ELSEVIER

MATT PHARR ○ GREG HUMPHREYS

PHYSICALLY BASED RENDERING

FROM THEORY TO IMPLEMENTATION

SERIES IN INTERACTIVE 3D TECHNOLOGY

Physically Based Rendering

FROM THEORY TO IMPLEMENTATION

MATT PHARR
NVIDIA

GREG HUMPHREYS
Department of Computer Science
University of Virginia

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Senior Editor Tim Cox
Publishing Services Manager Simon Crump
Project Editor Kyle Sarofeen
Project Management Elisabeth Beller
Editorial Coordinator Rick Camp
Cover Design Chen Design Associates, San Francisco
Cover Image scene modeled by Oliver Deussen and Bernd Lintermann; rendered by `pbrt`
Text Design Chen Design
Composition Windfall Software, using ZzTeX
Technical Illustration Dartmouth Publishing
Copyeditor Ken DellaPenta
Proofreader Jennifer McClain
Indexer Steve Rath
Interior and Cover Printer Hing Yip (China) Printing & Binding Co. Ltd.

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Library of Congress Cataloguing-in-Publication: applied for

ISBN: 0-12-553180-X

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com.

Printed in China
08 07 06 05 04 5 4 3 2 1

Foreword

Over the last 10 years, I've had the pleasure of teaching a course in image synthesis at Princeton and Stanford. That course provides a broad overview of the theory and practice of rendering, concentrating on the algorithms and techniques needed to make realistic imagery. Typically, it is the second course in computer graphics (assuming as a prerequisite an introductory computer graphics course) and is taken by upper-level undergraduates and first-year graduate students.

Like my colleagues around the world, I initially had the students build a ray tracer from scratch, step by step, over the term. Writing a ray tracer is an excellent way to learn computer graphics since it includes modules for geometry, materials, lighting, imaging—all the basic concepts. In fact, the process of writing a ray tracer has become a rite of passage for computer graphics students.

Eventually, however, a few problems with this approach became clear. First, everyone spent way too much time building support libraries. This included code for reading geometric file formats and for writing images to files. The wise students also invested time building solid utility libraries, like foundation classes for matrices, points, and vectors. At the end of the term I would ask them whether they thought writing a ray tracer was worth all the time. They all agreed they learned a lot by writing all this infrastructural code. However, they also complained that they had very little time for the more interesting aspects of rendering—the actual topics of the course. The fun parts that I wanted them to work on—like lighting simulation, participating media, photon mapping, and so on—were left behind.

Another problem was that while I had many students who wrote beautiful code, some students produced "spaghetti code." Once they got going in the wrong direction, it was very hard for them to recover. They learned after each new assignment that the way the previous assignment had been implemented may not have been such a good idea after all, since it was difficult to extend. Unfortunately, there was not enough time to go back and rewrite the original libraries. It became apparent that it was important that students learn how to develop systems that are well-structured, robust, and extensible—skills that often aren't explicitly taught.

Gradually over the years, I provided better and more complete support libraries so that students had more time to concentrate on the subject material. Then Matt and Greg, both former teaching assistants, had the idea of building a ray tracer using literate programming. One day they said to me, "Suppose we wrote a literate ray tracer. Would you use it in the course?" I said, "That's a great idea—sign me up!" The book before you documents pbrt, the sixth version of the system. We have used it since 2000 at Stanford.

The idea of literate programming was invented by Donald Knuth. He believed that programs were works of literature in the eyes of the nerdy beholder and could be presented to the reader as a book. He invented the system web for writing literate programs. Using web, the programmer uses a language like $\TeX$ for formatting the text and a programming language like C++ for writing the program. The program is embedded in the text as a graph of interconnected parts, which Knuth called a "web," similar in spirit to the way that web pages are interconnected using links. To compile and run the program, the program is extracted automatically from the text. Donald Knuth has published several books documenting $\TeX$ and Metafont as literate programs. Another great example of a literate program is lcc, the subject of the book *A Retargetable C Compiler* written by Christopher Fraser and David Hanson. They document a complete ANSI C compiler in the literate style. The lcc book in many ways is the model for this book.

