lookup function $T(\omega_i)$ will return the index of the view map texel i_ω. Note that there is a surface map texture coordinate and view map texture coordinate for each sample i.

If we write the derivative with respect to a surface map parameter, g_j, we have

$$\frac{\partial}{\partial g_j} m(\mathbf{p},\mathbf{x}) = \sum_{i=1}^{S}\left(g_{i_s}\frac{\partial}{\partial g_j}h_{i_\omega} + h_{i_\omega}\frac{\partial}{\partial g_j}g_{i_s}\right) = \sum_{\{i:i_s=j\}} h_{i_\omega} \tag{7.3}$$

The product rule for derivatives is used here. Because h_{i_ω} does not depend on g_j, the term involving its derivative is zero. $i : i_s = j$ means "for all i such that a nearest texture lookup using $\mathbf{s}_i$ results in returning index j". This comes from the derivative of the surface map function:

$$\frac{\partial}{\partial \mathbf{g}_j} \mathbf{g}_{i_s} = \begin{matrix} 1 & \text{if } i_s = j \\ 0 & \text{if } i_s \neq j \end{matrix} \tag{7.4}$$

Although, we began with a sum over all data points i, only a subset result in nonzero partial derivatives, which is why c_j, the number of nonzero terms, was mentioned in Chapter 6.

To compute the derivative with respect to a surface parameter requires a sum reduction across view map parameters indexed by $i : i_s = j$, as indicated in Equation 7.3. This is accomplished by virtue of the fact that only one view direction is used for each image. In other words, the condition $i_s = j$ corresponds to the condition that the texel i_s is visible in the image.

Similarly, the derivative with respect to a view map parameter would be

$$\frac{\partial}{\partial \mathbf{v}_j} m(\mathbf{p}, \mathbf{x}) = \sum_{\{i : i_\omega = j\}} \mathbf{g}_{i_s)} \tag{7.5}$$

To compute the derivative with respect to the view map parameter requires a sum reduction across the surface map. This must occur on a per-image basis. Since there is only one view per image for a given patch, the single scalar that results from the reduction is rendered into the appropriate place in the view map. This is the process described in Section 5.6.

There is an important distinction in the model used in this process in contrast to how the model is actually used once it is complete. In each image, the view direction is computed at the vertex, and this single view direction is used across the entire patch. In rendering the final model, the view direction is computed at each vertex, but it is interpolated across the entire patch, such that each surface patch element uses a potentially different

Models	Poly. Count	Param. Count	Param. Tex. Size	Image Count	Image Size
Bust	7228	0.91MT	1.64MT	339	161MB
Star	6093	0.69MT	1.23MT	282	159MB

Table 7.1: *Models used in the experiments. The 3rd column shows the model parameter count, the 4th column shows the count of pixels used to store the model parameters. The counts are given in megatexels [MT].*

view direction. This is exactly the same system used in the original LFM work (Chen et al., 2002a; Chen, 2002).

Rather than using the Wolfe conditions, we found that simply checking for sufficient decrease using

$$f(\mathbf{p}+\alpha\mathbf{d}) < f(\mathbf{p}) \tag{7.6}$$

was adequate in our test cases.

7.2 Models

Figure 7.1 shows the bust and the star models from (Chen et al., 2002a) that were used for the LFM experiments. Table 7.1 lists the pertinent information about the datasets: the polygon count, the model parameter count, the size of the radiance dataset and the number of images.

The total number of model parameters used for each model is given in Table 7.1. Textures used to store the model parameters are slightly larger because of the extra space required for packing. (We were not using a very efficient packing scheme; half of the space for surface maps was unused.) The size of the textures allocated to the model parameters is also given in the table. The implementation uses fixed-size surface maps for all triangles, each having 32 texels. View map resolution is 8×8 texels for the 2π hemisphere of directions.

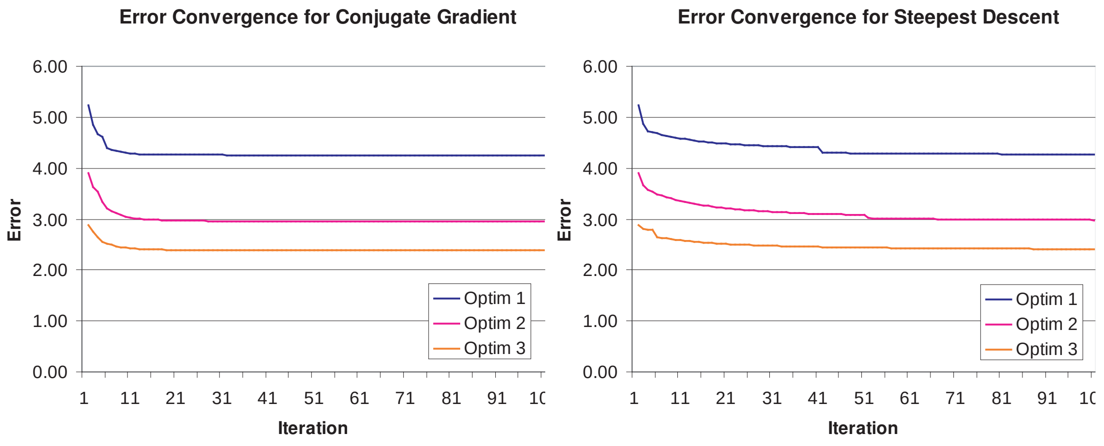


Figure 7.2: *The left graph shows the converge rate of the conjugate gradient method on three randomly selected optimization problems from the bust dataset. The right graph shows the convergence rate for the steepest descent method on the same three problems. Note that the conjugate gradient method included a quadratic model for computing a stepsize (Section 3.4.2).*

7.3 Steepest Descent vs. Conjugate Gradient

The conjugate gradient method (Section 3.3.2) with a quadratic stepsize estimate (Section 3.4.2) achieves much better convergence than the steepest descent method (Section 3.3.1) with a backtracking line search (Section 3.4.1). It often requires an order of magnitude fewer iterations to converge and usually finds a better local minimum. The comparison is shown in Figure 7.2. The left graph has the convergence rate for the conjugate gradient method on 3 randomly selected optimization problems from the star dataset. The right graph has the rate for the steepest descent method on the same 3 problems. The conjugate gradient method converges significantly faster. For example, “Optim 2” reaches the minimum in about 25 iterations using the first method and it takes about 250 iteration to get down to the same error level using the second method. This ratio is consistent across different optimizations and different datasets. This comparison was performed using a CPU implementation.