Writing a literate program is a *lot* more work than writing a normal program. After all, who ever documents their programs in the first place!? Moreover, who documents them in a pedagogical style that is easy to understand? And finally, who ever provides commentary on the theory and design issues behind the code as they write the documentation? All of that is here in the pages that follow.

The real value of pbrt is not just that it is a well-documented ray tracer. There are two other important reasons to read this book.

First, computer graphics, and rendering in particular, is full of beautiful theory. The theory covers physical concepts, such as light fields and the interaction of light with different materials, and mathematical concepts, such as integral equations and Monte Carlo integration. The great thing about computers is that they allow us to build rendering systems based on the best theory. pbrt in essence turns the theory of image-making into a practical method for creating images. Each physical process or mathematical abstraction is translated into a class, with methods that implement algorithms that model the corresponding concept. There is a direct correspondence between the architecture of the system and the theory. Good theory thus leads to good architectures. Seeing in detail how theory is transformed into practice is perhaps the strongest reason for reading this book.

Second, it is really, really hard to build a good rendering system. The best system designers pay attention to every detail. They are craftsmen who polish every pixel, prevent every crack between triangles, and stamp out every aliasing and quantization artifact. They also build robust systems that handle every case, not just the easy ones. Choosing algorithms that work is an art in itself. Finally, there are many subtle issues in handling large models (by conserving memory in the right places) and efficient execution (optimizing the right code) that only come from experience building and using real systems. There are not many opportunities to learn about these issues, and seeing how they have been solved by others is one of the best ways to do so.

Matt and Greg are skilled programmers. Matt worked with the RenderMan team at Pixar, is one of the cofounders of Exluna, and is a designer and implementor of their Entropy renderer. Greg has also developed many systems, including WireGL and Chromium, a large open-source project to support parallel rendering. Greg also wrote the tangling and untangling software used to produce this book. They are both master craftsmen who will convert you from a rendering journeyman to a rendering master.

Pat Hanrahan
Canon USA Professor
Stanford University

Preface

[Just as] other information should be available to those who want to learn and understand, program source code is the only means for programmers to learn the art from their predecessors. It would be unthinkable for playwrights not to allow other playwrights to read their plays [or to allow them] at theater performances where they would be barred even from taking notes. Likewise, any good author is well read, as every child who learns to write will read hundreds of times more than it writes. Programmers, however, are expected to invent the alphabet and learn to write long novels all on their own. Programming cannot grow and learn unless the next generation of programmers has access to the knowledge and information gathered by other programmers before them. —Erik Naggum

Rendering is a fundamental component of computer graphics. At the highest level of abstraction, rendering is the process of converting a description of a three-dimensional scene into an image. Algorithms for animation, geometric modeling, texturing, and other areas of computer graphics all must pass their results through some sort of rendering process so that they can be made visible in an image. Rendering has become ubiquitous; from movies to games and beyond, it has opened new frontiers for creative expression, entertainment, and visualization.

In the early years of the field, research in rendering focused on solving fundamental problems such as determining which objects are visible from a given viewpoint. As effective solutions to these problems have been found and as richer and more realistic scene descriptions have become available thanks to continued progress in other areas of graphics, modern rendering has grown to include ideas from a broad range of disciplines, including physics and astrophysics, astronomy, biology, psychology and the study of perception, and pure and applied mathematics. The interdisciplinary nature of rendering is one of the reasons that it is such a fascinating area of study.

This book presents a selection of modern rendering algorithms through the documented source code for a complete rendering system. All of the images in this book, including the one on the front and back covers, were rendered by this software. All of the algorithms that came together to generate these images are described in these pages. The system, `pbrt`, is written using a programming methodology called *literate programming* that mixes prose describing the system with the source code that implements it. We believe that the literate programming approach is a valuable way to introduce ideas in computer graphics and computer science in general. Often, some of the subtleties of an algorithm can be unclear or hidden until it is implemented, so seeing an actual implementation is a good way to acquire a solid understanding of that algorithm's details. Indeed, we believe that deep understanding of a small number of algorithms in this manner provides a stronger base for further study of computer graphics than does superficial understanding of many.