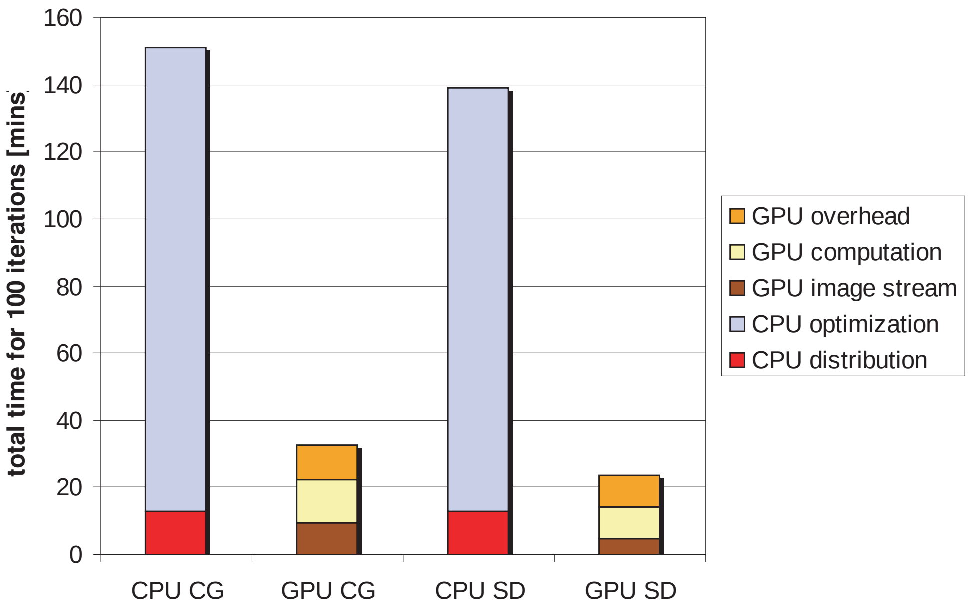


Figure 7.3: *Execution time comparison between the CPU and the GPU implementations of the light field mapping approximation running on the star model. We compare the performance of the conjugate gradient (CG) and the steepest descent (SD) methods.*

7.4 CPU and GPU Implementation Comparison

To evaluate performance relative to a CPU implementation, we implemented a CPU version of the light field mapping approximation using nonlinear optimization following the approach discussed in Section 4.1.1 and compared it against the GPU implementation.

The distribution step for the CPU implementation writes out the following information to the disk for each radiance sample falling on a given triangle: target value, residual value, surface map coordinate and 3 view map coordinates. The distribution step in this case is still computed on the GPU—implementing the whole renderer in software would require substantial effort. Once data distribution is finished, the algorithm sequentially solves each individual optimization on the CPU.

Figure 7.3 compares the computational performance of the GPU and the CPU implementations. For the CPU, the total run time is divided into the distribution step and the optimization step. For the GPU, it is divided into image streaming, computation and overhead for context switches. The numbers are given for the full star model running for exactly 100 iterations. We used ATI's All-In-Wonder® Radeon™ 9700 to run the GPU implementation and Intel's 2GHz Pentium® 4 for the CPU implementation.

Image data were stored on the disk in uncompressed format. Streaming images to the graphics card consumes approximately 20% of the run time. The raw image data was no more than about 160MB in size. Therefore, this time could be reduced by reading the images into the host memory and streaming them to the GPU from there. In the CPU implementation, this would not be possible, as it works with resampled data that would not fit into host memory.

In both cases, the GPU performance is more than five times better than the CPU performance. The GPU implementation of the steepest descent method spends approximately half the total execution time on context switches. For the same implementation of the conjugate gradient this portion of execution time goes up to almost 66%. Section 7.7 proposes ideas on how this might be reduced.

7.5 Memory Requirements

An image-streaming approach uses memory efficiently, since it only requires $O(n)$ of storage, where n is the number of model parameters. However, texture sizes become larger when using floating-point texels. Memory on the graphics card is approximately an order of magnitude smaller than on the CPU. Although host memory is also accessible to the graphics processor, the data path is much slower; also, host memory cannot be used directly as a render target.

Since the Radeon 9700 has only about 110MB of video memory available for the textures, we were limited in the size of the models we could process. For example, the conjugate gradient algorithm running on the full star model uses about 7.8 megatexels for the data structures related to the surface maps and 1.1 megatexels for the data structures related to the view maps. Since each 32-bit floating-point RGBA texel requires 16 bytes, this translates to 142MB of texture memory. Therefore, to solve the full star model, we had to use fairly low resolution light field maps and 16-bit floating-point numbers. The use of 16-bit floating-point numbers did not have a negative impact on the convergence as shown in Section 7.6.

Better tiling and distribution of texels according to the projected triangle screen size would have allowed for more efficient use of this limited resource, but we did not implement these features.

7.6 Convergence and Error Analysis

To compare the error of the original light field mapping approximation algorithm with the nonlinear optimization implementation, we computed a one-term approximation of the star model using principal component analysis (Chen et al., 2002a) and the CPU version of nonlinear optimization. In both cases, we used high resolution surface maps (144 texels on average) and view maps (256 texels).

The errors reported in Figure 7.4 are computed based on the difference between the input image and the rendered image using both APE (average pixel error) and PSNR (pixel signal-to-noise ratio) for the *foreground* pixels only. The errors for the two methods described above indicate that, when using the same resolution light field maps, the nonlinear optimization algorithm yields a better approximation than the original algorithm based on matrix factorization. The most likely cause of this is that the new method does not perform the extensive resampling required by the matrix factorization approach.

To compare the error of nonlinear optimization running on the CPU and the GPU, we had to use low resolution light field maps and 16-bit floating-point numbers on the GPU. The errors reported in Figure 7.4 for these two methods, although much higher than in the first experiment, are very similar to each other. This indicates that the use of 16-bit floating-point numbers on the GPU did not have a detrimental effect on the convergence