In addition to clarifying how an algorithm is implemented in practice, presenting these algorithms in the context of a complete and nontrivial software system also allows us to address issues in the design and implementation of medium-sized rendering systems. The design of a rendering system's basic abstractions and interfaces has substantial implications for both the elegance of the implementation and the ability to extend it later, yet the trade-offs in this design space are rarely discussed.

`pbrt` and the contents of this book focus exclusively on *photorealistic rendering*, which can be defined variously as the task of generating images that are indistinguishable from those that a camera would capture in a photograph, or as the task of generating images that evoke the same response from a human observer as looking at the actual scene. There are many reasons to focus on photorealism. Photorealistic images are crucial for the movie special-effects industry because computer-generated imagery must often be mixed seamlessly with footage of the real world. In entertainment applications where all of the imagery is synthetic, photorealism is an effective tool for making the observer forget that he or she is looking at an environment that does not actually exist. Finally, photorealism gives a reasonably well-defined metric for evaluating the quality of the rendering system's output.

A consequence of our approach is that this book and the system it describes do not exhaustively cover the state of the art in rendering; many interesting topics in photorealistic rendering will not be introduced either because they don't fit well with the architecture of the software system (e.g., finite-element radiosity algorithms) or because we believed that the pedagogical value of explaining the algorithm was outweighed by the complexity of its implementation (e.g., Metropolis light transport). We will note these decisions as they come up and provide pointers to further resources so that the reader can follow up on topics of interest. Many other areas of rendering, including interactive rendering, visualization, and illustrative forms of rendering such as pen-and-ink styles, aren't covered in this book at all. Nevertheless, many of the algorithms and ideas in this system (e.g., algorithms for texture map anti-aliasing) are applicable to a wider set of rendering styles.

AUDIENCE

Our primary intended audience for this book is students in graduate or upper-level undergraduate computer graphics classes. This book assumes existing knowledge of computer graphics at the level of an introductory college-level course, although certain key concepts such as basic vector geometry and transformations will be reviewed here. For students who do not have experience with programs that have tens of thousands of lines of source code, the literate programming style gives a gentle introduction to this complexity. We pay special attention to explaining the reasoning behind some of the key interfaces and abstractions in the system in order to give these readers a sense of why the system is structured in the way that it is.

Our secondary, but equally important, audiences are advanced graduate students and researchers, software developers in industry, and individuals interested in the fun of writing their own rendering systems. Although many of the ideas in this book will likely be familiar to these readers, seeing explanations of the algorithms presented in the literate style may provide new perspectives. `pbrt` includes implementations of a number of newer and/or difficult-to-implement algorithms and techniques, such as subdivision surfaces, Monte Carlo light transport, and volumetric scattering models; these should be of particular interest to experienced practitioners in rendering. We hope that delving into one particular organization of a complete and nontrivial rendering system will also be thought provoking to this audience.

OVERVIEW AND GOALS

`pbrt` is based on the *ray-tracing* algorithm. Ray tracing is an elegant technique that has its origins in lens making; Carl Freidrich Gauss traced rays through lenses by hand in the 19th century. Ray-tracing algorithms on computers follow the path of infinitesimal rays of light through the scene until they intersect a surface. This approach gives a simple method for finding the first visible object as seen from any particular position and direction, and is the basis for many rendering algorithms.

`pbrt` was designed and implemented with three main goals in mind: it should be *complete*, it should be *illustrative*, and it should be *physically based*.

Completeness implies that the system should not lack key features found in high-quality commercial rendering systems. In particular, it means that important practical issues, such as antialiasing, robustness, and the ability to efficiently render complex scenes, should all be addressed thoroughly. It is important to consider these issues from the start of the system's design, since these features can have subtle implications for all components of the system and can be quite difficult to retrofit into the system at a later stage of implementation.