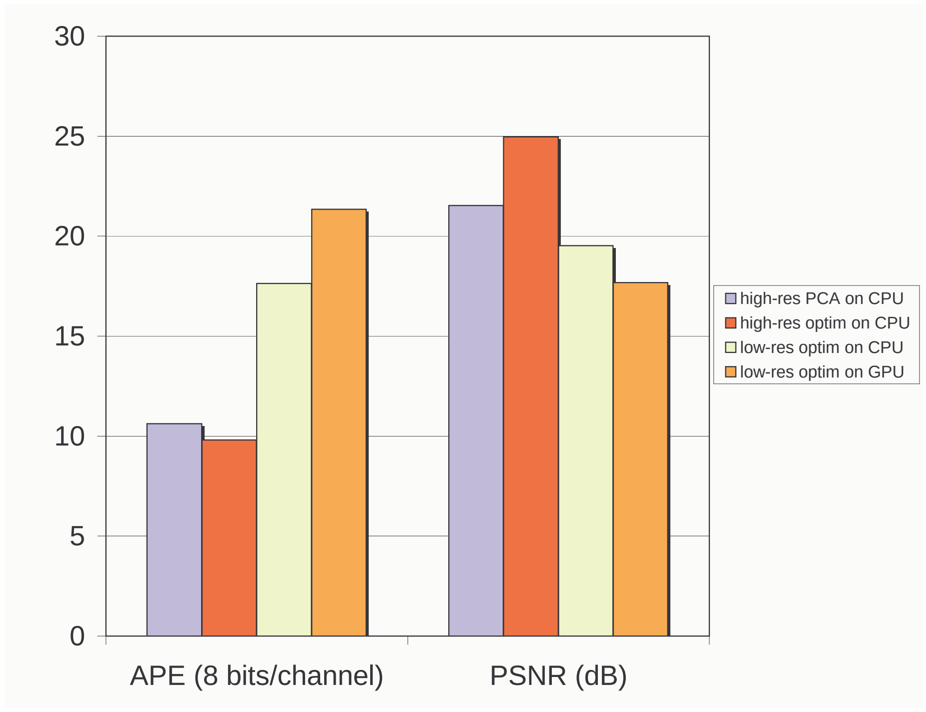


Figure 7.4: *Convergence and error analysis for four different approximation methods. Each value is an average across three 8-bit channels. APE is in the range of [0-255].*

of the algorithm and that when we have more memory available on the card, we will be able to achieve the high quality of approximation that the light field mapping method provides.

7.7 Computation and Bandwidth Requirements

We calculated the computation and the memory bandwidth requirements of our algorithms by counting the number of instructions of each fragment program and the number of memory reads and writes, and multiplying them by the number of times each fragment program is executed. Figure 7.5 shows computation and memory bandwidth utilization for the star model. The Radeon 9700 Pro is capable of a peak 2.6 Gops of arithmetic operations in the fragment unit and 0.65 Gops of 64-bit wide texture reads in parallel.

Image data are stored on the disk in uncompressed format. As shown in Section 7.4,

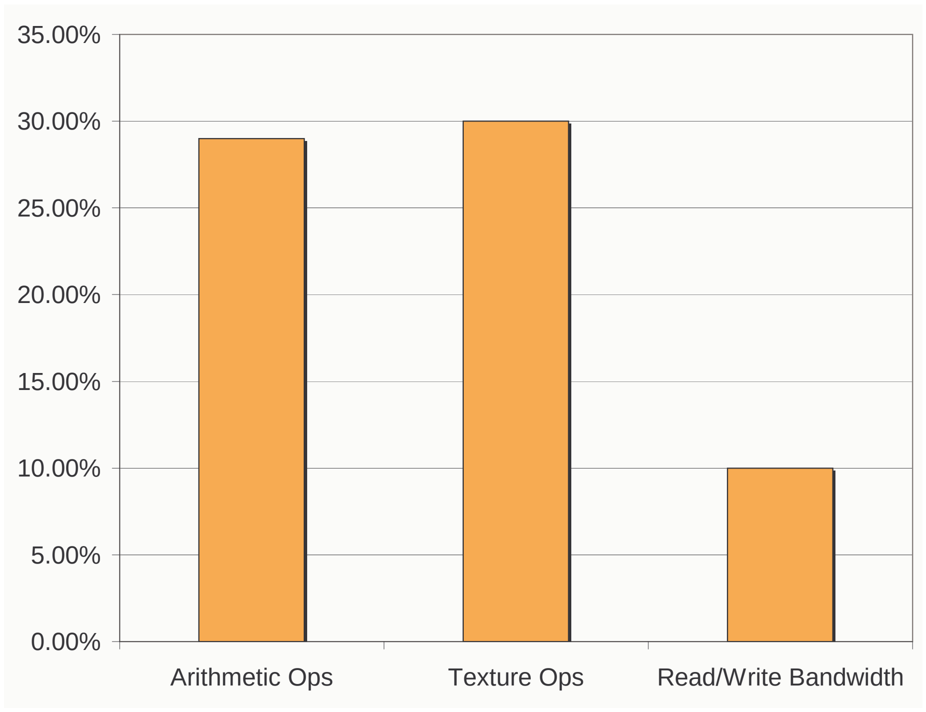


Figure 7.5: *Computation and memory bandwidth utilization for LFM generation using the steepest descent method.*

streaming the reference images to the graphics card consumes about 20% of run time. This time could be reduced by reading the images into the host memory and streaming them to the GPU from there.

The 29% utilization rate for the LFM approximation experiment could become much higher in the future. The two most obvious improvements are: (1) streaming images from the host memory instead of the disk; (2) combining rendering passes by using multiple render targets and by having longer fragment programs. Combining passes would improve the performance by reducing the effects of the synchronization that happens with each context switch and by eliminating redundant texture reads. This latter change would shift the utilization towards the arithmetic maximum by reducing texture operations. We used only a fraction of the available memory bandwidth between the GPU and video memory.

Chapter 8

Case Study: Spatially-Varying BRDFs

This chapter shows how this framework has been applied to an SBRDF model based on per-texel Lafortune reflectance functions, which is a parametric BRDF model. The results were first reported in the SIGGRAPH paper (Hillesland et al., 2003). There was no reference CPU implementation available, so this case study was focused on simply demonstrating that the SBRDF model does fit into the proposed framework, and to look at its computational efficiency.

8.1 Implementation

The SBRDF model represents a simpler model to implement in this framework in the sense that the texel-sized patch does not require sophisticated gathers. The texel-sized patch means there are far more independent patches to be solved relative to the LFM model, but requires fewer parameters for each patch. The main difficulty of the SBRDF model is that it is more highly nonlinear than the LFM model due to an exponent parameter.

We followed McAllister et al. (McAllister et al., 2002; McAllister, 2002) in adopting the Lafortune model:

$$m(\mathbf{p},\mathbf{x}) = \rho_d + \sum_j \rho_{s,j} (C_{x,j}\, u_x\, v_x + C_{y,j}\, u_y\, v_y + C_{z,j}\, u_z\, v_z)^{n_j}.$$

The parameters **p** in the above function are ρ_d, $\rho_{s,j}$, $C_{x,j}$, $C_{y,j}$, $C_{z,j}$, and n_j. The input variables, **x**, are u_x, u_y, u_z, v_x, v_y, and v_z. They are projections of the viewing and lighting directions on the local coordinate system.

We assume isotropic reflection ($C_{kx} = C_{ky}$) modeled by a single ($K = 1$) forward-reflective term (no retro-reflection). Additionally, we use just one m for all color channels; therefore, the model function actually returns a three-vector of red, green and blue values. This means that the model parameter vector **p** uses nine independent parameters to describe the BRDF at a given surface location: $[\rho_{Rd}\ \rho_{Gd}\ \rho_{Bd}]$, $[\rho_{Rs}\ \rho_{Gs}\ \rho_{Bs}]$ and $[C_{xy}\ C_z\ n]$, where the k index has been dropped, and I have defined $C_{xy} = C_x = C_y$. The diffuse parameters $[\rho_{Rd}\ \rho_{Gd}\ \rho_{Bd}]$ were included in the optimization process for the first several iterations; the parameters were fixed after this point for the remainder of the process.

Stating the error function again:

$$E(\mathbf{p}) = \frac{1}{2}\sum_{i=1}^{S}(m(\mathbf{p},\mathbf{x}_i) - y_i)^2, \tag{8.1}$$

where S is the number of data samples used in the fit and where each data sample consists of an input set ($\mathbf{x}_i$) and an output (y_i). Since the model covers all three color channels, one difference between this case study and the case of LFM is that the sum over samples includes a sum across color channels. The other difference is that every parameter is used for each data sample, so that $c_j = S$, where c_j is the number of nonzero values in the Jacobian. c_j was introduced in Chapter 6.

Texture filtering is not directly part of the SBRDF model. The resampling from image space to SLS (discussed in Section 5.2.2) is what decouples the texture filtering issue from the model fitting process.

The lighting model is a single point light source. Therefore, the radiance can be expressed as

$$L_r(\omega_r) = m(\mathbf{p},\mathbf{x})L_c \cos(\theta_i)/r^2, \tag{8.2}$$

where L_c is a constant, and r is the distance from the light to the point on the surface. The light was far enough from the object that $1/r^2$ could be considered a constant and can be absorbed into L_c.

The final issue is that it was necessary to convert radiance to reflectance, which was computed on the fly. One drawback of fitting to reflectance is that the data samples become unbounded after dividing out the cosine; therefore, we rejected samples where the light hit the surface at a grazing angle. In particular, we chose to throw out all samples where $\cos(\theta_i) < 0.1$. With both incident and exitant radiance in [0,1], this bounds the value of the reflectance data sample to [0,10].

The gradient with respect to $[\rho_{Rd}\ \rho_{Gd}\ \rho_{Bd}]$ is simply a constant vector of ones. In other words,

$$\frac{\partial}{\partial_{\rho_{Rd}}} m(\mathbf{x},\mathbf{p}) = \frac{\partial}{\partial_{\rho_{Rg}}} m(\mathbf{x},\mathbf{p}) = \frac{\partial}{\partial_{\rho_{Rb}}} m(\mathbf{x},\mathbf{p}) = 1. \tag{8.3}$$

Its simple linear nature is what drove the decision to only include it in the first several iterations. This is in contrast to previous work, where either a minimum (Lafortune et al., 1997; Lensch et al., 2003) or constant factor (McAllister et al., 2002; McAllister, 2002) were used.

The derivatives with respect to $[\rho_{Rs}\ \rho_{Gs}\ \rho_{Bs}]$ are

$$\frac{\partial}{\partial_{\rho_{Rs}}} m(\mathbf{x},\mathbf{p}) = \frac{\partial}{\partial_{\rho_{Rs}}} m(\mathbf{x},\mathbf{p}) = \frac{\partial}{\partial_{\rho_{Rs}}} m(\mathbf{x},\mathbf{p}) = (C_{xy}\ (u_x\, v_x + u_y\, v_y) + C_z\, u_z\, v_z)^n \tag{8.4}$$

The derivatives with respect to C_{xy} and C_z are

$$\frac{\partial}{\partial_{C_{xy}}} m(\mathbf{x},\mathbf{p}) = n\ (u_x\, v_x + u_y\, v_y)\ (C_{xy}\ (u_x\, v_x + u_y\, v_y) + C_z\, u_z\, v_z)^{n-1} \tag{8.5}$$

$$\frac{\partial}{\partial_{C_z}} m(\mathbf{x},\mathbf{p}) = n\, u_z\, v_z\ (C_{xy}\ (u_x\, v_x + u_y\, v_y) + C_z\, u_z\, v_z)^{n-1} \tag{8.6}$$

Models	Poly. Count	Param. Count	Param. Tex. Size	Image Count	Image Size
Ornament	3690	0.86MT	1.73MT	1760	0.98GB

Table 8.1: *SBRDF Model Statistics. The 3rd column shows the model parameter count, the 4th column shows the count of pixels used to store the model parameters. The counts are given in megatexels [MT].*

The derivative with respect to n is

$$\frac{\partial}{\partial_n} m(\mathbf{x}, \mathbf{p}) = a^n \ln(a) \tag{8.7}$$

where $a = C_{xy}\ (u_x\, v_x + u_y\, v_y) + C_z\, u_z\, v_z$.

Note that there are a number of constraints worth establishing. The first is the forward-reflective constraint. This was enforced by throwing out all samples that correspond to retro-reflection, i.e. throwing out all samples where $u_x\, v_x + u_y\, v_y > 0$ and constraining $C_{xy} \leq 0$. Back-face culling removes all samples where $u_z < 0$. By throwing out samples where $v_z < 0$ and constraining $C_z \geq 0$ we keep the C_z term positive as well. Most of these constraints were maintained by clamping, rather than using the penalty method.

The SBRDF model was solved using steepest descent for direction, and the back-tracking method for stepsize.

8.2 Model

The ornament model, shown in Figure 8.1, was used to compute the Lafortune fit to SBRDF data. The dataset for it was generated synthetically in 3D Studio Max®. The surface consists of 3 distinct materials produced using 2 reflectance models: Oren-Nayar-Blinn (white) and Lafortune (gold and green). Table 8.1 lists the pertinent information about the dataset: the polygon count, the model parameter count, the size of the radiance dataset and the number of images.

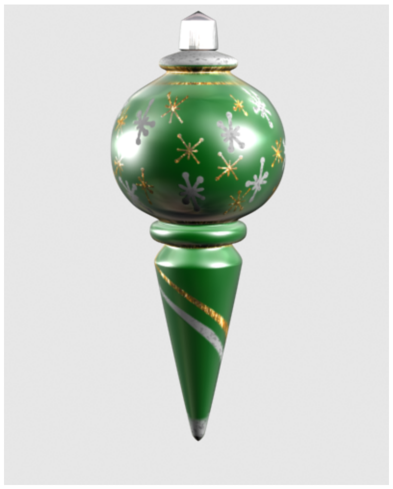


Figure 8.1: *The ornament model is used for the Lafortune fit to SBRDF data.*

As explained in Section 8.1, the Lafortune fit to SBRDF data uses nine parameters per surface location, which we store in three texture maps. We use fixed-size textures for all triangles, each having 128 texels.

The total number of model parameters used for the model is given in Table 8.1. Just as in the case for LFM, textures used to store the model parameters are slightly larger because of the extra space required for packing. The size of the textures allocated to the model parameters is also given in the table.