Our second goal means that we tried to choose algorithms, data structures, and rendering techniques with care and with an eye toward readability and clarity. Since their implementations will be examined by more readers than is the case for many other rendering systems, we tried to select the most elegant algorithms that we were aware of and implement them as well as possible. This goal also required that the system be small enough for a single person to understand completely. We have implemented `pbrt` using a plug-in architecture, with only a small core of basic glue, and as much of the functionality as possible in external modules. The result is that one doesn't need to understand all of the various plug-ins in order to understand the basic structure of the system. This makes it easier to delve deeply into parts of interest and skip others, without losing sight of how the overall system fits together.

There is a tension between the two goals of being complete and being illustrative. Implementing and describing every possible useful technique would not only make this book extremely long, but also would make the system prohibitively complex for most readers. In cases where `pbrt` lacks a particularly useful feature, we have attempted to design the architecture so that the feature could be added without altering the overall system design.

The basic foundations for physically based rendering are the laws of physics and their mathematical expression. `pbrt` was designed to use the correct physical units and concepts for the quantities it computes and the algorithms it implements. When configured to do so, `pbrt` can compute images that are *physically correct;* they accurately reflect the lighting as it would be in a real-world version of the scene. One advantage of the decision to use a physical basis is that it gives a concrete standard of program correctness: for simple scenes, where the expected result can be computed in closed form, if `pbrt` doesn't compute the same result, we know there must be a bug in the implementation. Similarly, if different physically based lighting algorithms in `pbrt` give different results for the same scene, or if `pbrt` doesn't give the same results as another physically based renderer, there is certainly an error in one of them. Finally, we believe that this physically based approach to rendering is valuable because it is rigorous. When it is not clear how a particular computation should be performed, physics gives an answer that guarantees a consistent result.

Efficiency was given lower priority than these three goals. Since rendering systems often run for many minutes or hours in the course of generating an image, efficiency is clearly important. However, we have mostly confined ourselves to *algorithmic* efficiency rather than low-level code optimization. In some cases, obvious micro-optimizations take a backseat to clear, well-organized code, although we did make some effort to optimize the parts of the system where most of the computation occurs. For this reason, as well as to ensure portability, `pbrt` is not presented as a parallel or multithreaded application, although parallelizing `pbrt` would not be very difficult.

In the course of presenting `pbrt` and discussing its implementation, we hope to convey some hard-learned lessons from years of rendering research and development. There is more to writing a good renderer than stringing together a set of fast algorithms; making the system both flexible and robust is a difficult task. The system's performance must degrade gracefully as more geometry or light sources are added to it, or as any other axis of complexity is pushed. Numeric stability must be handled carefully, and algorithms that don't waste floating-point precision are critical.

The rewards for developing a system that addresses all these issues are enormous–it is a great pleasure to write a new renderer or add a new feature to an existing renderer and use it to create an image that couldn't be generated before. Our most fundamental goal in writing this book was to bring this opportunity to a wider audience. Readers are encouraged to use the system to render the example scenes on the companion CD as they progress through the book. Exercises at the end of each chapter suggest modifications to the system that will help clarify its inner workings, and more complex projects to extend the system by adding new features.

The Web site for this book is located at *www.pbrt.org*. There we will post errata and bug fixes, updates to `pbrt`'s source code, additional scenes to render, supplemental utilities, and new plug-in modules. Any bugs in `pbrt` or errors in this text that are not listed at the Web site can be reported to the email address *bugs@pbrt.org*. We greatly value your feedback!

ACKNOWLEDGMENTS

Pat Hanrahan has contributed to this book in more ways than we could hope to acknowledge; we owe a profound debt to him. He tirelessly argued for clean interfaces and finding the right abstractions to use throughout the system, and his understanding of and approach to rendering deeply influenced its design. His willingness to use `pbrt` and this manuscript in his rendering course at Stanford was enormously helpful, particularly in the early years of its life when it was still in very rough form; his feedback throughout this process has been crucial for bringing the text to its current state. Finally, the group of people that Pat helped assemble at the Stanford Graphics Lab, and the open environment that he fostered, made for an exciting, stimulating, and fertile environment. We feel extremely privileged to have been there.