8.3 Memory Requirements

The same issues with respect to memory requirements were also encountered with the SBRDF model. Again, we used a fairly inefficient tiling scheme, and chose to use the same number of texels for every triangle of the model rather than choosing a resolution according to screen-space projection size. We also resorted to using 16-bit floating point textures, but these proved to be sufficient for adequate convergence.

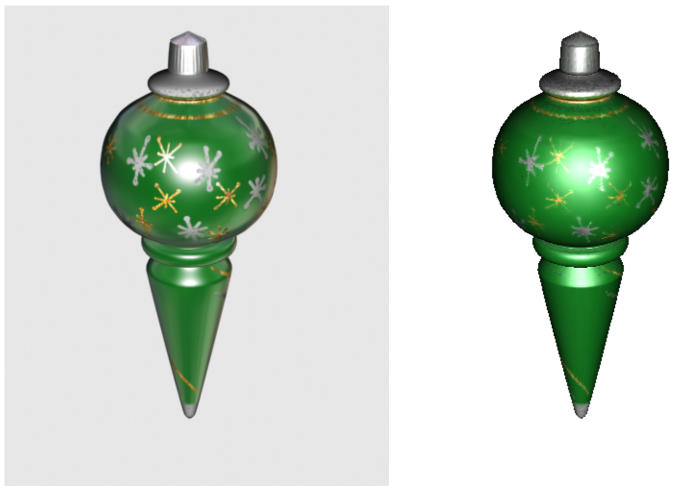

(a) *Original Image* **(b)** *Model Image*

Figure 8.2: *Visualization of the results for the Ornament.*

8.4 Convergence and Error Analysis

As already mentioned, there was no CPU implementation available to us. Another complication was that the dataset was somewhat poor in the sense that the image format could not capture the high dynamic range caused by specularities, and therefore, the data had clamped specular peaks. However, one of the advantages of the image-streaming approach using graphics hardware is that the solution process can be directly visualized. We were able to observe that the model did indeed converge to a reasonable solution by visual inspection. The solution included a distinction between highly specular and more diffuse regions, as well as correct colors.

8.5 Computation and Bandwidth Requirements

We calculated the computation and the memory bandwidth requirements by counting the number of instructions of each fragment program and the number of memory reads and writes and multiplying them by the number of times each fragment program is executed. Figure 8.3 shows computation and memory bandwidth utilization for the ornament

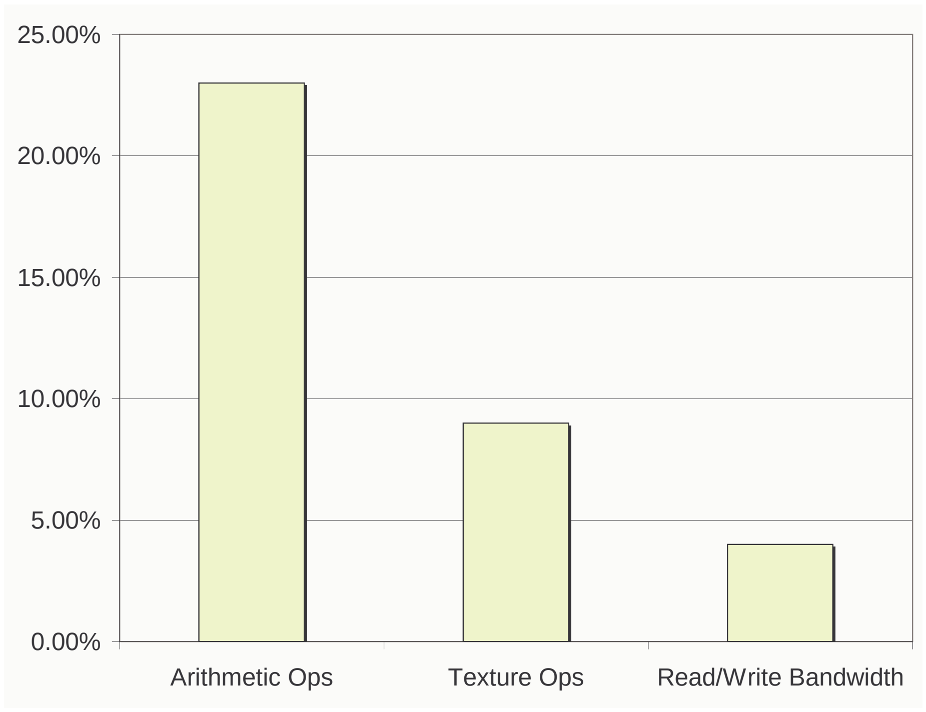


Figure 8.3: *Computation and memory bandwidth utilization for the ornament model using the steepest descent method.*

model. The Radeon 9700 Pro is capable of a peak 2.6 Gops of arithmetic operations in the fragment unit and 0.65 Gops of 64-bit wide texture reads in parallel. For the SBRDF experiment, streaming images from the disk constituted about 70% of the execution time.

Chapter 9

Conclusions and Future Work

I have proposed an image-streaming framework for generating image-based models. This framework differs from previous approaches in that it keeps the complete model as the working set, and processes the raw images. The image-streaming framework is described in Chapter 4. As the technique is described, I show why this approach works nicely for building image-based models using nonlinear optimization. Section 4.1 shows where this approach has its advantage: in reducing the bandwidth requirements for the data stream. Chapter 5 describes the tools to implement this framework on programmable graphics hardware.

This technique should apply to shading models that store parameters in textures, particularly those that include spatial variation. Chapter 2 lays out the characteristics of both analytic and decomposition models and show that they can both be handled under the framework of a function fitting process. Therefore, the framework proposed in this dissertation will apply to a fairly wide class of models.

Many numerical techniques rely on access to double precision support, which is not available on modern GPUs. Chapter 6 addresses the issue of roundoff error in a limited precision context, particularly with respect to summation, which is especially problematic. I apply this analysis to nonlinear optimization for image-based modeling to identify where problems may occur in my proposed framework. I also discuss some of the measures that can be taken to alleviate these problems.

Modern GPUs do not conform to IEEE floating-point operation standards even for single precision, nor do vendors document their behavior. Appendix A shows that floating-point behavior can be measured, and that current graphics hardware exhibits reasonable behavior compared to what is called for in the IEEE standard.

The above concepts are brought together in two case studies: one for constructing LFM models (Chapter 7) and one for constructing an SBRDF model (Chapter 8), which represent two diverse image-based model types.

In summary, I have shown that the technique is widely applicable (Chapter 2), produces correct results (Chapter 6), and shows favorable performance (Chapter 4). I also back up these arguments with empirical evidence (Chapters 7 and 8). I have therefore demonstrated the thesis.

> An image-streaming approach to nonlinear optimization is a widely applicable and efficient technique for image-based model construction.

9.1 Limitations

The limitations of this framework bear repeating.

- I have assumed known geometry and viewing conditions. To handle relighting, only models with known point light sources have been tested.
- Nonlinear optimization can be a tricky process, requiring a good initial guess to obtain a reasonable solution. This is because the techniques explored in this work are only designed to find a local minimum. Furthermore, the error function must be at least C^1.
- The effectiveness of the technique relies on the number of iterations being less than the number of patches, and concentrates on reducing bandwidth requirements in exchange for additional computational cost.

9.2 Future Work

There are two main aspects to the image-streaming framework that motivate future work. The first is its wide applicability. The first two suggestions for future work build on this by suggesting how to simplify the process further with the goal of making it possible to simply plug in different image-based model functions without much effort. This would allow for relative ease in exploring different representations for a given data set.

The conventional approach to image-based modeling uses a pre-process that relies on fixed geometry and iteration over a fixed set of images. By contrast, the image-streaming approach makes no such assumptions, while maintaining favorable performance. This is the second main aspect that motivates future work. The second two suggestions are ways in which the framework can be adapted for applications that are not possible in the conventional pre-processing approach.

9.2.1 Finite Differencing and Automatic Differentiation

Evaluating the gradient can sometimes be difficult. In the case of LFM, for example, the gradient required a number of gathers in various spaces, as determined by the analytic form of the gradient. In contrast, a finite differencing approach only requires the objective function in an analytic form; which in this case, is mostly just the rendering function itself. Therefore, I see two possible advantages to using a finite differencing technique. First, the rendering function is specifically constructed to run quickly in hardware, and should presumably be quite fast to compute. Second, it should be relatively easy to implement new models using a finite differencing approach, once the framework is in place.

Automatic differentiation refers to techniques that automatically generate analytical expressions for the gradient. This would be much more difficult to implement than finite differencing, but would produce a more accurate derivative evaluation. Automatic differentiation breaks the function down into elementary math operations, and applies the chain rule to build the analytic derivative. This is accomplished by either analyzing the source code, or by tracking the calculation as it progresses. Since the rendering algorithms are implemented by a combination of code on the host, which are traceable, and vertex and fragment shaders which are not directly traceable, a hybrid approach would perhaps be appropriate: code analysis for the shader, and runtime tracking using a combination of readbacks and analysis of execution on the host.

9.2.2 Hybrid Models

Many objects are often best represented by more than one kind of shader; however, image-based modeling typically focuses on a single model at a time. With a general approach, particularly one using finite differencing such as the one mentioned above, it becomes much more feasible to try different shader models in different areas of the object. In fact, shader selection could become part of the solution process.

9.2.3 Online Methods

The idea behind a stream is that the data can be thrown away once it has been processed. This was not particularly the case of nonlinear optimization, as it was important to use the same set of data at each iteration. However, the image-streaming framework is already in place for an *online* method. By this I mean a method where there is a continuous stream of new data. An examples is an online SVD method proposed by Brand et al. (Brand, 2003), which could be used to construct an LFM model from a continuous stream of data - for example a continuous image stream from a tracked video camera. Another advantage of an online method is that it can continuously adapt to a changing environment as well.

9.2.4 Geometry

Another limitation of this work is that is assumes known geometry. However, it would be interesting to introduce geometric parameters in the image-based modeling process. Others have already begun to pursue the simultaneous recovery of shape and non-specular surface reflectance models (Lee and Kuo, 1997; Yang et al., 2003; Hertzmann and Seitz, 2003; Jin et al., 2003; Yu et al., 2004), but have each chosen a particular reflectance model, typically an analytical model like the Phong or Torrance-Sparrow models. In the image-streaming framework, moving vertices of the model incurs no cost, as there is no preprocess as in the conventional approach. Therefore, I am proposing research for simultaneous recovery of shape and parameters of a shading model, but allowing for a greater variance in the kind of shading models applied.

Appendix A

Measuring Error in GPU Floating-Point Operations

Up until the late eighties, each computer vendor was left to develop their own conventions for floating-point computation as they saw fit. As a result, programmers needed to familiarize themselves with the peculiarities of each system in order to write effective software and evaluate numerical error. To alleviate this problem, a standard was established for floating-point computation, and CPU vendors now design to this standard (IEEE, 1987).

Today there is an interest in the use of graphics processing units, or GPUs, for non-graphics applications such as scientific computing. GPUs have floating-point representations similar to, and sometimes matching, the IEEE standard. However, GPUs do not adhere to IEEE standards for floating-point operations, nor do the vendors give the information necessary to establish bounds on error for these operations. Another complication is that floating-point behavior seems to be in a constant state of flux due to the dependence on the hardware, drivers, and compilers of a rapidly changing industry.

The goal is to determine the error bounds on floating-point operation results for quickly evolving graphics systems. I have created a tool to measure the error for four basic floating-point operations: addition, subtraction, multiplication and division. This work was originally presented as a poster at the *ACM Workshop on General Purpose Computing on Graphics Processors* in 2004, and is available in its proceedings (Hillesland and Lastra, 2004).

A.1 IEEE Standard Floating Point

Ideally, GPUs would follow the IEEE standard for floating-point operations. The IEEE standard provides a guarantee on error bounds for certain operations, including addition, subtraction, multiplication and division. It does so by requiring that these operations follow the *exact rounding* convention. Under this convention, the result of an operation must be the same as a result computed exactly, and then rounded according to an unambiguous rounding convention. In this discussion I focus on the default rounding mode, which is to *round to nearest*, which rounds the result to the nearest representable number, or to the closest even number in the case of a tie. This means a bound of [-0.5 , 0.5] in units of the last bit of the significand for the nearest rounding case.

A.2 Paranoia

Paranoia (Karpinski, 1985), originally written by William Kahan in the 1980's, explores a number of aspects of floating-point operation. I have adopted Paranoia's guard bit and rounding mode tests for subtraction, multiplication, and division. Of the GPUs I tested, all operations use guard bits.

Paranoia looks for two kinds of rounding: round to nearest, and chopping. Chopping could also be called "round to zero". The GPUs I tested did not follow either of these models. In order to obtain a bound on error, I turned to a more empirical approach, which I describe next.