Marc Levoy, Leonidas Guibas, and Ron Fedkiw were similarly instrumental in making Stanford such an extraordinary place to study. We thank them for everything they have done to create this truly unique place.

We owe a debt of gratitude to the many students who used early drafts of this book in courses at Stanford and the University of Virginia between 1999 and 2004. These students provided an enormous amount of feedback about the book and `pbrt`. The teaching assistants for these courses deserve special mention: Tim Purcell, Mike Cammarano, Ian Buck, and Ren Ng at Stanford, and Nolan Goodnight at Virginia. A number of students in those classes gave particularly valuable feedback and sent bug reports and bug fixes; we would especially like to thank Evan Parker and Phil Beatty. A draft of the manuscript of this book was used in classes taught by Bill Mark and Don Fussell at the University of Texas, Austin, and Raghu Machiraju at Ohio State University; their feedback was invaluable, and we are grateful for their adventurousness in incorporating this system into their courses, even while it was still being edited and revised.

Matt Pharr would like to specifically acknowledge colleagues and coworkers in rendering-related endeavors who have been a great source of education and who have substantially influenced his approach to writing renderers and his understanding of the field: notably, Tony Apodaca, Tom Duff, Doug Epps, Reid Gershbein, Larry Gritz, Craig Kolb, Tom Lokovic, Mark VandeWettering, and Eric Veach. Particular thanks go to Craig Kolb, who provided a cornerstone of Matt's early computer graphics education through the freely available source code to the `rayshade` ray-tracing system and who has been a great colleague to have the opportunity to work with through a series of rendering research projects and commercial adventures. Eric Veach has also been generous with his time and expertise; his insistence on the importance of continuing to work on problems and software designs until robust, clean, and correct solutions have been found has been an influential lesson. Thanks also to Doug Shult and Stan Eisenstat for formative lessons in mathematics and computer science during high school and college, respectively, and most importantly to Matt's parents, for the education they've provided and continued encouragement along the way. Finally, thanks also to Nick Triantos, Jayant Kolhe, and NVIDIA for their understanding and support through the final stages of this project.

Greg Humphreys is very grateful to all the professors and TAs who tolerated him when he was an undergraduate at Princeton. Many people encouraged his interest in graphics, specifically Michael Cohen, David Dobkin, Adam Finkelstein, Michael Cox, Gordon Stoll, Patrick Min, and Dan Wallach. Doug Clark, Steve Lyon, and Andy Wolfe also supervised various independent research boondoggles without even laughing once. Once, in a group meeting about a year-long robotics project, Steve Lyon became exasperated and yelled, "Stop telling me why it can't be done, and figure out how to do it!"—an impromptu lesson that will never be forgotten. Eric Ristad fired Greg as a summer research assistant after his freshman year (before the summer even began), pawning him off on an unsuspecting Pat Hanrahan and beginning an advising relationship that would span 10 years and both coasts. Finally, Dave Hanson taught Greg that literate programming was a great way to work, and that computer programming can be a beautiful and subtle art form.

We would also like to thank the many friends and colleagues who made the Stanford Graphics Lab such a rich environment to work in and such a fun place to be: Maneesh Agrawala, Ian Buck, Milton Chen, Brian Curless, James Davis, Matthew Eldridge, Brian Freyburger, Chase Garfinkle, Reid Gershbein, John Gerth, François Guimbretière, Olaf Hall-Holt, Chris Holt, Mike Houston, Homan Igehy, Henrik Wann Jensen, Brad Johanson, Craig Kolb, Venkat Krishnamurthy, Phil Lacroute, Tolis Lerios, Bill Lorensen, Bill Mark, Steve Marschner, Tamara Munzner, Ren Ng, John Owens, Hans Pedersen, Kekoa Proudfoot, Tim Purcell, Jonathan Ragan-Kelley, Ravi Ramamoorthi, Szymon Rusinkiewicz, Philipp Slusallek, Jeff Solomon, Gordon Stoll, Maureen Stone, Diane Tang, and Li-Yi Wei.