A.3 GPUBench

GPUBench is a set of tests to evaluate the performance and precision characteristics of graphics hardware. It was also presented at the *ACM Workshop on General Purpose Computing on Graphics Processors* both as a poster and as described in an invited speech

Significand patterns:

- 1.00...010... (all zeros except a single one, or Schryer's "Type 1")
- 1.10...010... (same as previous, but with additional leading one)
- 1.11...101... (all ones except a single zero)
- 1.00...01... (zeros then ones)
- 1.10...01... (same as previous, but with additional leading one)
- 1.11...10... (ones then zeros, or Schryer's "Type 2")

Relative exponents: 0, -1, -(significand bits / 2), -significand bits

Table A.1: *Floating-Point Test Patterns.*

by Patrick Hanrahan in 2004. Precision was only measured for some unary transcendental functions. GPUBench tests the precision of transcendental functions by trying a vector of evenly spaced values in some range. It does not target particularly problematic values as was done for the more elementary functions tested here or in Paranoia (Karpinski, 1985), or in the work of Schryer (Schryer, 1981), which is described in the next section.

A.4 Measuring Floating-Point Error

To ensure complete error bounds in an empirical test would require exhaustive tests of all combinations of all floating-point numbers. Since this is impractical, I chose a subset of floating-point numbers that I believe does a reasonable job of characterizing the entire set. This is an approach used by others for testing correct operation of floating-point hardware. I used a superset of significands suggested by Schryer (Schryer, 1981). The test combinations include all the test cases in Paranoia, as well as cases that push the limits of round-off error and cases where the most work must be performed, such as extensive carry propagation. The significands and exponents used are given in Table A.1.

Table A.2 gives results for some example systems. For comparison, the IEEE-754 standard for floating-point operations using the round-to-nearest convention would have

Operation	R300/arbfp	NV30/fp30
Addition	[-1.000, 0.000]	[-1.000, 0.000]
Subtraction	[-1.000, 1.000]	[-0.750, 0.750]
Multiplication	[-0.989, 0.125]	[-0.782, 0.625]
Division	[-2.869, 0.094]	[-1.199, 1.375]

Table A.2: *Floating-Point Error in ULPs (Units in Last Place). Note that the R300 has a 16 bit significand, whereas the NV30 has 23 bits. Therefore one ULP on an R300 is equivalent to* 2^7 *ULPs on an NV30. Division is implemented by a combination of reciprocal and multiply on these systems. Cg version 1.2.1. ATI driver 6.14.10.6444. NVIDIA driver 56.72. arbfp and fp30 are architectures targetted by the Cg compiler, and represent the most sophisticated architectures supported for the respective hardware under Cg.*

error bars of [-0.5, 0.5] for all operations.

A.5 System Considerations

Results are for specific configurations of graphics card, driver, operating system, CPU, chipset, compiler version, and other factors. The tool is intended to be run each time any of these items change.

Semantics for programming GPUs currently allow for considerable leeway in how a program is implemented. Instructions can be re-ordered, subexpressions involving constants or “uniform” parameters may be evaluated on the CPU, and associative and distributive properties, which do not hold for floating-point operations, may be applied in optimization. The tool does not take into consideration the kinds of optimizations possible in larger program contexts.

A.6 Conclusion

The goal is to give the developer a tool to characterize GPU floating-point error in order to aid them in developing compute-intensive applications. An empirical approach is adopted to establish error bounds for addition, subtraction, multiplication and division. This tool has been applied to two contemporary graphics hardware systems, showing that although not perfect, they provide reasonable accuracy relative to the IEEE standard for floating-point operations within their limited precision support.

BIBLIOGRAPHY

Adelson, E. H. and Bergen, J. R. (1991). The plenoptic function and the elements of early vision. *M. Landy and J. A. Movshon, (eds) Computational Models of Visual Processing.*

Apodaca, A. A., Gritz, L., and Barsky, B. A. (1999). *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Clarendon Press.

Brand, M. (2003). Fast online svd revisions for lightweight recommender systems. In *Proceedings of SIAM International Conference on Data Mining*.

Buchholz, W., editor (1962). *Planning A Computer System*. McGraw-Hill.

Buck, I., Fatahalian, K., and Hanrahan, P. (2004). Gpubench: Evaluating gpu performance for numerical and scientific applications. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*.

Buck, I. and Purcell, T. J. (2004). A toolkit for computation on gpus. In Fernando, R., editor, *GPU Gems*, pages 621–636. Addison Wesley.

Carr, N. A. and Hart, J. C. (2002). Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph.*, 21(2):106–131.

Chen, W.-C. (2002). *Light Field Mapping: Efficient Representation of Surface Light Fields*. PhD thesis, University of North Carolina at Chapel Hill.

Chen, W.-C., Bouguet, J.-Y., Chu, M. H., and Grzeszczuk, R. (2002a). Light Field Mapping: Efficient Representation and Hardware Rendering of Surface Light Fields. *ACM Transactions on Graphics*, 21(3):447–456.

Chen, W.-C., Grzeszczuk, R., Molinov, S., Bouguet, J.-Y., Smirnov, A., and Simakov, D. (2002b). OpenLF.
`http://sourceforge.net/projects/openlf/`.

Cignoni, P., Montani, C., Scopigno, R., and Rocchini, C. (1998). A general method for preserving attribute values on simplified meshes. In *Proceedings of the conference on Visualization '98*, pages 59–66. IEEE Computer Society Press.

Cohen, M. F. and Wallace, J. R. (1993). *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, MA.

Dana, K. J., van Ginneken, B., Nayar, S. K., and Koenderink, J. J. (1999). Reflectance and texture of real-world surfaces. *ACM Trans. Graph.*, 18(1):1–34.

Debevec, P., Hawkins, T., Tchou, C., Duiker, H.-P., Sarokin, W., and Sagar, M. (2000). Acquiring the Reflectance Field of a Human Face. In Akeley, K., editor, *Proceedings of ACM SIGGRAPH 2000*, pages 145–156. ACM Press / ACM SIGGRAPH / Addison Wesley Longman.

Dennis, J. E. J. and Schnabel, R. B. (1996). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics, 16. SIAM.

Fournier, A. and Fussell, D. (1988). On the power of the frame buffer. *ACM Trans. Graph.*, 7(2):103–128.

Furukawa, R., Kawasaki, H., Ikeuchi, K., and Sakauchi, M. (2002). Appearance Based Object Modeling Using Texture Database: Acquisition, Compression and Rendering. *Eurographics Rendering Workshop 2002*.

Gill, P., Murray, W., and Wright, M. (1981). *Practical Optimization*. Academic Press.

Golub, G. H. and Loan, C. F. V. (1991). *Matrix Computation*. John Hopkins University Press. Second Edition.

Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F. (1996). The Lumigraph. In *Proceedings of ACM SIGGRAPH 1996*, pages 43–54, New Orleans.

Hertzmann, A. and Seitz, S. M. (2003). Shape and materials by example: A photometric stereo approach. In *CVPR (1)*, pages 533–540.

Hillesland, K. and Lastra, A. (2004). Gpu floating-point paranoia. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*.

Hillesland, K. E., Molinov, S., and Grzeszczuk, R. (2003). Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware. *ACM Transactions on Graphics and Proceedings of ACM SIGGRAPH 2003*, pages 925–934.

IEEE (1987). IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25.

Jin, H., Soatto, S., and Yezzi, A. J. (2003). Multi-view stereo beyond lambert. In *CVPR (1)*, pages 171–178.

Kahan, W. (1965). Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40.

Karpinski, R. (1985). Paranoia: A floating-point benchmark. *Byte Magazine*, 10(2):223–235.

Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., and Chang, A. (2001). Imagine: Media Processing with Streams. *IEEE Micro*, pages 35–46.

Knuth, D. E. (1998). *Fundamental Algorithms*, volume 2 of *The Art of Computer Programming*, chapter 4. Addison-Wesley, Reading, Massachusetts, third edition.

Lafortune, E. P. F., Foo, S.-C., Torrance, K. E., and Greenberg, D. P. (1997). Non-Linear Approximation of Reflectance Functions. In *Proceedings of ACM SIGGRAPH 1997*, pages 117–126.

Lee, D. D. and Seung, H. S. (1999). Learning the Parts of Objects by Non-Negative Matrix Factorization. *Nature*, 401:788–791.

Lee, K. M. and Kuo, C.-C. J. (1997). Shape from shading with a generalized reflectance map model. *Comput. Vis. Image Underst.*, 67(2):143–160.

Lensch, H. P., Kautz, J., Goesele, M., Heidrich, W., and Seidel, H.-P. (2001). Image-based reconstruction of spatially varying materials. In *Proceedings of the 12th Eurographics Workshop on Rendering*, pages 104–115.

Lensch, H. P. A., Lang, J., S, A. M., and Seidel, H.-P. (2003). Planned sampling of spatially varying BRDFs. In Brunet, P. and Fellner, D. W., editors, *EUROGRAPHICS 2003 (EUROGRAPHICS-03) : the European Association for Computer Graphics, 24th Annual Conference*, pages 473–482, Granada, Spain. Blackwell.

Levoy, M. and Hanrahan, P. (1996). Light Field Rendering. In *Proceedings of ACM SIGGRAPH 1996*, pages 31–42, New Orleans.

Lindholm, E., Kligard, M. J., and Moreton, H. (2001). A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, pages 149–158. ACM Press.

Marschner, S. R. (1998). *Inverse rendering for computer graphics*. PhD thesis, Cornell University.

McAllister, D. (2002). *A Generalized Surface Appearance Representation for Computer Graphics*. PhD thesis, University of North Carolina at Chapel Hill.

McAllister, D. K., Lastra, A., and Heidrich, W. (2002). Efficient rendering of spatial bi-directional reflectance distribution functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 79–88. Eurographics Association.

McCool, M. D., Ang, J., and Ahmad, A. (2001). Homomorphic Factorization of BRDFs for High-Performance Rendering. In Fiume, E., editor, *Proceedings of ACM SIGGRAPH 2001*, pages 171–178. ACM Press / ACM SIGGRAPH.

Message Passing Interface Forum (1995). *MPI: A Message-Passing Interface Standard, Version 1.1*.

Migdalas, A., Toraldo, G., and Kumar, V. (2003). Nonlinear optimization and parallel computing. *Parallel Comput.*, 29(4):375–391.

Miller, G. S. P., Rubin, S., and Ponceleon, D. (June 1998). Lazy Decompression of Surface Light Fields for Precomputed Global Illumination. *Eurographics Rendering Workshop 1998*, pages 281–292.

Müller, G., Meseth, J., Sattler, M., Sarlette, R., and Klein, R. (2004). Acquisition, synthesis and rendering of bidirectional texture functions. In *Eurographics*. Eurographics, Eurographics. State-of-the-Art Report.

Nicodemus, F. E., Richmond, J. C., Hsia, J. J., Ginsberg, I. W., and Limperis, T. (1977). Geometrical considerations and nomenclature for reflectance. *NBS Monograph 160*.

Nishino, K., Sato, Y., and Ikeuchi, K. (1999). Appearance Compression and Synthesis based on 3D Model for Mixed Reality. In *International Conference on Computer Vision*, pages 38–45.

Nocedal, J. and Wright, S. J. (1999). *Numerical Optimization*. Springer Series in Operations Research. Springer-Verlag New York, Inc.

Nusselt, W. (1928). Grapische bestimmung des winkelverhaltnisses bei der warmestrahlung. *Zeitshrift des Vereines Deutscher Ingenieure*, 72(20):673.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712.

Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P. (2003). Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50. Eurographics Association.

Ramamoorthi, R. and Hanrahan, P. (2001). A Signal-Processing Framework for Inverse Rendering. In Fiume, E., editor, *Proceedings of ACM SIGGRAPH 2001*, pages 117–128. ACM Press / ACM SIGGRAPH.

Sato, Y., Wheeler, M. D., and Ikeuchi, K. (1997). Object Shape and Reflectance Modeling from Observation. In *Proceedings of ACM SIGGRAPH 1997*, pages 379–388.

Schnabel, R. B. (1995). A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Comput.*, 21(6):875–905.

Schryer, N. L. (1981). A test of a computer's floating-point arithmetic unit. Technical Report Computer Science Technical Report 89, AT&T Bell Laboratories.

Segal, M. and Akeley, K. (1999). *The OpenGL Graphics System: A Specificaion (Version 1.2.1)*. The OpenGL Architecture Review Board.

Wilkinson, J. H. (1963). *Rounding Errors in Algebraic Processes*. Englewood Cliffs, NJ, USA.

Williams, L. (1978). Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274. ACM Press.

Wood, D. N., Azuma, D. I., Aldinger, K., Curless, B., Duchamp, T., Salesin, D. H., and Stuetzle, W. (July 2000). Surface Light Fields for 3D Photography. In *Proceedings of ACM SIGGRAPH 2000*, pages 287–296.

Yang, R. and Pollefeys, M. (2003). Multi-resolution real-time stero on commodity graphics hardware. In *CVPR*.

Yang, R., Pollefeys, M., and Welch, G. (2003). Dealing with textureless regions and specular highlignes – a progressive space carving scheme using a novel photo-consistency measure. In *ICCV*.

Yu, T., Xu, N., and Ahuja, N. (2004). Recovering shape and reflectance model of non-lambertian objects from multiple views. In *CVPR (2)*, pages 226–233.

Yu, Y., Debevec, P. E., Malik, J., and Hawkins, T. (1999). Inverse Global Illumination: Recovering Reflectance Models of Real Scenes From Photographs. In *Proceedings of ACM SIGGRAPH 1999*, pages 215–224.

Yu, Y. and Malik, J. (1998). Recovering Photometric Properties of Architectural Scenes from Photographs. In *Proceedings of ACM SIGGRAPH 1998*, pages 207–218.