We are also grateful to Don Mitchell, for his help with understanding some of the details of sampling and reconstruction; Thomas Kollig and Alexander Keller, for explaining the finer points of low-discrepancy sampling; and Dave Eberly, "Just d'FAQs," Hans-Bernhard Broeker, Steve Westin, and Gernot Hoffmann, for many interesting threads on *comp.graphics.algorithms*. Christer Ericson had a number of suggestions for improving our kd-tree implementation and generously made available a chapter from his upcoming book on collision detection that pointed us to a number of techniques that helped improve our implementation.

Many people and organizations have generously in supplyied us with scenes and models for use in this book and the accompanying CD. Their generosity has been invaluable in helping us create interesting example images throughout the text. The bunny, Buddha, and dragon models are courtesy of the Stanford Computer Graphics Laboratory's scanning repository at *graphics.stanford.edu/data/3Dscanrep/*. The ecosystem scene was created by Oliver Deussen and Bernd Lintermann for a paper by them and collaborators (Deussen, Hanrahan, Lintermann, Mech, Pharr, and Prusinkiewicz 1998). The "killeroo" model is included with permission of Phil Dench and Martin Rezard (3D scan and digital representations by headus, design and clay sculpt by Rezard). The physically accurate smoke data sets were created by Duc Nguyen and Ron Fedkiw. Nolan Goodnight created environment maps with a realistic skylight model. Finally, the Cornell Program of Computer Graphics Light Measurement Laboratory allowed us to include measured BRDF data.

We would especially like to thank Marko Dabrovic and Mihovil Odak at RNA Studios, *www.rna.hr*, who supplied us with a bounty of excellent models and scenes, including the Sponza atrium, the Sibenik cathedral, the Audi TT car model, and others that we were unable to incorporate into the book by the publication deadline.

We would also like to thank the book's reviewers, all of whom had insightful and constructive feedback about the manuscript at various stages of its progress: Ian Ashdown, Per Christensen, Doug Epps, Dan Goldman, Eric Haines, Erik Reinhard, Pete Shirley, Peter-Pike Sloan, Greg Ward, and a host of anonymous reviewers.

Finally, we would also like to thank Tim Cox (senior editor), for his willingness to take on this slightly unorthodox project and for both his direction and patience throughout the process. We are very grateful to Elisabeth Beller (project manager), who has gone well beyond the call of duty for this book; her ability to keep this complex project in control and on schedule has been remarkable, and we particularly thank her for the measurable impact she has had on the quality of the final result. Thanks also to Rick Camp (editorial assistant) for his many contributions along the way. Paul Anagnostopoulos and Jacqui Scarlott at Windfall Software did the book's composition; their ability to take the authors' homebrew literate programming file format and turn it into high-quality final output while also juggling the multiple unusual types of indexing we asked for is greatly appreciated. Thanks also to Ken DellaPenta (copyeditor) and Jennifer McClain (proofreader) as well as to Max Spector at Chen Design (text and cover designer), and Steve Rath (indexer).

ADDITIONAL READING

Donald Knuth's article *Literate Programming* (Knuth 1984) describes the main ideas behind literate programming as well as his `web` programming environment. The seminal TEX typesetting system was written with `web` and has been published as a series of books (Knuth 1986, Knuth 1993a). More recently, Knuth has published a collection of graph algorithms in literate format in *The Stanford GraphBase* (Knuth 1993b). These programs are enjoyable to read and are excellent presentations of their respective algorithms. The Web site *www.literateprogramming.com* has pointers to many articles about literate programming, literate programs to download, and a variety of literate programming systems; many refinements have been made since Knuth's original development of the idea.

The only other literate program we know of that has been published as a book is the implementation of the `lcc` compiler, which was written by Christopher Fraser and David Hanson and published as *A Retargetable C Compiler: Design and Implementation* (Fraser and Hanson 1995).

Contents

* An asterisk denotes a section with advanced content that can be skipped on a first reading.